Alfred Arnold

Makroassembler AS V1.42

Benutzeranleitung

Stand März 2014

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. .

PicoBlaze ist ein eingetragenes Warenzeichen der Xilinx Inc.

UNIX ist ein eingetragenes Warenzeichen der Open Group.

Linux ist ein eingetragenes Warenzeichen von Linus Thorvalds.

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 **9**

1.1 Lizenzbedingungen 9
1.2 allgemeine Fähigkeiten des Assemblers 11
1.3 Unterstützte Plattformen 16

2 Benutzung des Assemblers **19**

2.1 Hardware-Anforderungen 19
2.2 Lieferumfang 20
2.3 Installation 21
2.4 Aufruf, Parameter 26
2.5 Format der Eingabedateien 35
2.6 Format des Listings 37
2.7 Symbolkonventionen 39
2.8 Temporäre Symbole 41
2.8.1 Temporäre Symbole mit Namen 42
2.8.2 Zusammengesetzte temporäre Symbole 44
2.9 Formelausdrücke 45
2.9.1 Integerkonstanten 45
2.9.2 Gleitkommakonstanten 47
2.9.3 Stringkonstanten 47
2.9.4 Evaluierung 49
2.9.5 Operatoren 49
2.9.6 Funktionen 49
2.10 Vorwärtsreferenzen und andere Desaster 53
2.11 Registersymbole 56
2.12 Sharefile 57
2.13 Prozessor-Aliasse 58

3 Pseudobefehle **61**
3.1 Definitionen . . . 61
3.1.1 SET, EQU und CONSTANT . . . 61
3.1.2 SFR und SFRB . . . 62
3.1.3 XSFR und YSFR . . . 63
3.1.4 LABEL . . . 63
3.1.5 BIT . . . 64
3.1.6 DBIT . . . 65
3.1.7 PORT . . . 65
3.1.8 REG und NAMEREG . . . 65
3.1.9 LIV und RIV . . . 66
3.1.10 CHARSET . . . 66
3.1.11 CODEPAGE . . . 67
3.1.12 ENUM . . . 68
3.1.13 PUSHV und POPV . . . 68
3.2 Codebeeinflussung . . . 69
3.2.1 ORG . . . 69
3.2.2 RORG . . . 74
3.2.3 CPU . . . 74
3.2.4 SUPMODE, FPU, PMMU . . . 88
3.2.5 FULLPMMU . . . 88
3.2.6 PADDING . . . 89
3.2.7 PACKING . . . 89
3.2.8 MAXMODE . . . 90
3.2.9 EXTMODE und LWORDMODE . . . 90
3.2.10 SRCMODE . . . 90
3.2.11 BIGENDIAN . . . 91
3.2.12 WRAPMODE . . . 91
3.2.13 SEGMENT . . . 92
3.2.14 PHASE und DEPHASE . . . 93
3.2.15 SAVE und RESTORE . . . 94
3.2.16 ASSUME . . . 95
3.2.17 EMULATED . . . 102
3.2.18 BRANCHEXT . . . 103
3.3 Datendefinitionen . . . 103
3.3.1 DC[.size] . . . 104
3.3.2 DS[.size] . . . 105
3.3.3 DB,DW,DD,DQ & DT . . . 105
3.3.4 DS, DS8 . . . 107
3.3.5 BYT oder FCB . . . 107
3.3.6 BYTE . . . 107

3.3.7 DC8 . . . 108
3.3.8 ADR oder FDB . . . 108
3.3.9 WORD . . . 108
3.3.10 DW16 . . . 108
3.3.11 LONG . . . 108
3.3.12 SINGLE, DOUBLE und EXTENDED . . . 109
3.3.13 FLOAT und DOUBLE . . . 109
3.3.14 EFLOAT, BFLOAT, TFLOAT . . . 109
3.3.15 Qxx und LQxx . . . 110
3.3.16 DATA . . . 110
3.3.17 ZERO . . . 110
3.3.18 FB und FW . . . 111
3.3.19 ASCII und ASCIZ . . . 111
3.3.20 STRING und RSTRING . . . 111
3.3.21 FCC . . . 111
3.3.22 DFS oder RMB . . . 112
3.3.23 BLOCK . . . 112
3.3.24 SPACE . . . 112
3.3.25 RES . . . 112
3.3.26 BSS . . . 112
3.3.27 DSB und DSW . . . 113
3.3.28 DS16 . . . 113
3.3.29 ALIGN . . . 113
3.3.30 LTORG . . . 113
3.4 Makrobefehle . . . 114
3.4.1 MACRO . . . 114
3.4.2 IRP . . . 120
3.4.3 IRPC . . . 120
3.4.4 REPT . . . 121
3.4.5 WHILE . . . 121
3.4.6 EXITM . . . 122
3.4.7 SHIFT . . . 122
3.4.8 MAXNEST . . . 123
3.4.9 FUNCTION . . . 123
3.5 Strukturen . . . 125
3.5.1 Definition . . . 125
3.5.2 Nutzung . . . 126
3.5.3 geschachtelte Strukturen . . . 127
3.5.4 Unions . . . 127
3.5.5 Strukturen und Sektionen . . . 127
3.5.6 Strukturen und Makros . . . 127

3.6 bedingte Assemblierung . . . 128
3.6.1 IF / ELSEIF / ENDIF . . . 128
3.6.2 SWITCH / CASE / ELSECASE / ENDCASE . . . 129
3.7 Listing-Steuerung . . . 131
3.7.1 PAGE . . . 131
3.7.2 NEWPAGE . . . 132
3.7.3 MACEXP . . . 132
3.7.4 LISTING . . . 133
3.7.5 PRTINIT und PRTEXIT . . . 133
3.7.6 TITLE . . . 134
3.7.7 RADIX . . . 135
3.7.8 OUTRADIX . . . 135
3.8 lokale Symbole . . . 135
3.8.1 Grunddefinition (SECTION/ENDSECTION) . . . 136
3.8.2 Verschachtelung und Sichtbarkeitsregeln . . . 138
3.8.3 PUBLIC und GLOBAL . . . 140
3.8.4 FORWARD . . . 141
3.8.5 Geschwindigkeitsaspekte . . . 142
3.9 Diverses . . . 142
3.9.1 SHARED . . . 142
3.9.2 INCLUDE . . . 143
3.9.3 BINCLUDE . . . 144
3.9.4 MESSAGE, WARNING, ERROR und FATAL . . . 144
3.9.5 READ . . . 145
3.9.6 RELAXED . . . 146
3.9.7 END . . . 147

4 Prozessorspezifische Hinweise 149
4.1 6811 . . . 149
4.2 PowerPC . . . 151
4.3 DSP56xxx . . . 151
4.4 H8/300 . . . 152
4.5 SH7000/7600/7700 . . . 152
4.6 MELPS-4500 . . . 155
4.7 6502UNDOC . . . 155
4.8 MELPS-740 . . . 159
4.9 MELPS-7700/65816 . . . 160
4.10 M16 . . . 163
4.11 4004/4040 . . . 164
4.12 MCS-48 . . . 164
4.13 MCS-51 . . . 165

4.14 MCS-251 . . . 165
4.15 8085UNDOC . . . 168
4.16 8086..V35 . . . 170
4.17 8X30x . . . 172
4.18 XA . . . 173
4.19 AVR . . . 174
4.20 Z80UNDOC . . . 175
4.21 Z380 . . . 176
4.22 Z8 und eZ8 . . . 176
4.23 TLCS-900(L) . . . 178
4.24 TLCS-90 . . . 181
4.25 TLCS-870 . . . 182
4.26 TLCS-47 . . . 182
4.27 TLCS-9000 . . . 183
4.28 29xxx . . . 183
4.29 80C16x . . . 184
4.30 PIC16C5x/16C8x . . . 186
4.31 PIC17C4x . . . 187
4.32 ST6 . . . 187
4.33 ST7 . . . 188
4.34 ST9 . . . 189
4.35 6804 . . . 189
4.36 TMS3201x . . . 190
4.37 TMS320C2x . . . 191
4.38 TMS320C3x . . . 191
4.39 TMS9900 . . . 192
4.40 TMS70Cxx . . . 192
4.41 TMS370xxx . . . 193
4.42 MSP430 . . . 194
4.43 COP8 & SC/MP . . . 194
4.44 SC144xxx . . . 195
4.45 75K0 . . . 196
4.46 78K0 . . . 197
4.47 78K2 . . . 197
4.48 uPD772x . . . 197
4.49 F2MC16L . . . 198

5 Dateiformate **199**
5.1 Code-Dateien . . . 199
5.2 Debug-Dateien . . . 202

6 Hilfsprogramme **207**
6.1 PLIST . . . 208
6.2 BIND . . . 209
6.3 P2HEX . . . 209
6.4 P2BIN . . . 214
6.5 AS2MSG . . . 215

A Fehlermeldungen von AS **217**

B E/A-Fehlermeldungen **257**

C Häufig gestellte Fragen **261**

D Pseudobefehle gesammelt **265**

E Vordefinierte Symbole **279**

F Mitgelieferte Includes **283**
F.1 BITFUNCS.INC . . . 283
F.2 CTYPE.INC . . . 284

G Danksagungen **287**

H Änderungen seit Version 1.3 **289**

I Hinweise zum Quellcode von AS **305**
I.1 Verwendete Sprache . . . 305
I.2 Abfangen von Systemabhängigkeiten . . . 306
I.3 Systemunabhängige Dateien . . . 307
I.3.1 Von AS genutzte Module . . . 307
I.3.2 Zusätzliche Module für die Hilfsprogramme . . . 312
I.4 Während der Erzeugung von AS gebrauchte Module . . . 313
I.5 Generierung der Nachrichtendateien . . . 315
I.5.1 Format der Quelldateien . . . 315
I.6 Dokumentationserzeugung . . . 317
I.7 Testsuite . . . 318
I.8 Einhängen eines neuen Zielprozessors . . . 319
I.9 Lokalisierung auf eine neue Sprache . . . 325

Kapitel 1

Allgemeines

Diese Anleitung wendet sich an Leute, die bereits in Assembler programmiert haben 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 Programmes 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:

- Hardware:
  - Prozessortyp (mit/ohne Koprozessor)
  - Speicherausbau
  - Grafikkarte
  - Festplatte und Typ deren Interfaces
- Software:
  - Betriebssystem (MS/DR/Novell-DOS, OS/2, Windows) und Version
  - installierte speicherresidente Programme
  - benutzte Version von AS + Datum des EXE-Files
- möglicht 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 (Eilfschornsteinstraße 16, Keller Philosophengebäude Rückseite).

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

Die neueste Version von AS (DOS, DPMI, OS/2, C) findet sich auf folgendem Server:

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

oder auch kurz

```
http://www.alfsembler.de
```

Die Quellen der C-Version können weiterhin von folgendem Server geholt werden:

```
sunsite.unc.edu
  Verzeichnis pub/Linux/devel/lang/assemblers/asl-<version>.tar.gz
```

...und damit natürlich von jedem Sunsite-Spiegel der Welt!

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 inkl. Koprozessor und MMU
- Motorola ColdFire
- Motorola DSP5600x,DSP56300
- Motorola/IBM MPC601/MPC505/PPC403/MPC821
- Motorola M-Core
- Motorola 6800, 68(HC)11(K4) sowie Hitachi 6301
- Motorola/Freescale 6805, 68HC(S)08
- Motorola 6809 / Hitachi 6309
- Motorola/Freescale 68HC12(X) inklusive XGATE

- Freescale 68RS08
- Motorola 68HC16
- Hitachi H8/300(H)
- Hitachi H8/500
- Hitachi SH7000/7600/7700
- Rockwell 6502 und 65(S)C02
- CMD 65816
- Mitsubishi MELPS-740
- Mitsubishi MELPS-7700
- Mitsubishi MELPS-4500
- Mitsubishi M16
- Mitsubishi M16C
- Intel 4004/4040
- Intel MCS-48/41
- 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, Z180, Z380

- Zilog Z8, eZ8
- Xilinx KCPSM/KCPSM3 ('PicoBlaze')
- LatticeMico8
- Toshiba TLCS-900(L)
- Toshiba TLCS-90
- Toshiba TLCS-870
- Toshiba TLCS-47
- Toshiba TLCS-9000
- Microchip PIC16C54..16C57
- Microchip PIC16C84/PIC16C64
- Microchip PIC17C42
- SGS-Thomson ST6
- SGS-Thomson ST7
- SGS-Thomson ST9
- SGS-Thomson 6804
- Texas Instruments TMS32010/32015
- Texas Instruments TMS3202x
- Texas Instruments TMS320C3x
- Texas Instruments TMS320C20x/TMS320C5x
- Texas Instruments TMS320C54x
- Texas Instruments TMS320C6x
- Texas Instruments TMS9900
- Texas Instruments TMS7000
- Texas Instruments TMS370xxx
- Texas Instruments MSP430

- National Semiconductor SC/MP
- National Semiconductor INS807x
- National Semiconductor COP4
- National Semiconductor COP8
- National Semiconductor SC144xx
- Fairchild ACE
- NEC μPD78(C)1x
- NEC μPD75xx
- NEC μPD75xxx (alias 75K0)
- NEC 78K0
- NEC 78K2
- NEC μPD7720/7725
- NEC μPD77230
- Fujitsu F^{2}MC8L
- Fujitsu F^{2}MC16L
- Symbios Logic SYM53C8xx (ja, die kann man programmieren!)
- Intersil CDP1802/1805
- XMOS XS1

in Arbeit / Planung / Überlegung :

- NEC 78K4
- Intel 8008
- Analog Devices ADSP21xx
- SGS-Thomson ST20
- Texas Instruments TMS320C4x

- Texas Instruments TMS320C8x
- Toshiba TC9331

Noch gesucht werden Unterlagen für:

- die ganze Palette der OKI-Controller

ungeliebt, aber *doch* vorhanden :

- Intel 8086, 80186, NEC V30&V35 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 Codeformat, 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 Zwischergebnisse bei Formeln — mit einer Breite von 32 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 Includefiles 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 Selberübersetzen 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

OS 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):

PMI 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...

???

Wer die bisherige Aufzählung liest, wird feststellen, daß das meistverkaufte Betriebssystem der Welt aus Redmont in dieser Aufzählung fehlt. Wer mich persönlich kennt, weiß, daß ich Windows (egal, ob 3.X, 95 oder NT) *nicht* für das Ei des Kolumbus halte. Kurzgesagt, ich bin ein ,,Windows-Hasser". Auch wenn eine große Zahl an Leuten diese Einstellung für überholt bis lächerlich erachten und mir jetzt vorhalten, ich würde hier einem großen Teil potentieller Anwender AS vorenthalten, so werden sie sich doch damit abfinden müssen: Ich treibe die Entwicklung an AS primär weiter, weil sie mir *Spaß* macht; AS ist ein nicht-kommerzielles Projekt und ich nehme mir deswegen die Freiheit, nicht auf potentielle Marktanteile zu schielen. Ich suche mir die Plattformen aus, auf denen das Programmieren *mir* Spaß macht, und Programmieren unter Windows macht mir definitiv keinen Spaß! Ich habe übrigens durchaus schon einmal Windows-Programme schreiben müssen, es ist also nicht so, daß ich hier ohne Erfahrung etwas daherreden würde. Sofern irgendjemand AS in diese Richtung portieren will, werde ich mich ihm nicht in den Weg stellen, über die Sourcen hinaus hat er aber nicht viel Hilfe von mir zu erwarten (und muß sich selber mit den Anfragen der Qualität herumschlagen, warum AS denn jetzt nicht mehr läuft, nachdem man den Brummi-CAD 18.53-Eintrag in der Registry von Groß- in Kleinbuchstaben geändert hat...).

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.

OS/2

Die Hardware-Anforderungen der OS/2-Version ergeben sich weitestgehend durch die des darunterliegenden Betriebssytemes, d.h. mindestens ein 80386SX-Prozessor, 8 Mbyte RAM (bzw. 4 ohne grafische Benutzeroberfläche) sowie ca 100..150 Mbyte Platz auf der Festplatte. Da AS2 nur eine 16-Bit-Applikation ist, sollte er theoretisch auch auf älteren OS/2-Versionen (und damit 80286-Prozessoren) lauffähig sein; ausprobieren konnte ich dies aber nicht.

UNIX

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 I 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 den Tabellen 2.1 bis 2.3. Falls eine der in diesen (oder den folgenden) Tabellen aufgeführten Dateien fehlt, hat jemand (im Zweifelsfalle ich) beim Kopieren geschlafen...

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. Für die DOS-DPMI-Version ergeben sich die in Tabelle 2.4 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. *DPMI*

Eine OS/2-Binärdistribution enthält neben den Basisdateien eine Reihe von DLLs, die zur Laufzeitumgebung des verwendeten emx-Compilers gehören (Tabelle 2.5). Falls man diese DLLs (oder neuere Versionen davon) bereits besitzt, kann man diese auch wieder löschen und seine eigenen benutzen. *OS/2*

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:

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	
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	
BITFUNCS.INC	Funktionen zur Bitmanipulation
CTYPE.INC	Funktionen zur Klassifizierung von Zeichen
80C50X.INC	Registeradressen SAB C50x
80C552.INC	Registeradressen 80C552
H8_3048.INC	Registeradressen H8/3048
REG166.INC	Adressen & Befehlsmakros 80C166/167
REG251.INC	Adressen & Bits 80C251
REG29K.INC	Peripherieadressen AMD 2924x
REG53X.INC	Registeradressen H8/53x
REG683XX.INC	Registeradressen 68332/68340/68360
REG7000.INC	Registeradressen TMS70Cxx
REG78K0.INC	Registeradressen 78K0
REG96.INC	Registeradressen MCS-96
REGACE.INC	Registeradressen ACE

Tabelle 2.1: Standardumfang einer Binärdistribution - Teil 1

Datei	Funktion
Verzeichnis INCLUDE	
REGAVR.INC	Register- & Bitadressen AVR-Familie
REGCOP8.INC	Registeradressen COP8
REGGP32.INC	Registeradressen 68HC908GP32
REGHC12.INC	Registeradressen 68HC12
REGM16C.INC	Registeradressen Mitsubishi M16C
REGMSP.INC	Registeradressen TI MSP430
REGST9.INC	Register- & Makrodefinitionen ST9
REGZ380.INC	Registeradressen Z380
STDDEF04.INC	Registeradressen 6804
STDDEF16.INC	Befehlsmakros und Registeradressen PIC16C5x
STDDEF17.INC	Registeradressen PIC17C4x
STDDEF18.INC	Registeradressen PIC16C8x
STDDEF2X.INC	Registeradressen TMS3202x
STDDEF37.INC	Register- & Bitadressen TMS370xxx
STDDEF3X.INC	Peripherieadressen TMS320C3x
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
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
Verzeichnis LIB	

Tabelle 2.2: Standardumfang einer Binärdistribution - Teil 2

Datei	Funktion
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 2.3: Standardumfang einer Binärdistribution - Teil 3

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

Tabelle 2.4: Zusätzliche Dateien in einer DPMI-Binärdistribution

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

Tabelle 2.5: Zusätzliche Dateien in einer OS/2-Binärdistribution

```
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.

DPMI

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!

OS/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 bwkannt 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.

UNIX

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 I.

2.4 Aufruf, Parameter

AS ist ein kommandozeilengesteuertes 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.

OS 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

```
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 PC's laufen sollte) und eine rudimentäre Bildschirmsteuerung benötigt, gibt er während der Assemblierung

OS/

PMI

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.

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 ASX in der Lage, bei Bedarf den vorhandenen Speicher durch eine Swap-Datei zu ,,erweitern". Dazu belegt man eine Environment-Variable `ASXSWAP` folgendermaßen:

```
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`: Assemblerlisting auf Konsole ausgeben. Falls mehrere Passes ausgeführt werden müssen, landen im Gegensatz zur nächsten Option die Listings aller Durchgänge auf der Ausgabe!
- `L`: Assemblerlisting 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.
- `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-Programmes 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 5.3 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.

- `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 Assemblerlistings ein-und ausblenden. Welcher Teil dabei welchem Bit zugeordnet ist, ist im übernächsten Abschnitt, der genauer auf das Format des Assemblerlistings 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. Jedesmal, 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 selbstdefinierte 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 verwenden. 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 Codefile).

- `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!!

- `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 und es sich nicht um 68008-Code handelt.

- `alias <neu>=<alt>`:
  definiert den Prozessortyp `<neu>` als einen Alias für den Typen `<alt>`. Zu den Sinn und Zweck von Aliassen 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!

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:

```
as 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 Includeverzeichnis 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 sog. *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 kein Codefile 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 OS/2 erweitert wie Unix das Datensegment einer Anwendung erst dann, wenn sie wirklich mehr Speicher anfordert. Eine Angabe wie

```
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...

NIX 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.

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 der Befehl 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. Dieser Knopf ist aus Eindeutigkeitsgründen 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), 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 hineinspringen 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 2.6).

Da sich diese Anleitung nicht gleichzeitig als Handbuch für die von AS unterstützten Prozessorfamilien versteht, ist dies leider auch nicht der richtige Platz,

Attribut	arithmetisch-logischer Befehl	Sprungbefehl
B	Byte (8 Bit)	———
W	Wort (16 Bit)	———
L	Langwort (32 Bit)	16-Bit-Displacement
Q	Vierfachwort (64 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 2.6: Erlaubte Attribute (Beispiel 680x0)

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:

```
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. [110], [24], [44] bzw. [45] 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.

Anstelle eines Kommentars am Ende kann die Zeile auch nur aus einem Kommentar bestehen, wenn er in der ersten Spalte beginnt.

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. 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.

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`.

Der erzeugte Code selber steht dahinter im Feld `Code` in hexadezimaler Schreibweise. Je nach Prozessortyp und aktuellem Segment können die Werte entweder als Bytes oder 16/32-Bit-Worte formatiert sein. 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 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 2.7 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 dort) 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 2.8 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.

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:

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

Tabelle 2.7: Zuordnung der Bits zu den Listingkomponenten

Name	Bedeutung
`TRUE`	logisch ,,wahr"
`FALSE`	logisch ,,falsch"
`CONSTPI`	Kreiszahl Pi (3.1415.....)
`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 Prozesor-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 2.8: Vordefinierte Symbole

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

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 E.

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 Symol 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 2.9 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.

2.8 Temporäre Symbole

Besonders bei Programmen mit vielen aufeinanderfolgenden 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

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

Tabelle 2.9: Rückgabewerte der `SYMTYPE`-Funktion

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 da 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 (.) anfngt, wird das Symbol nicht mit diesem Namen in der Symboltabelle abgelegt. Stattdessen wird der Name des zuletzt definierten Symbols ohne vorangestellten Punkt davorgehängt. Auf diese Weise nehmen Sybole, 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-tempor"ares Symbol 'proc1'

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

proc2:                      ; nicht-tempor"ares 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 Integerkonstanten

Integerkonstanten bezeichnen ganze Zahlen. Sie dürfen entweder als eine Folge von Ziffern oder als eine Reihe von in *einfachen* Hochkommas eingeschlossenen Zeichen geschrieben werden. Werden sie als Ziffernfolgen geschrieben, so kann dies in verschiedenen Zahlensystemen erfolgen, deren Kennzeichnung von verwendeten Zielprozessor abhängt (Tabelle 2.10).

Falls das Zahlensystem nicht explizit durch vor-oder nachgestelle Zeichen vorgegeben wird, nimmt AS die Basis an, die mit dem `RADIX`-Befehl vorgegeben wurde (der Default dieser Einstellung ist wiederum 10). Mit diesem Befehl lassen sich auch ,,ungewöhnlicheSZahlensysteme, 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 Verwendung von

	Intel-Modus (Intel, Zilog, Thomson, Texas, Toshiba, NEC, Siemens, Philips, Fujitsu, Fairchild, Intersil)	Motorola-Modus (Rockwell, Motorola, Microchip, Thomson, Hitachi)	C-Modus (PowerPC, AMD29K, National, Symbios, Atmel)
dezimal	direkt	direkt	direkt
hexadezimal	nachgestelltes H	vorangestelltes $	vorangestelltes 0x
binär	nachgestelltes B	vorangestelltes %	vorangestelltes 0b
oktal	nachgestelltes O nachgestelltes Q	vorangestelltes @	vorangestellte 0

Tabelle 2.10: mögliche Zahlensysteme

Buchstaben in Integerkonstanten 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 Integerkonstante, 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 hintangestellte 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 Integerkonstante kennzeichnen, kennen dieses Problem nicht. (*hihihi!*).

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!**

Mit Hilfe des `RELAXED`-Befehls (siehe Abschnitt 3.9.6) kann die starre Zuordnung einer Schreibweise zu einem Zielprozessor aufgehoben werden, so daß man eine beliebige Schreibweise verwenden kann (auf Kosten der Kompatibilität zu Standard-Assemblern). Defaultmäßig ist diese Option aber ausgeschaltet. Ebenfalls mit dieser Option eröffnet sich eine weitere, vierte Schreibweise für Integerkonstanten, wie man sie bei manchen Fremdassemblern antrifft: 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'00010010000110100010101100111000'
```

Diese Schreibweise ist für keine Prozessorarchitektur der Default und nur im `RELAXED`-Modus erreichbar. Sie dient in erster Linie der einfacheren Portierung von Fremdquellen und wird nicht für neu erstellte Programme empfohlen.

Wie bereits angesprochen, können Integer-Konstanten auch als ASCII-Werte geschrieben werden, so entsprechen

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

Wichtig ist, daß hier die Zeichen in *einfachen Hochkommas* geschrieben werden, um sie von den weiter unten beschriebenen Stringkonstanten zu unterscheiden.

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 Integerkonstante 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 *doppelte Hochkommas* (um sie von den oben beschrieben ASCII-Integers zu unterscheiden) eingeschlossen werden. Um nun aber auch Gänsefüßchen und Sonderzeichen ohne Verrenkungen in String-Konstanten einbauen zu können, wurde ein ,,Escape-Mechanismus" eingebaut, der Programmierer(inne)n aus C 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
NIX 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!

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 durcheinanderkommt. 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 Integerkonstanten zulässig, z.B. so:

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

Jedoch hat alles seine Grenzen, weil der darüberliegende 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 Evaluierung

Die Berechnung von im Formelausdruck entstehenden Zwischenergebnissen erfolgt immer mit der höchsten verfügbaren Wortbreite, d.h. 32 Bit für Ganzzahlen, 80 Bit für Gleitkommazahlen und 255 Zeichen für Strings. Eine eventuelle Prüfung auf Wertebereichsüberschreitung findet erst am Endergebnis statt.

Die portable C-Version kann nur mit 64-Bit-Gleitkommazahlen umgehen, ist daher auf einen Maximalwert von ca. 10^{308} beschränkt. Als Ausgleich werden auf einigen Plattformen Integers mit 64 Bit Breite behandelt. *UNIX*

2.9.5 Operatoren

Der Assembler stellt zur Verknüpfung die in Tabelle 2.11 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 Bit breit und vorzeichenbehaftet. Für die logischen Operatoren ist ein Ausdruck TRUE, falls er ungleich 0 ist, ansonsten FALSE.

Die Bitspiegelung ist wohl etwas erklärungsbedürftig: Der Operator spiegelt die untersten Bits im ersten Operanden, läßt die darüberliegenden 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!

2.9.6 Funktionen

Zusätzlich zu den Operatoren definiert der Assembler noch eine Reihe in erster Linie transzendenter Funktionen mit Gleitkommaargument, die Tabellen 2.12 und 2.13 auflisten. Die Funktionen `FIRSTBIT`, `LASTBIT` und `BITPOS` liefern als Ergebnis

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

Tabelle 2.11: in AS definierte Operatoren

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 2.12: vordefinierte Funktionen in AS - Teil 1 (Integer- und Gleitkommafunktionen)

-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 behandlet, 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.

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 selbstdefinierten 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 ≥ 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 Variablendefinitionen 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, leerzulassen, 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 vermaledeite, 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 undefinerten 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 kürzestmöglichen 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, 4004/4040, 80C16x, AVR, XS1, Z8

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 aus Geschwindigkeitsgründen gesondert, so daß ein besonderer Mechanismus vonnöten ist, um symbolische Register zu definieren. Ein Registersymbol wird üblicherweise durch die `REG`-Anweisung definiert und hat ansonsten die gleiche Form wie eine `EQU`-Definition. Sie unterliegt jedoch einer Reihe von Einschränkungen: Zum einen ist ein Registersymbol eine reine 'as is' gespeicherte Zeichenkette, die auch nur in dieser Form verwendet werden kann. Es ist also z.B. keine Arithmetik möglich, um aus einem Register den Nachfolger zu berechnen, etwa so:

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

Zum anderen muß ein Registersymbol vor seiner ersten Nutzung definiert werden; eine Vorwärtsreferenz würde dazu führen, daß AS bei nicht gefundenem Registersymbol eine Vorwärtsreferenz auf eine Speicherstelle vermutet, und bei den meisten Prozessoren sind die Nutzungsmöglichkeiten für Speicherstellen als Operanden deutlich eingeschränkter als für Register, so daß es mit ziemlicher Sicherheit Fehler hagelt...

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. Aufgrund der fehlenden Möglichkeit zur Vorwärtsreferenz gibt es aber keine Entsprechung zur `FORWARD`-Direktive, und da Registersymbole im allgemeinen nur in einem sehr eng umschränkten Kontext eine Bedeutung haben, ist ein Export per `PUBLIC` oder `GLOBAL` auch nicht vorgesehen.

Sind in einem Kontext ein normales als auch ein Registersymbol gleichen Namens bekannt, so wird immer das Registersymbol vorgezogen. Dies ist aber nicht der Fall, wenn der Name nicht alleine, sondern eingebunden in einen Ausdruck steht (dazu reichen Klammern!), dann wird das normale Symbol benutzt.

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 dringelassen. 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-Aliasse

Mit Varianten gängiger Mikrocontroller-Familien ist es wie mit Kaninchen: Sie vermehren sich schneller, als man mit der Versorgung hinterherkommen 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 Prozessorkernes wichtig, um z.B. in den Includefiles 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 Includefiles 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.

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 in Kleinbuchstaben	String	String
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

Tabelle 2.13: vordefinierte Funktionen in AS - Teil 2 (Integer- und String-Funktionen)

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 Segmentverquickung. 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 Interruptvektoren, 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.

Anstelle von `EQU` darf auch einfach ein Gleichheitszeichen geschrieben werden, analog kann man anstelle von `SET` bzw. `EVAL` einfach `:=` schreiben.

Aus Kompatibilitätsgründen zum Originalassembler gibt es für das KCPSM-Target auch den `CONSTANT`-Befehl, der im Gegensatz zu `EQU` Namen und Wert als Argument 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 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!

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 RAM-Zellen und (wie der Name ahnen läßt) im Datenbereich eingeblendeten Hardwareregistern. Der dabei zugelassene Wertebereich ist identisch mit dem bei `ORG` für das `DATA`-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, ST9

`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.

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 PORT

Gültigkeit: 8080/8085/8086, XA, Z80, 320C2x/5x, TLCS-47, AVR

`PORT` arbeitet analog zu `SFR`, nur wird das Symbol dem I/O-Adreßbereich zugeordnet. Erlaubte Werte sind 0..7 beim 3201x, 0..15 beim 320C2x, 0..65535 beim 8086 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
```

3.1.8 REG und NAMEREG

*Gültigkeit: AVR, M*Core, ST9, 80C16x, KCPSM (*`NAMEREG` *nur für KCPSM(3)), LatticeMico8, MSP430*

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
```

3.1.9 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 4.17.

3.1.10 CHARSET

Gültigkeit: alle Prozessoren

Einplatinensysteme, 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. Um nun aber keine fehlerträchtigen Handumkodierungen vornehmen zu müssen, enthält der Assembler eine Umsetzungstabelle für Zeichen, die jedem Quellcode ein Zielzeichen zuordnet. Zur Modifikation dieser Tabelle (die initial 1:1 übersetzt), dient der Befehl `CHARSET`. `CHARSET` kann mit verschiedenen Parameterzahlen und -typen angewendet werden. Ist die Parameterzahl eins, so muß es sich um einen String-Ausdruck handeln, der von AS als Dateiname interpretiert wird. 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 in einem Schlag aktivieren. In allen anderen Varianten muß der erste Parameter ein Integer im Bereich von 0 bis 255 sein, der den Startpunkt der in der Übersetzungstabelle zu modifizierenden Einträge angibt. Es folgen dann ein oder zwei weitere Parameter, die die Art der Übersetzung angeben:

Ein einzelner, weiterer Integer verändert genau einen Eintrag. So bedeutet z.B.

```
CHARSET 'ä',128
```

daß das Zielsystem das ä mit der Zahl 128 kodiert. Sind jedoch zwei weitere Integers angegeben, so ist der erste von ihnen der letzte zu modifizierende Eintrag, der zweite der neue Wert des ersten Eintrags; alle weiteren Einträge bis zum Bereichsende werden sequentiell neu belegt. Falls z.B. das Zielsystem keine Kleinbuchstaben unterstützt, können mit

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

alle Kleinbuchstaben auf die passenden Großbuchstaben automatisch umgemappt werden.

In der letzten Variante folgt nach dem Startindex ein String, der die ab dem Startindex abzulegenden Zeichen angibt. Das letzte Beispiel könnte man also auch so formulieren:

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

`CHARSET` kann auch ganz ohne Parameter aufgerufen werden, allerdings mit ziemlich gründlichen Folgen: Dies bewirkt eine Reinitialisierung der Übersetzungstabelle in ihren Urzustand, d.h. man bekommt wieder eine 1:1-Übersetzung.

ACHTUNG! `CHARSET` beeinflußt nicht nur im Speicher abgelegte Stringkonstanten, sondern auch als ,,ASCII" formulierte Integerkonstanten. Dies bedeutet, daß eine evtl. bereits modifizierte Umsetzungstabelle in den obigen Beispielen zu anderen Ergebnissen führen kann!

3.1.11 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.

3.1.12 ENUM

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.

`ENUM`-Befehle sind von Hause aus einzeilig, d.h. bei einem neuen `ENUM`-Befehl beginnt die Numerierung wieder bei Null. Mehrzeilige Aufzählungen kann man aber mit einem kleinen Trick erreichen, der die Tatsache ausnutzt, daß man mit einer expliziten Zuweisung den internen Zähler neu setzen kann, wie in dem folgenden Fall:

```
ENUM Januar=1,Februar,März,April,Mai,Juni
```

Hier werden den Monatsnamen die Zahlenwerte 1..6 zugewiesen. Möchte man die Aufzählung nun fortsetzen, geht das folgendermaßen:

```
ENUM Juli=Juni+1,August,September,Oktober
ENUM November=Oktober+1,Dezember
```

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

3.1.13 PUSHV und POPV

Gültigkeit: alle Prozessoren

Mit `PUSHV` und `POPV` ist es möglich, den Wert von (nicht makrolokalen) 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 herunterholen 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 assemblerinternen Adreßzähler mit einem neuen Wert zu besetzen. Der Wertebereich ist vom momentan gewählten Segment und vom Prozessortyp abhängig (Tabellen 3.1 bis 3.4). 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	IDATA	XDATA	YDATA	BITDATA	IO	REG	ROMDATA
68xxx	4G	—	—	—	—	—	—	—	—
DSP56000/ DSP56300	64K/ 16M	—	—	64K/ 16M	64K/ 16M	—	—	—	—
PowerPC	4G	—	—	—	—	—	—	—	—
M*Core	4G	—	—	—	—	—	—	—	—
6800,6301, 6811	64K	—	—	—	—	—	—	—	—
6805/ HC08	8K/ 64K	— —	— —	— —	— —	— —	— —	— —	— —
6809, 6309	64K	—	—	—	—	—	—	—	—
68HC12(X), XGATE	64K	—	—	—	—	—	—	—	—
68HC16	1M	—	—	—	—	—	—	—	—
68RS08	16K	—	—	—	—	—	—	—	—
H8/300 H8/300H	64K 16M	—	—	—	—	—	—	—	—
H8/500 (Min) H8/500 (Max)	64K 16M	— —	— —	— —	— —	— —	— —	— —	— —
SH7000/ 7600/7700	4G	—	—	—	—	—	—	—	—
6502, MELPS740	64K	—	—	—	—	—	—	—	—
65816, MELPS-7700	16M	—	—	—	—	—	—	—	—
MELPS-4500	8K	416	—	—	—	—	—	—	—
M16	4G	—	—	—	—	—	—	—	—
M16C	1M	—	—	—	—	—	—	—	—

Tabelle 3.1: Adreßbereiche für `ORG` — Teil 1

Ziel	CODE	DATA	IDATA	XDATA	YDATA	BITDATA	IO	REG	ROMDATA
4004	4K	256	—	—	—	—	—	—	—
8008	16K	8	—	—	—	—	—	—	—
MCS-48, MCS-41	4K	—	256	256	—	—	—	—	—
MCS-51	64K	256	256* In. 80H	64K	—	256	—	—	—
80C390	16M	256	256* In. 80H	16M	—	256	—	—	—
MCS-251	16M	—	—	—	—	—	512	—	—
MCS-96 196(N)/ 296	64K 16M	—	—	—	—	—	—	—	—
8080, 8085	64K	—	—	—	—	—	256	—	—
80x86,	64K	64K	—	64K	—	—	64K	—	—
68xx0	4G	—	—	—	—	—	—	—	—
8X30x	8K	—	—	—	—	—	—	—	—
2650	8K	—	—	—	—	—	—	—	—
XA	16M	16M	—	—	—	—	2K In. 1K	—	—
AVR	8K	64K	—	—	—	—	64	—	—
29XXX	4G	—	—	—	—	—	—	—	—
80C166, 80C167	256K 16M	—	—	—	—	—	—	—	—
Z80, Z180, Z380	64K 512K+ 4G	—	—	—	—	—	256 256 4G	—	—
Z8	64K	256	—	—	—	—	—	—	—
eZ8	64K	256	—	64K	—	—	—	—	—
KCPSM	256	256	—	—	—	—	—	—	—
* Da der 8051 kein RAM jenseits 80h hat, muß der Initialwert für den 8051 als Zielprozessor auf jeden Fall mit `ORG` angepaßt werden!!									
+ Da der Z180 weiterhin logisch nur 64K ansprechen kann, ist der ganze Adreßraum nur mittels `PHASE`-Anweisungen erreichbar!									

Tabelle 3.2: Adreßbereiche für `ORG` — Teil 2

Ziel	CODE	DATA	IDATA	XDATA	YDATA	BITDATA	IO	REG	ROMDATA
KCPSM3	256	64	—	—	—	—	256	—	—
Mico8	4096	256	—	—	—	—	256	—	—
TLCS-900(L)	16M	—	—	—	—	—	—	—	—
TLCS-90	64K	—	—	—	—	—	—	—	—
TLCS-870	64K	—	—	—	—	—	—	—	—
TLCS-47	64K	1K	—	—	—	—	16	—	—
TLCS-9000	16M	—	—	—	—	—	—	—	—
PIC 16C5x	2K	32	—	—	—	—	—	—	—
PIC 16C5x	2K	32	—	—	—	—	—	—	—
PIC 16C64, 16C86	8K	512	—	—	—	—	—	—	—
PIC 17C42	64K	256	—	—	—	—	—	—	—
ST6	4K	256	—	—	—	—	—	—	—
ST7	64K	—	—	—	—	—	—	—	—
ST9	64K	64K	—	—	—	—	—	256	—
6804	4K	256	—	—	—	—	—	—	—
32010 32015	4K 4K	144 256	—	—	—	—	8 8	—	—
320C2x	64K	64K	—	—	—	—	16	—	—
320C3x	16M	—	—	—	—	—	—	—	—
320C5x/ 320C20x/ 320C54x	64K	64K	—	—	—	—	64K	—	—
TMS 9900	64K	—	—	—	—	—	—	—	—

Tabelle 3.3: Adreßbereiche für `ORG` — Teil 3

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 →
MCF5200 → 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-Famile haben den gleichen Prozessorkern (eine leicht abgemagerte 68020-CPU), jedoch völlig unterschiedliche Peripherie. MCF5200 repräsentiert die ColdFire-Familie von Motorola, 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

Ziel	CODE	DATA	IDATA	XDATA	YDATA	BITDATA	IO	REG	ROMDATA
TMS 70Cxx	64K	—	—	—	—	—	—	—	—
370xxx	64K	—	—	—	—	—	—	—	—
MSP430	64K	—	—	—	—	—	—	—	—
SC/MP	64K	—	—	—	—	—	—	—	—
807x	64K	—	—	—	—	—	—	—	—
COP4	512	—	—	—	—	—	—	—	—
COP8	8K	256	—	—	—	—	—	—	—
ACE	4K In. 800H/ 0C00H	—	—	—	—	—	—	—	—
μPD 78(C)10	64K	—	—	—	—	—	—	—	—
7566	1K	64	—	—	—	—	—	—	—
7508	4K	256	—	—	—	—	16	—	—
75K0	16K	4K	—	—	—	—	—	—	—
78K0	64K	—	—	—	—	—	—	—	—
78K0	1M	—	—	—	—	—	—	—	—
7720	512	128	—	—	—	—	—	—	512
7725	2K	256	—	—	—	—	—	—	1024
77230	8K	—	—	512	512	—	—	—	1K
53C8XX	4G	—	—	—	—	—	—	—	—
F^{2}MC8L	64K	—	—	—	—	—	—	—	—
F^{2}MC16L	16M	—	—	—	—	—	—	—	—
XS1	4G	—	—	—	—	—	—	—	—

Tabelle 3.4: Adreßbereiche für `ORG` — Teil 4

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 mikrocontrollerspezifische 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 - [58] 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) MCORE

e) XGATE

f) 6800 → 6301 → 6811

Während der 6301 nur neue Befehle definiert, liefert der 6811 neben weiteren Befehlen ein zweites Indexregister Y zur Adressierung.

g) 6809/6309 und 6805/68HC08/68HCS08

Diese Prozessoren sind zwar teilweise quellcodekompatibel zu den anderen 68xx-ern, haben aber ein anderes Binärcodeformat 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 [37]).

h) 68HC12 ⟶ 68HC12X

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

i) 68HC16

j) HD6413308 ⟶ 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.

k) HD6475328 ⟶ HD6475348 ⟶ HD6475368 ⟶ HD6475388

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

l) SH7000 ⟶ SH7600 ⟶ SH7700

Der Prozessorkern des 7600ers bietet eine Handvoll Befehle mehr, die Lücken im Befehlssatz des 7000ers schließen (verzögerte, bedingte sowie relative und indirekte Sprünge, Multiplikationen mit 32-Bit-Operanden sowie Multiplizier/Addier-Befehle). Die 7700er-Reihe (auch als SH3 geläufig) bietet weiterhin eine zweite Registerbank, bessere Schiebebefehle sowie Befehle zur Cache-Steuerung.

m) 6502 → 65(S)C02 / MELPS740

Die CMOS-Version definiert einige zusätzliche Befehle, außerdem sind bei einigen Befehlen Adressierungsarten hinzugekommen, die beim 6502 nicht möglich waren. 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 Akkumulatos sind die Erweiterungen disjunkt. Dem 65SC02 fehlen die Bitmanipulationsbefehle des 65C02. 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.

n) 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.

o) MELPS4500

p) M16

q) M16

r) 4004 → 4040

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

s) 4008 → 8008NEW

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

t) 8021, 8022, 8039, 80C39, 8048, 80C48, 8041, 8042

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. Die CMOS-Versionen lassen sich mit dem `IDL`-Befehl in einen Ruhezustand niedriger Stromaufnahme überführen. Der 8041 und 8042 haben einige Zusatzbefehle zur Steuerung der Busschnittstelle, dafür fehlen aber einige andere Befehle. Darüber hinaus ist bei diesen Prozessoren der Programmadreßraum nicht extern erweiterbar, weshalb AS das Codesegment bei diesen Prozessoren auf 1 bzw. 2 Kbyte begrenzt.

u) 87C750 → 8051, 8052, 80C320, 80C501, 80C502,
80C504, 80515, and 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.

v) 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.

w) 8080 → 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 4.15 dokumentiert.

x) 8086 → 80186 → V30 → V35

Hier kommen wieder nur neue Befehle dazu. Die entsprechenden 8-Bitter sind wegen ihrer Befehlskompatibilität nicht aufgeführt, für ein 8088-System ist also z.B. 8086 anzugeben.

y) 80960

z) 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.

aa) XAG1, XAG2, XAG3

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

ab) AT90S1200 → AT90S2313 → AT90S4414 → AT90S8515 → ATMEGA8 → ATMEGA16

Der erste Vertreter der AVR-Reihe stellt die Minimalkonfiguration dar, ohne RAM-Speicher und demzufolge auch ohne Load/Store-Befehle. Die nächsten drei Prozessoren unterscheiden sich nur im Speicherausbau und in der eingebauten Peripherie, was in `REGAVR.INC` differenziert wird. Ähnlich verhält es sich mit den Mega-AVRs, die gegenüber den Vorgängern aber auch neue Maschinenbefehle mitbringen.

ac) 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.

ad) 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.

ae) Z80 → Z80UNDOC → Z180 → 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.

af) Z8601, Z8604, Z8608, Z8630, Z8631 → eZ8

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 ist jedoch der eZ8, mit einem stark erweiterten Befehlssatz, der auch nur auf Quellebene weitestgehens aufwärts-kompatibel ist.

ag) 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!

ah) 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.

ai) 96C141, 93C141

Diese beiden Prozessoren repräsentieren die beiden Varianten der Prozessorfamilie: TLCS-900 und TLCS-900L. Die Unterschiede dieser beiden Varianten werden in Abschnitt 4.23 genauer beleuchtet.

aj) 90C141

ak) 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.

al) 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.

am) 97C241

an) 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.

ao) 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.

ap) 17C42

aq) ST6210/ST6215 → ST6220/ST6225

Die einzige Unterscheidung, die AS zwischen den beiden Paaren vornimmt, ist der bei den ersten beiden kleinere Adreßraum (2K anstelle 4K). Die Feinunterscheidung dient zur automatischen Unterscheidung in der Quelldatei, welche Hardware jeweils vorhanden ist (analog zum 8051/52/515).

ar) ST7

as) ST9020, ST9030, ST9040, ST9050

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

at) 6804

au) 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.

av) 320C25 → 320C26 → 320C28

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

aw) 320C30, 320C31

Der 320C31 ist eine etwas ,,abgespeckte" Version mit dem gleichen Befehlssatz, jedoch weniger Peripherie. In STDDEF3X.INC wird diese Unterscheidung ausgenutzt.

ax) 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.

ay) 320C541

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

az) TMS9900

ba) 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.

bb) 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.

bc) MSP430

bd) SC/MP

be) 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.

bf) 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...

bg) COP410 → COP420 → COP440 → COP444 Die COP42x-Derivate bieten einige weitere Befehle, des weiteren wurden Befehlen in ihrem Wertebereich erweitert.

bh) 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.

bi) 7810 → 78C10

Die NMOS-Version besitzt keinen STOP-Modus; der entspechende Befehl sowie das ZCM-Register fehlen demzufolge. **VORSICHT!** NMOS- und CMOS-Version differieren zum Teil in den Reset-Werten einiger Register!

bj) 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!

bk) 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.

bl) 78070

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

bm) 78214

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

bn) 7720 → 7725

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

bo) 77230

bp) 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.

bq) MB89190

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

br) MB9500

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

bs) XS1

Dieser Typ repräsentiert die XCore-"Familie".

Beim CPU-Befehl muß der Prozessortyp als einfache Konstante 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.

Default für den Prozessortyp ist `68008`, sofern dieser über die gleichnamige Kommandozeilenoption nicht verändert wurde.

3.2.4 SUPMODE, FPU, PMMU

*Gültigkeit: 680x0, FPU auch 80x86, i960, SUPMODE auch TLCS-900, SH7000, i960, 29K, XA, PowerPC, M*CORE und TMS9900*

Mit diesen drei 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 bzw. 68881/68882. Die Statusvariable heißt `FPUAVAIL`.
- `PMMU`: erlaubt bzw. sperrt die Befehle der Speicherverwaltungseinheit 68851 bzw. der im 68030 eingebauten MMU. **ACHTUNG!** Die 68030-MMU erlaubt nur eine rel. kleine Untermenge der 68851-Befehle. Der Assembler kann hier keine Prüfung vornehmen! Die Statusvariable heißt `PMMUAVAIL`.

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

3.2.5 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!

3.2.6 PADDING

*Gültigkeit: 680x0, 68xx, M*Core, XA, H8, SH7000, TMS9900, MSP430, ST7*

Prozessoren der 680x0-Familie stehen ungeraden Adressen ziemlich kritisch gegenüber: Befehle dürfen nicht auf einer ungeraden Adresse beginnen, und Datenzugriffe sind mit ungeraden Adressen bis zum 68010 nur byteorientiert erlaubt. Die H8-Familie setzt bei Zugriffen auf ungerade Adressen das unterste Adreßbit einfach ganz auf Null, die 500er ,,bedanken" sich wiederum mit einer Exception... AS bemüht sich daher, mit `DC` oder `DS` angelegte Datenstrukturen immer mit einer geraden Bytezahl abzulegen. Das bedeutet bei den Befehlen `DS.B` und `DC.B` aber unter Umständen, daß ein Füllbyte eingefügt werden muß. Dieses Verhalten kann man mit dem `PADDING`-Befehl ein- und ausschalten. Als Argument 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!

3.2.7 PACKING

Gültigkeit: AVR

`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 hinterienander hat, fängt sich bei ungeraden Argumentzahlen trotzdem halbvolle Wörter ein!

3.2.8 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.

3.2.9 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.

3.2.10 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`.

3.2.11 BIGENDIAN

Gültigkeit: MCS-51/251, PowerPC

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` und `DT` 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.

3.2.12 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.

3.2.13 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.

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      ?
```

3.2.14 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 aufgehoben.

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

3.2.15 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);
- Flag, ob Expansionen folgender Makos im Listing ausgegeben werden sollen (mit `MACEXP` 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 Includefiles 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 Includefiles 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.

3.2.16 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.

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 konnten die Entwickler letzten Endes nicht dem 'Sündenfall' widerstehen und haben in den K4 ein Banking-Schema eingebaut, um mit 16 Adreßleitungen mehr als 64 Kbyte anzusprechen. Die Register `MMSIZ`, `MMWBR`, `MM1CR` und `MM2CR` legen fest, ob wie die beiden zusätzlichen 512K-Bereiche in den physikalischen Adreßraum eingeblendet werden sollen. Initial nimmt AS den Reset-Zustand dieser Register an, d.h. alle mit $00 belegt und das Windowing ist abgeschaltet.

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	Datenbank	0-$ff	0
PG	Code-Bank	0-$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 4.9. 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`!

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
```

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.

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.

ST6

Die Mikrokontroller der ST62-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.

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.

μPD78(C)10

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.

320C3x

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 Bit aus dem DP-Register ergänzt werden. Bei Adressierungen kann man aber trotzdem die volle 24-Bit-Adresse angeben, AS prüft dann, ob die oberen 8 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 zurückgreifen: 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.

3.2.17 EMULATED

Gültigkeit: 29K

AMD hat die Ausnahmebehandlung für undefinierte Befehle bei der 29000-Serie so definiert, daß für jeden einzelnen Befehl ein Exceptionvektor 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
```

3.2.18 BRANCHEXT

Gültigkeit: XA

`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!

3.3 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, SH7000, DSP56xxx, XA, ST7*

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 Fragenzeichen als Operanden nicht mit 'normalen' Konstanten in einer Anweisung gemischt werden.

3.3.2 DS[.size]

*Gültigkeit: 680x0, M*Core, 68xx, H8, SH7x00, DSP56xxx, XA, ST7*

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 DB,DW,DD,DQ & DT

Gültigkeit: Intel (außer 4004/4040), Zilog, Toshiba, NEC, TMS370, Siemens, AMD, M16(C), MELPS7700/65816, National, ST9, TMS7000, µPD77230, Signetics, Fairchild, Intersil, XS1

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:

- DB: Byte oder ASCII-String wie bei DC.B
- DW: 16-Bit-Integer
- DD: 32-Bit-Integer oder single precision
- DQ: double precision (64 Bit)
- DT: extended precision (80 Bit)

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 DUPs 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!

Um mit dem M80 verträglich zu sein, darf im Z80-Modus anstelle von DB/DW auch DEFB/DEFW geschrieben werden.

Analog stellen BYTE/ADDR bzw. WORD/ADDRW beim COP4/8 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.

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 Integerargumenten auch Gleitkommawerte (und legt sie prozessorspezifischen 32-Bit-Format ab). DUP gibt es *nicht*!

3.3.4 DS, DS8

Gültigkeit: Intel, Zilog, Toshiba, NEC, TMS370, Siemens, AMD, M16(C), National, ST9, TMS7000, Intersil

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 (gell, Michael?) so fest verdrahtet zu sein, daß sie ihn als eingebauten Befehl erwarten...hoffentlich sind selbige jetzt zufrieden `;-)`

`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.

3.3.5 BYT oder FCB

Gültigkeit: 6502, 68xx

Mit diesem Befehl werden im 65xx/68xx-Modus Byte-Konstanten oder ASCII-Strings abgelegt, er entspricht also `DC.B` beim 68000 oder `DB` bei Intel. Ein Wiederholungsfaktor darf analog zu `DC` jedem einzelnen Parameter in eckigen Klammern vorangestellt werden.

3.3.6 BYTE

Gültigkeit: ST6, 320C2(0)x, 320C5x, MSP, TMS9900

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.

3.3.7 DC8

Gültigkeit: SC144xx

Dieser Befehl ist ein Alias für `DB`, d.h. mit ihm können Byte-Konstanten oder Strings im Speicher abgelegt werden.

3.3.8 ADR oder FDB

Gültigkeit: 6502, 68xx

Mit diesem Befehl werden im 65xx/68xx-Modus Wortkonstanten abgelegt, er entspricht also `DC.W` beim 68000 oder `DW` bei Intel. Ein Wiederholungsfaktor darf analog zu `DC` jedem einzelnen Parameter in eckigen Klammern vorangestellt werden.

3.3.9 WORD

Gültigkeit: ST6, i960, 320C2(0)x, 320C3x, 320C5x, MSP

Für den 320C3x 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.

3.3.10 DW16

Gültigkeit: SC144xx

Diser Befehl ist beim SC144xx der Weg, Konstanten mit Wortlänge (16 Bit) im Speicher abzulegen und damit ein ALIAS für DW.

3.3.11 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).

3.3.12 SINGLE, DOUBLE und EXTENDED

Gültigkeit: 320C3x (nicht `DOUBLE`*), 320C6x (nicht* `EXTENDED`*)*

Mit diesen Befehlen werden Gleitkomma-Konstanten im Speicher abgelegt, jedoch beim 320C3x 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.

3.3.13 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).

3.3.14 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
- `DFLOAT`: 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).

3.3.15 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...

3.3.16 DATA

Gültigkeit: PIC, 320xx, AVR, MELPS-4500, 4004/4040, µPD772x

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 und 4500er 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 sogar derer 4 (MSB zuerst). Im Gegensatz dazu muß im Datensegment des 4500ers ein Zeichen auf zwei Speicherstellen verteilt werden, ebenso wie beim 4004. Der Wertebereich für Integers entspricht der Wortbreite des jeweiligen Prozessors im jeweiligen Segment. Das bedeutet, daß `DATA` beim 320C3x die Funktion von `WORD` mit einschließt (die von `SINGLE` übrigens auch, wenn AS das Argument als Gleitkommazahl erkennt).

3.3.17 ZERO

Gültigkeit: PIC

Dieser Befehl legt einen durch den Parameter spezifizierte Zahl von Nullworten (=NOPs) im Speicher ab. Es können maximal 512 Nullen mit einem Befehl abgelegt werden.

3.3.18 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. Die Maximalgröße des Blocks beträgt 1024 Elemente für `FB` bzw. 512 Elemente für `FW`.

3.3.19 ASCII und ASCIZ

Gültigkeit: ST6

Mit diesen beiden 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.

3.3.20 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 mögliich 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.

3.3.21 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.

3.3.22 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.

3.3.23 BLOCK

Gültigkeit: ST6

Dito.

3.3.24 SPACE

Gültigkeit: i960

Dito.

3.3.25 RES

Gültigkeit: PIC, MELPS-4500, 3201x, 320C2(0)x, 320C5x, AVR, µPD772x

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 Nibbles sowie bei Texas ebenfalls Wörter.

3.3.26 BSS

Gültigkeit: 320C2(0)x, 320C3x, 320C5x, 320C6x, 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.

3.3.27 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`.

3.3.28 DS16

Gültigkeit: SC144xx

Dieser Befehl reserviert Speicher in Schritten von vollständigen Worten, d.h. 16 Bit. Er stellt einen Alias zu `DW` dar.

3.3.29 ALIGN

Gültigkeit: alle Prozessoren

`ALIGN` mit einem Integerausdruck 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. Wie auch bei `DS.x 0` ist der freibleibende Speicherraum undefiniert.

3.3.30 LTORG

Gültigkeit: SH7x00

Da der SH7000-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 SH7000-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. Die Parameternamen werden wie üblich durch Kommas getrennt und müssen — wie der Makroname selber — den Konventionen für Symbolnamen (2.7) genügen.

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.

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` 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 case-sensitiven Modus in Großbuchstaben umsetzt, in Strings aber nie eine Umwandlung in Großbuchstaben erfolgt. Die Makroparameternamen müssen in den Stringkonstanten daher groß geschrieben 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 Rekusion 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 intelverblödeter Programmierer möchte die Befehle `PUSH/POP` unbedingt auch auf dem 68000 haben. Er löst das ,,Problem" 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 Includefilegrenzen hinausgehen.

In Makrorümpfen definierte Labels werden immer als lokal betrachtet, auer 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 darauffolgenden 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...

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 Steueranweisungen `GLOBALSYMBOLS` bzw. `NOGLOBALSYMBOLS` (durch geschweifte Klammern als solche gekennzeichnet) enthalten, um zu steuern, ob benutzte Labels für jeden Durchgang automatisch lokal sind oder nicht.

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.

Ob Symbole für jede einzelne Repetition lokal sind oder nicht, kann wiederum durch die Steuerparameter `GLOBALSYMBOLS` bzw. `NOGLOBALSYMBOLS` (durch geschweifte Klammern als solche gekennzeichnet) bestimmt werden.

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 assenbliert, 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
```

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 zurückgesetzt. 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      "ARG"<>""
           push    arg
           shift
           pushlist ALLARGS
          endif
          endm
```

Also eine Rekursion, in der pro Schritt die Argumentliste (`ALLARGS`) um eins verküzt wird. Der wichtige Trick ist, daß jedes Mal eine neue Expansion gestartet wird...

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 momentante Einstellung kann aus ener 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.

Selbstdefinierte 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!

Da die Berechnung des Funktionsergebnisses anhand des Formelausdruckes auf textueller Ebene erfolgt, kann der Ergebnistyp von dem Typ des Eingangsargumentes 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. `ENDS` schreiben). Ein 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 zurückgesetzt. 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.

ACHTUNG! Obwohl AS keine Argumente bei der Definition einer Struktur erwartet, werden gegebene Argumente nicht als Fehler gemeldet, sondern schlicht ignoriert. Dies ist vorgesehen, um in Zukunft einer deklarierten Struktur direkt Werte zuweisen zu können.

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
```

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 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.

3.5.6 Strukturen und Makros

Will man Strukturen über Makros instantiieren, 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` <Symbol> : wahr, falls das Symbol definiert wurde. Die Definition muß vor `IFDEF` erfolgt sein.
- `IFNDEF` <Symbol> : Umkehrung zu `IFDEF`
- `IFUSED` <Symbol> : wahr, falls das Symbol bisher mindestens einmal benutzt wurde.
- `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.

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

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.

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 zurückgesetzt, 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

Mit dem Befehl

```
MACEXP  off
```

kann man erreichen, daß bei Makroexpansionen nur noch der Makroaufruf und nicht der expandierte Text ausgegeben wird. Die ist bei makrointensivem Code sinnvoll, um das Listing nicht ins Uferlose wachsen zu lassen. Mit

```
MACEXP  on
```

wird die vollständige Listingform wieder eingeschaltet, dies ist auch die Default-Vorgabe.

Zwischen der Bedeutung von MACEXP für Makros und der für alle anderen makroartigen Konstrukte (z.B. REPT) besteht ein subtiler Unterschied: Während Makros intern ein Flag besitzen, das anzeigt, ob Expansionen dieses Makros ausgegeben werden sollen oder nicht, wirkt MACEXP direkt auf alle anderen Konstrukte, die ,,vor Ort" aufgelöst werden. Der Sinn dieser Differenzierung besteht darin, daß es Makros geben kann, die ausgetestet sind und die man nicht mehr sehen will, andere aber sehr wohl noch. MACEXP dient hier als Default für das bei der Definition des Makros zu setzende Flag, der mit den Steuerparametern NOEXPAND/EXPAND übersteuert werden kann.

Die momentane Einstellung läßt sich aus dem Symbol MACEXP auslesen.

3.7.4 LISTING

funktioniert wie MACEXP und akzeptiert die gleichen Parameter, arbeitet aber wesentlich radikaler: Mit

```
LISTING off
```

wird *überhaupt* nichts mehr im Listing ausgegeben. Diese Anweisung macht Sinn für erprobte Codeteile oder Includefiles, 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.

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 Listingbeginn 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 angesprochenenen 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!

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 Programmes 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 unübersichlich 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-Programmes 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 3.5 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 3.5: 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 Programmes 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 PUBLICs 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 dann 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.

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 er automatisch um die Endung `INC` erweitert.

Mit der Kommandozeilenoption

```
-i <Pfadliste>
```

läßt sich eine Liste von Verzeichnissen angeben, in denen automatisch zusätzlich nach der Includedatei 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!

Sollte der Dateiname eine Pfadangabe enthalten, so wird die Suchliste ignoriert.

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 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 diffenzierte 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 beliebig (berechneter?!) Stringausdruck, der damit sowohl eine Konstante als auch variabel sein darf.

Diese Anweisungen 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 auf der Konsole bzw. im Listing erzeugt. 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 ,,herumstochern" 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 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.

3.9.7 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 Programmes 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 Adreäusdrücken 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 [57] genannten RS6000-Befehle (die aber hoffentlich keiner vermißt...). Das wird aber nachgeholt, sobald Informationen verfügbar sind!

4.3 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 [52] 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.

4.4 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 ähnlich wie bei Microchip Hochkommas verwendet. Das konnte (und wollte) ich bei AS nicht nachvollziehen, bei dem Hochkommas zur Einrahmung von ASCII-Sequenzen benutzt werden. Anstelledessen werden Zahlen in der üblichen Motorola-Syntax geschrieben, d.h. mit einem Dollarzeichen.

4.5 SH7000/7600/7700

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 Programmes 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 SH7x00 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).

4.6 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  .
```

4.7 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 ensprechenden 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

4.8 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.

4.9 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[41] 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).

4.10 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`.

4.11 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`.

4.12 MCS-48

Der maximale Adreßraum dieser Prozessoren beträgt 4 Kbyte. 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` bzw. `SEL MB1` vorgegeben wird. Um den Wechsel zwischen den Bänken zu vereinfachen, ist eine Automatik in den Befehlen `JMP` und `CALL` eingebaut, die einen dieser beiden Befehle einfügt, falls die Adresse des Sprungbefehles und das Sprungziel in unterschiedlichen Bänken liegen. Die explizite Benutzung der `SEL MBx`-Befehle sollte daher nicht notwendig sein (obwohl sie möglich ist) 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.

4.13 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 Programmes 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.

4.14 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 gutgemeinter 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-Anweisungem, 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).

4.15 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]`
Funktion : HL ← HL - reg
Flags : CY, S, X5, AC, Z, V, P
Argumente : `reg` = B für BC (optional)

Anweisung : `ARHL`
Funktion : HL,CY ← HL >> 1 (arithmetisch)
Flags : CY
Argumente : keine

Anweisung : `RDEL`
Funktion : CY,DE ← DE << 1
Flags : CY, V
Argumente : keine

Anweisung : `LDHI d8`
Funktion : DE ← HL + `d8`
Flags : keine
Argumente : `d8` = 8-Bit-Konstante

Anweisung : `LDSI d8`
Funktion : DE ← SP + `d8`
Flags : keine
Argumente : `d8` = 8-Bit-Konstante

Anweisung : `RST flag`
Funktion : Restart zu 40h wenn `flag`=1
Flags : keine
Argumente : `flag` = V für Overflow-Bit

Anweisung : `SHLX [reg]`
Funktion : [reg] ← HL
Flags : keine
Argumente : `reg` = D für DE (optional)

Anweisung : `LHLX [reg]`
Funktion : HL ←[reg]
Flags : keine
Argumente : `reg` = D für DE (optional)

Anweisung : `JNX5 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.

4.16 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.
- 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.

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 4.1: Unterschiede in der Adressierungssyntax AS↔MASM

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 4.1 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.

4.17 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...

4.18 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.

4.19 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 WAVRSIM 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.

4.20 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 Z380 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 `SLIA`-Befehl zugänglich ist. `SLIA` funktioniert wie `SLA`, es wird jedoch eine Eins und nicht eine Null in Bit 0 eingeschoben. Dieser Befehl kann, wie alle anderen Schiebebefehle auch, noch in einer weiteren Variante geschrieben werden:

```
SLIA    R,(XY+d)
```

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)
```

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...

4.21 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 Aritmetik 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 die mit LW eingestellte Operandengröße für einzelne Befehle mit den Präfixen `DDIR W` und `DDIR LW` übersteuert werden kann. AS merkt sich das Auftreten solcher Befehle und schaltet dann für den nächsten Prozessorbefehl automatisch mit um. Andere `DDIR`-Varianten als `W` und `LW` sollte man übrigens nie explizit verwenden, da AS bei zu langen Operanden diese automatisch einsetzt, und das könnte zu Verwirrungen führen. Die Automatik geht übrigens so weit, daß in der Befehlsfolge

```
DDIR    LW
LD      BC,12345678h
```

automatisch der erforderliche `IW`-Präfix mit in die vorangehende Anweisung hineingezogen wird, effektiv wird also der Code

```
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.

4.22 Z8 und eZ8

Der Prozessorkern der Z8-Mikrokontroller beinhaltet keine eigenen Register. Stattdessen kann ein 16er-Block des internen Adreraums aus RAM und I/O-Registern als 'Arbeitsregister' benutzt werden, die mit 4-Bit-Adressen angesprochen

werden knnen. Welcher 16er-Block als Arbeitsregister benutzt werden soll, wird mit dem RP-Register festgelegt: Bits 4 bis 7 von RP definieren den 'Offset', der auf die 4-Bit-Arbeitsregisteradresse addiert wird, um eine 8-Bit-Adresse zu erhalten.

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 mu. 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
```

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 mglich, 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 zustzlichen 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.

4.23 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...).

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 Integerkonstanten 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[107] genannt werden, aber durch Vertauschung auf eine genannte zurückgeführt 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 Programmes.

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 4.2).

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
`$NOLIST`	`LISTING OFF`	Assemblerlisting ausschalten
`$LIST`	`LISTING ON`	Assemblerlisting einschalten
`$EJECT`	`NEWPAGE`	neue Seite im Listing beginnen

Tabelle 4.2: äquivalente Befehle TAS900↔AS

4.24 TLCS-90

Vielleicht fragt sich der eine oder andere, ob bei mir die Reihenfolge durcheinandergekommen 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.

4.25 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
```

4.26 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.

4.27 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 hat sich leider auch vorläufig dazu entschieden, diesen Prozessor ,,auf Eis" zu legen, bis auf weiteres wird es also auch kein Silizium geben. Das hat 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 [110], noch vorläufig waren, also noch nicht bis ins letzte Klarheit lieferten.

Fehler in diesem Teil sind also durchaus noch möglich (und werden natürlich bereinigt, wenn es denn einmal gehen sollte!). Zumindest die Handvoll Beispiele in [110] werden aber richtig übersetzt.

4.28 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...

4.29 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 CALLs, JMPs oder RETs 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 bitadressierbar) 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
```

4.30 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!!

4.31 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>
```

4.32 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 (3.2.16), in dem steht, wie man mit diesem Befehl einigermaßen unfallfrei Konstanten aus dem ROM lesen kann.

Bei nähererer Betrachtung des Befehlssatzes fallen einige eingebaute ,,Makros" auf. Die Befehle, die mir aufgefallen sind (es gibt aber vielleicht noch mehr...), sind in Tabelle 4.3 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 4.3: versteckte Makros im ST6225-Befehlssatz

ASCII, ASCIZ, BLOCK, BYTE, END, ENDM, EQU, ERROR, MACRO, ORG, TITLE, WARNING

Tabelle 4.4 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	Listing-Einstellung
.PL	PAGE	Seitenlänge Listing
.ROMSIZE	CPU	Zielprozessor einstellen
.VERS	VERSION (Symbol)	Version abfragen
.SET	EVAL	Variablen neu setzen

Tabelle 4.4: äquivalente Befehle AST6↔AS

4.33 ST7

In [82] 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.

4.34 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.

4.35 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!

4.36 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!

4.37 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.

4.38 TMS320C3x

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 Aweisung 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.

4.39 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 `EQU`s.
- 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.

4.40 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!

4.41 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...

4.42 MSP430

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. Bei AS finden sich diese Befehle mit in der Datei `REGMSP.INC`. Wer diese Datei nicht einbindet, wird bei über der Hälfte der insgesamt von TI definierten Befehle Fehlermeldungen bekommen!!

4.43 COP8 & SC/MP

Leider Gottes hat sich auch National dazu entschieden, als Schreibweise für nichtdezimale Integer-Konstanten die von IBM-Großrechnern bekannte (und von mir vielgehaßte) 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...

4.44 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:

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

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!

4.45 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 [74] 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.

4.46 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.

4.47 78K2

Analog wie beim 78K0 benutzt NEC auch hier wieder Dollar- und Ausrufezeichen für verschiedene Längen von Adreßsausdrücken. 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...

4.48 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!

4.49 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 ;-)

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 Header:Byte OF
               $00:(Creator:ARRAY[] OF Char);
               $01..
               $7f:(StartAdr : LongInt;
                    Length   : Word;
                    Data     : ARRAY[0..Length-1] OF Byte);
               $80:(EntryPoint:LongInt);
               $81:(Header   : Byte;
                    Segment  : Byte;
                    Gran     : Byte;
                    StartAdr : LongInt;
                    Length   : Word;
                    Data     : ARRAY[0..Length-1] OF Byte);
             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 Tabellen 5.1 und 5.2).

Das Segment-Feld gibt an, in welchen Adreßraum des Prozessors der folgende Code gehört. Dabei gilt die in Tabelle 5.3 angegeben Zuordnung. Das Gran-Feld 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!

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

Header	Familie	Header	Familie
$01	680x0, 6833x	$02	ATARI_VECTOR
$03	M*Core	$04	XGATE
$05	PowerPC	$06	XCore
$09	DSP56xxx	$11	65xx/MELPS-740
$12	MELPS-4500	$13	M16
$14	M16C	$15	F^{2}MC8L
$16	F^{2}MC16L	$19	65816/MELPS-7700
$21	MCS-48	$25	SYM53C8xx
$27	KENBAK	$29	29xxx
$2a	i960	$31	MCS-51
$32	ST9	$33	ST7
$37	2650	$38	1802/1805
$39	MCS-96/196/296	$3a	8X30x
$3b	AVR	$3c	XA
$3f	4004/4040	$41	8080/8085
$42	8086..V35	$47	TMS320C6x
$48	TMS9900	$49	TMS370xxx
$4a	MSP430	$4b	TMS320C54x
$4c	80C166/167	$51	Z80/180/380
$52	TLCS-900	$53	TLCS-90
$54	TLCS-870	$55	TLCS-47
$56	TLCS-9000	$59	eZ8
$5b	KCPSM3	$5c	LatticeMico8
$5d	NEX 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
$71	PIC16C5x	$72	PIC17C4x

Tabelle 5.1: Headerbytes für die verschiedenen Prozessorfamilien

Header	Familie	Header	Familie
$73	TMS-7000	$74	TMS3201x
$75	TMS320C2x	$76	TMS320C3x
$77	TMS320C20x/TMS320C5x	$78	ST6
$79	Z8	$7a	μPD78(C)10
$7b	75K0	$7c	78K0
$7d	μPD7720	$7e	μPD7725
$7f	μPD77230		

Tabelle 5.2: Headerbytes für die verschiedenen Prozessorfamilien

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 5.3: Kodierungen des `Segment`-Feldes

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 Programmes 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 [4] 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 Programmes 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 5 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 schlußendlich durch eine 0 oder 1 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.

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 voranstehenden Dollarzeichen oder `0x` wie in C versieht (z.B. `$10` oder `0x10` anstelle von 16).

Unix-Shells ordnen dem Dollarzeichen allerdings eine spezielle Bedeutung zu *UNIX*
(Parameterexpansion), weshalb es nötig ist, einem Dollarzeichen direkt einen Backslash voranzustellen. Die `0x`-Variante ist hier sicherlich angenehmer.

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. BINDCMD für BIND), 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 BIND

BIND 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 BIND auch dazu verwendet werden, um eine Codedatei in mehrere aufzuspalten.

Die allgemeine Syntax von BIND lautet

```
BIND <Quelldatei(en)> <Zieldatei> [Optionen]
```

Wie auch AS betrachtet BIND alle nicht mit einem +, - oder / eingeleiteten Parameter als Dateiangaben, von denen die letzte die Zieldatei angeben muß. Alle anderen Dateiangaben bezeichnen Quellen, diese Angaben dürfen auch wieder Jokerzeichen enthalten.

An Optionen definiert BIND momentan nur eine:

- `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 BIND folgendermaßen:

```
BIND <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 BIND. Es besitzt alle Kommandozeilenoptionen von BIND und hat die gleichen Konventionen bzgl. Dateinamen. Im Gegensatz zu BIND wird die Zieldatei aber als Hexfile ausgegeben, d.h. als eine Folge von Zeilen, die den Code als ASCII-Hexzahlen enthalten.

P2HEX kennt 8 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 [4])

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.

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 Programiergeräte. 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 [8] 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.

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.

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
```

Defaultmäßig ist das Fenster 32 KByte groß und beginnt bei Adresse 0.

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.

Als Sonderwert für Start-und Endadresse beim r-Parameter ist ein schlichtes Dollar-Zeichen ($) bzw. ein '0x' erlaubt. Dies kennzeichnet die erste bzw. letzte in der Programmdatei belegte Adresse. Wer also sicher sein will, daß immer das ganze Programm in der Hex-Datei abgelegt wird, braucht sich mit dem Schalter

```
-r $-$
```

keine Gedanken mehr zu machen. Dollarzeichen und feste Adressen lassen sich selbstverständlich auch gemischt verwenden, z.B. kann mit

```
-r $-$7fff
```

das obere Ende auf die ersten 32K begrenzt werden.

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.

Da das TI-DSK-Format Daten und Code unterscheiden kann, läßt sich mit dem Schalter

```
-d <Start>-<Ende>
```

festlegen, welche Adreßbereiche als Daten ausgegeben werden sollen. Dollarzeichen sind hier **nicht** zugelassen. Diese Option sollte nicht mehr in neuen Projekten verwendet werden, da P2HEX inzwischen direkt Daten aus dem Datensegment korrekt umsetzen kann.

Für das DSK- sowie Intel- und Motorola-Format relevant ist dagegen 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 BIND erzeugt P2HEX keine Leerdatei, 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.

- `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 annnehmen:

  - `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.

0 Displacement=0, überflüssig

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
bei 680x0-, 6502- und 68xx-Prozessoren können bestimmte Speicherbereiche mit kurzen Adressen erreicht werden. Um keine Phasefehler zu erzeugen, wird zwar der kürzere Ausdruck erzeugt, der freie Platz wird aber mit NOPs aufgefüllt.

Argument:
keines

20 kurzer Sprung möglich

Type:
Warnung

Reason:
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

30 kein Sharefile angelegt, SHARED ignoriert

Type:
Warnung

Reason:
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)

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
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)

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
Bei der Bildung der Belegungsliste wurde festgestellt, daß ein Speicherbereich im Codesegment mehrfach benutzt wurde. Ursache können unüberlegte `ORG`-Anweisungen sein.

Argument:
keines

100 keine CASE-Bedingung zugetroffen

Type:
Warnung

Reason:
bei einem `SWITCH`..`CASE`-Konstrukt ohne `ELSECASE`-Zweig traf keiner der `CASE`-Zweige zu.

Argument:
keines

110 Seite möglicherweise nicht adressierbar

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
Die Hardware erlaubt nur ein Registerpaar zu verketten, dessen Startadresse gerade ist (RR0, RR2..., nur Z8).

Argument:
keines

130 veralteter Befehl

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
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

150 Lokaloperator außerhalb einer Sektion überflüssig

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
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

260 Bereichsüberschreitung

Type:
Warnung

Reason:
Ein Zahlenwert lag außerhalb 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

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
Ein einzelner X-Operand kann sowohl als Register X als auch X-inidizierte Adressierung mit Null-Displacement interpretiert werden, da sich Morola hier nicht festlegt. AS wählt die letztere Variante, was möglicherweise nicht das erwartete ist.

Argument:
keines

300 Bit-Nummer wird abgeschnitten werden

Type:
Warnung

Reason:
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

Type:
Warnung

Reason:
Gültige bzw. sinnvolle Werte fr den Registerzeiger sind nur Werte von 0x00...0x70 bzw. 0xf0, weil die anderen Registerbereiche unbelegt sind.

Argument:
keines

1000 Symbol doppelt definiert

Type:
Fehler

Reason:
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

Type:
Fehler

Reason:
Ein benutztes Symbol ist auch im 2.Pass noch nicht in der Symboltabelle enthalten.

Argument:
Name des nicht gefundenen Symbols

1020 Ungültiger Symbolname

Type:
Fehler

Reason:
Ein Symbolname entspricht nicht den Bedingungen für einen gültigen Symbolnamen. Beachten Sie, daß für Makro-und Funktionsparameter strengere Regeln gelten!

Argument:
der fehlerhafte Symbolname

1090 Ungültiges Format

Type:
Fehler

Reason:
Das benutzte Befehlsformat existiert bei diesem Befehl nicht.

Argument:
Der Kennbuchstabe des verwendeten Formates

1100 Überflüssiges Attribut

Type:
Fehler

Reason:
Der benutzte Befehl (Prozessor oder Pseudo) darf kein mit einem Punkt angehängtes Attribut haben.

Argument:
keines

1105 Attribut darf nur 1 Zeichen lang sein

Type:
Fehler

Reason:
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

Type:
Fehler

Reason:
Das an einem Befehl angefügte Attribut ist ungültig.

Argument:
keines

1110 Unpassende Operandenzahl

Type:
Fehler

Reason:
Die bei einem Befehl (Prozessor oder Pseudo) angegebene Operandenzahl liegt nicht in dem für diesen Befehl erlaubten Bereich.

Argument:
keines

1115 Unpassende Optionenzahl

Type:
Fehler

Reason:
Die bei diesem Befehl angegebene Zahl von Optionen liegt nicht in dem für diesen Befehl erlaubten Bereich.

Argument:
keines

1120 nur immediate-Adressierung erlaubt

Type:
Fehler

Reason:
Der benutzte Befehl läßt nur immediate-Operanden (mit vorangestelltem #) zu.

Argument:
keines

1130 Unpassende Operandengröße

Type:
Fehler

Reason:
Der Operand hat zwar einen für den Befehl zugelassenen Typ, jedoch nicht die richtige Länge (in Bits).

Argument:
keines

1131 Widersprechende Operandengrößen

Type:
Fehler

Reason:
Die angegebenen Operanden haben unterschiedliche Längen (in Bit).

Argument:
keines

1132 Undefinierte Operandengröße

Type:
Fehler

Reason:
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

1135 Ungültiger Operandentyp

Type:
Fehler

Reason:
Ein Ausdruck hat einen an dieser Stelle nicht zulässigen Typ (Integer/Gleitkomma/String).

Argument:
Die an dieser Stelle zulässigen Datentypen

1140 Zuviele Argumente

Type:
Fehler

Reason:
Einem Befehl wurden mehr als die unter AS zulässigen 20 Parameter übergeben.

Argument:
keines

1200 Unbekannter Befehl

Type:
Fehler

Reason:
Der benutzte Befehl ist weder ein Pseudobefehl von AS noch ein Befehl des momentan eingestellten Prozessors.

Argument:
keines

1300 Klammerfehler

Type:
Fehler

Reason:
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

Type:
Fehler

Reason:
Bei einer Division oder Modulooperation ergab die Auswertung des rechten Teilausdruckes 0.

Argument:
keines

1315 Bereichsunterschreitung

Type:
Fehler

Reason:
Der angegebene Integer-Wert unterschreitet den zulässigen Bereich.

Argument:
aktueller Wert und zulässiges Minimum (manchmal, ich stelle das gerade um...)

1320 Bereichsüberschreitung

Type:
Fehler

Reason:
Der angegebene Integer-Wert überschreitet den zulässigen Bereich.

Argument:
aktueller Wert und zulässiges Maximum (manchmal,ich stelle das gerade um...)

1325 Adresse nicht ausgerichtet

Type:
Fehler

Reason:
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

Type:
Fehler

Reason:
Der in einem Adreßausdruck enthaltene Displacement-Wert ist zu groß.

Argument:
keines

1340 Kurzadressierung nicht möglich

Type:
Fehler

Reason:
Die Adresse des Operanden liegt außerhalb des Speicherbereiches, in dem Kurzadressierung möglich ist.

Argument:
keines

1350 Unerlaubter Adressierungsmodus

Type:
Fehler

Reason:
Der benutzte Adressierungsmodus existiert generell zwar, ist an dieser Stelle aber nicht erlaubt.

Argument:
keines

1351 Nummer muß ausgerichtet sein

Type:
Fehler

Reason:
An dieser Stelle sind nur ausgerichtete (d.h z.B. gerade) Adressen erlaubt, da die untersten Bits für andere Zwecke verwendet werden oder reserviert sind.

Argument:
keines

1355 Adressierungsmodus im Parallelbetrieb nicht erlaubt

Type:
Fehler

Reason:
Die verwendeten Adressierungsmodi sind zwar im sequentiellen Modus zulässig, jedoch nicht bei parallelen Instruktionen.

Argument:
keines

1360 Undefinierte Bedingung

Type:
Fehler

Reason:
Die benutzte Bedingung für bedingte Sprünge existiert nicht.

Argument:
keines

1365 inkompatible Bedingungen

Type:
Fehler

Reason:
Die benutzte Kombination von Bedingungen kann nicht in einem Befehl verwendet werden.

Argument:
die Bedingung, bei der die Unverträglichkeit entdeckt wurde.

1370 Sprungdistanz zu groß

Type:
Fehler

Reason:
Sprungbefehl und Sprungziel liegen zu weit auseinander, um mit einem Sprung der benutzten Länge überbrückt werden zu können.

Argument:
keines

1375 Sprungdistanz ist ungerade

Type:
Fehler

Reason:
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

1380 ungültiges Schiebeargument

Type:
Fehler

Reason:
als Argument für die Schiebeamplitude darf nur eine Konstante oder ein Datenregister verwendet werden. (nur 680x0)

Argument:
keines

1390 Nur Bereich 1..8 erlaubt

Type:
Fehler

Reason:
Konstanten für Schiebeamplituden oder `ADDQ`-Argumente dürfen nur im Bereich 1..8 liegen. (nur 680x0)

Argument:
keines

1400 Schiebezahl zu groß

Type:
Fehler

Reason:
(nicht mehr verwendet)

Argument:
keines

1410 Ungültige Registerliste

Type:
Fehler

Reason:
Das Registerlisten-Argument von `MOVEM` oder `FMOVEM` hat ein falsches Format. (nur 680x0)

Argument:
keines

1420 Ungültiger Modus mit CMP

Type:
Fehler

Reason:
Die verwendete Operandenkombination von `CMP` ist nicht erlaubt. (nur 680x0)

Argument:
keines

1430 Ungültiger Prozessortyp

Type:
Fehler

Reason:
Den mit CPU angeforderten Zielprozessor kennt AS nicht.

Argument:
der unbekannte Prozessortyp

1440 Ungültiges Kontrollregister

Type:
Fehler

Reason:
Das bei z.B. `MOVEC` benutzte Kontrollregister kennt der mit CPU gesetzte Prozessor (noch) nicht.

Argument:
keines

1445 Ungültiges Register

Type:
Fehler

Reason:
Das benutzte Register ist zwar prinzipiell vorhanden, hier aber nicht erlaubt.

Argument:
keines

1450 RESTORE ohne SAVE

Type:
Fehler

Reason:
Es wurde ein `RESTORE`-Befehl gefunden, obwohl kein mit `SAVE` gespeicherter Zustand (mehr) auf dem Stapel vorhanden ist.

Argument:
keines

1460 fehlendes RESTORE

Type:
Fehler

Reason:
Nach der Assemblierung sind nicht alle `SAVE`-Befehle wieder aufgelöst worden.

Argument:
keines

1465 unbekannte Makro-Steueranweisung

Type:
Fehler

Reason:
Eine beim `MACRO`-Befehl zusätzlich angegebene Steueranweisung ist AS unbekannt.

Argument:
die fragliche Anweisung

1470 fehlendes ENDIF/ENDCASE

Type:
Fehler

Reason:
Nach der Assemblierung sind nicht alle Konstrukte zur bedingten Assemblierung aufgelöst worden.

Argument:
keines

1480 ungültiges IF-Konstrukt

Type:
Fehler

Reason:
Die Reihenfolge der Befehle in einem `IF`- oder `SWITCH`-Konstrukt stimmt nicht.

Argument:
keines

1483 doppelter Sektionsname

Type:
Fehler

Reason:
Es existiert bereits eine Sektion gleichen Namens auf dieser Ebene.

Argument:
der doppelte Name

1484 unbekannte Sektion

Type:
Fehler

Reason:
Im momentanen Sichtbarkeitsbereich existiert keine Sektion dieses Namens.

Argument:
der unbekannte Name

1485 fehlendes ENDSECTION

Type:
Fehler

Reason:
Nach Ende eines Durchganges sind nicht alle Sektionen wieder geschlossen worden.

Argument:
keines

1486 falsches ENDSECTION

Type:
Fehler

Reason:
die bei `ENDSECTION` angegebene Sektion ist nicht die innerste offene.

Argument:
keines

1487 ENDSECTION ohne SECTION

Type:
Fehler

Reason:
Es wurde ein `ENDSECTION`-Befehl gegeben, obwohl gar keine Sektion offen war.

Argument:
keines

1488 nicht aufgelöste Vorwärtsdeklaration

Type:
Fehler

Reason:
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

Type:
Fehler

Reason:
Ein Symbol wurde sowohl als privat als auch global definiert.

Argument:
der Name des Symbols

1490 falsche Argumentzahl für Funktion

Type:
Fehler

Reason:
Die Anzahl der Argumente für eine selbstdefinierte Funktion stimmt nicht mit der geforderten Anzahl überein.

Argument:
keines

1495 unaufgelöste Literale (LTORG fehlt)

Type:
Fehler

Reason:
Am Programmende oder beim Umachalten zu einem anderen Zielprozessor blieben noch nicht abgelegte Literale übrig.

Argument:
keines

1500 Befehl auf dem ... nicht vorhanden

Type:
Fehler

Reason:
Der benutzte Befehl existiert zwar grundsätzlich, das eingestellte Mitglied der Prozessorfamilie beherrscht ihn aber noch nicht.

Argument:
keines

1505 Adressierungsart auf dem ... nicht vorhanden

Type:
Fehler

Reason:
Der benutzte Adressierungsmodus existiert zwar grundsätzlich, das eingestellte Mitglied der Prozessorfamilie beherrscht ihn aber noch nicht.

Argument:
keines

1510 Ungültige Bitstelle

Type:
Fehler

Reason:
Die angegebene Bitnummer ist nicht erlaubt oder eine Angabe fehlt komplett.

Argument:
keines

1520 nur ON/OFF erlaubt

Type:
Fehler

Reason:
Dieser Pseudobefehl darf als Argument nur `ON` oder `OFF` haben.

Argument:
keines

1530 Stack ist leer oder nicht definiert

Type:
Fehler

Reason:
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

Type:
Fehler

Reason:
In einer Bitmaske, die der `BITPOS`- Funktion übergeben wurde, war nicht genau ein Bit gesetzt.

Argument:
keines

1550 ENDSTRUCT ohne STRUCT

Type:
Fehler

Reason:
Eine `ENDSTRUCT`-Anweisung wurde gegeben, obwohl momentan keine Strukturdefinition in Gange war.

Argument:
keines

1551 offene Strukturdefinition

Type:
Fehler

Reason:
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

Type:
Fehler

Reason:
Der Namensparameter einer `ENDSTRUCT`-Anweisung entspricht nicht der innersten, offenen Strukturdefinition.

Argument:
keines

1553 Phasendefinition nicht in Strukturen erlaubt

Type:
Fehler

Reason:
Was gibt es dazu zu sagen? `PHASE` in einem Record ergibt einfach keinen Sinn und nur Verwirrung...

Argument:
keines

1554 ungültige `STRUCT`-Direktive

Type:
Fehler

Reason:
Als Direktive für `STRUCT` ist nur `EXTNAMES` oder `NOEXTNAMES` zugelassen.

Argument:
die unbekannte Direktive

1560 Anweisung nicht wiederholbar

Type:
Fehler

Reason:
Diese Maschinenanweisung kann nicht mit Hilfe eines `RPT`-Konstruktes wiederholt werden.

Argument:
keines

1600 vorzeitiges Dateiende

Type:
Fehler

Reason:
Es wurde mit einem `BINCLUDE`-Befehl versucht, über das Ende einer Datei hinauszulesen.

Argument:
keines

1700 ROM-Offset geht nur von 0..63

Type:
Fehler

Reason:
Das Konstanten-ROM der 680x0-Koprozessoren hat nur max. 63 Einträge.

Argument:
keines

1710 Ungültiger Funktionscode

Type:
Fehler

Reason:
Als Funktionscodeargument darf nur SFC, DFC, ein Datenregister oder eine Konstante von 0..15 verwendet werden. (nur 680x0-MMU)

Argument:
keines

1720 Ungültige Funktionscodemaske

Type:
Fehler

Reason:
Als Funktionscodemaske darf nur ein Wert von 0..15 verwendet werden. (nur 680x0-MMU)

Argument:
keines

1730 Ungültiges MMU-Register

Type:
Fehler

Reason:
Die MMU hat kein Register mit dem angegebenen Namen. (nur 680x0-MMU)

Argument:
keines

1740 Level nur von 0..7

Type:
Fehler

Reason:
Die Ebene für `PTESTW` und `PTESTR` muß eine Konstante von 0..7 sein. (nur 680x0-MMU)

Argument:
keines

1750 ungültige Bitmaske

Type:
Fehler

Reason:
Die bei den Bit-Feld-Befehlen angegebene Bitmaske hat ein falsches Format. (nur 680x0)

Argument:
keines

1760 ungültiges Registerpaar

Type:
Fehler

Reason:
Das angegebene Registerpaar ist hier nicht verwendbar oder syntaktisch falsch. (nur 680x0)

Argument:
keines

1800 offene Makrodefinition

Type:
Fehler

Reason:
Eine Makrodefinition war am Dateiende nicht zuende. Vermutlich fehlt ein `ENDM`.

Argument:
keines

1805 EXITM außerhalb eines Makrorumpfes

Type:
Fehler

Reason:
`EXITM` bricht die Expansion von Makro-Konstrukten ab. Dieser Befehl macht nur innerhalb von Makros Sinn und es wurde versucht, ihn außerhalb aufzurufen.

Argument:
keines

1810 mehr als 10 Makroparameter

Type:
Fehler

Reason:
Ein Makro darf höchstens 10 Parameter haben.

Argument:
keines

1815 doppelte Makrodefinition

Type:
Fehler

Reason:
Ein Makronamne wurde in einer Sektion doppelt vergeben.

Argument:
der doppelt verwendete Name

1820 Ausdruck muß im ersten Pass berechenbar sein

Type:
Fehler

Reason:
Der benutzte Befehl beeinflußt die Codelänge, daher sind Vorwärtsreferenzen hier nicht erlaubt.

Argument:
keines

1830 zu viele verschachtelte IFs

Type:
Fehler

Reason:
(nicht mehr verwendet)

Argument:
keines

1840 ELSEIF/ENDIF ohne ENDIF

Type:
Fehler

Reason:
es wurde ein `ELSEIF`- oder `ENDIF`-Befehl gefunden, obwohl kein offener `IF`-Befehl vorhanden ist.

Argument:
keines

1850 verschachtelter/rekursiver Makroaufruf

Type:
Fehler

Reason:
(nicht mehr verwendet)

Argument:
keines

1860 unbekannte Funktion

Type:
Fehler

Reason:
Die angesprochene Funktion ist weder eingebaut noch nachträglich definiert worden.

Argument:
der Funktionsname

1870 Funktionsargument außerhalb Definitionsbereich

Type:
Fehler

Reason:
Das Argument liegt nicht im Bereich der angesprochenen transzendenten Funktion.

Argument:
keines

1880 Gleitkommaüberlauf

Type:
Fehler

Reason:
Das Argument liegt zwar im Bereich der angesprochenen transzendenten Funktion, das Ergebnis wäre aber nicht mehr darstellbar.

Argument:
keines

1890 ungültiges Wertepaar

Type:
Fehler

Reason:
Das benutzte Pärchen aus Basis und Exponent kann nicht berechnet werden.

Argument:
keines

1900 Befehl darf nicht auf dieser Adresse liegen

Type:
Fehler

Reason:
Die Prozessorhardware erlaubt keine Sprünge von dieser Adresse.

Argument:
keines

1905 ungültiges Sprungziel

Type:
Fehler

Reason:
Die Prozessorhardware erlaubt keine Sprünge zu dieser Adresse.

Argument:
keines

1910 Sprungziel nicht auf gleicher Seite

Type:
Fehler

Reason:
Sprungbefehl und Sprungziel müssen bei diesem Befehl auf der gleichen Seite liegen.

Argument:
keines

1920 Codeüberlauf

Type:
Fehler

Reason:
Es wurde versucht, mehr als 1024 Bytes Code oder Daten in einer Zeile zu erzeugen.

Argument:
keines

1925 Adreßüberlauf

Type:
Fehler

Reason:
Der Adreßraum dieses Prozessors wurde überschritten.

Argument:
keines

1930 Konstanten und Platzhalter nicht mischbar

Type:
Fehler

Reason:
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

Type:
Fehler

Reason:
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

Type:
Fehler

Reason:
Entweder sind die beiden Instruktionen prinzipiell nicht parallel ausführbar, oder sie stehen nicht unmittelbar untereinander.

Argument:
keines

1960 ungültiges Segment

Type:
Fehler

Reason:
Das angegebene Segment ist an dieser Stelle nicht anwendbar.

Argument:
der benutzte Segmentname

1961 unbekanntes Segment

Type:
Fehler

Reason:
Das angegebene Segment existiert bei diesem Prozessor nicht.

Argument:
der benutzte Segmentname

1962 unbekanntes Segmentregister

Type:
Fehler

Reason:
Das angegebene Segmentregister existiert nicht (nur 8086).

Argument:
keines

1970 ungültiger String

Type:
Fehler

Reason:
Der angegebene String hat ein ungültiges Format.

Argument:
keines

1980 ungültiger Registername

Type:
Fehler

Reason:
Das angegebene Register existiert nicht oder darf hier nicht verwendet werden.

Argument:
keines

1985 ungültiges Argument

Type:
Fehler

Reason:
Der angegebene Befehl darf nicht mit einem `REP`-Präfix versehen werden.

Argument:
keines

1990 keine Indirektion erlaubt

Type:
Fehler

Reason:
in dieser Kombination ist keine indirekte Adressierung erlaubt.

Argument:
keines

1995 nicht im aktuellen Segment erlaubt

Type:
Fehler

Reason:
(nicht mehr verwendet)

Argument:
keines

1996 nicht im Maximum-Modus zulässig

Type:
Fehler

Reason:
Dieses Register ist nur im Minimum-Modus definiert.

Argument:
keines

1997 nicht im Minimum-Modus zulässig

Type:
Fehler

Reason:
Dieses Register ist nur im Maximum-Modus definiert.

Argument:
keines

2000 Anweisungspaket überschreitet Adreßgrenze

Type:
Fehler

Reason:
Ein Anweisungspaket dard nicht über eine 32-Byte-Adreßgrenze reichen.

Argument:
keines

2001 Ausführungseinheit mehrfach benutzt

Type:
Fehler

Reason:
Eine der Ausführungseinheiten des Prozessors wurde in einem Anweisungspaket mehrfach benutzt.

Argument:
der Name der Funktionseinheit

2002 mehrfache Lang-Leseoperation

Type:
Fehler

Reason:
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

Type:
Fehler

Reason:
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

Type:
Fehler

Reason:
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

Type:
Fehler

Reason:
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

Type:
Fehler

Reason:
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

Type:
Fehler

Reason:
Ein Anweisungspaket beinhaltet mehr als einen direkten Sprung, was nicht erlaubt ist.

Argument:
keines

2009 Anweisung nicht auf diese Funktionseinheit ausführbar

Type:
Fehler

Reason:
Diese Anweisung kann nicht auf dieser Funktionseinheit ausgeführt werden.

Argument:
none

2010 Ungültige Escape-Sequenz

Type:
Fehler

Reason:
Das mit einem Backslash eingeleitete Sonderzeichen ist nicht definiert.

Argument:
keines

2020 ungültige Präfix-Kombination

Type:
Fehler

Reason:
Die angegebene Kombination von Präfixen ist nicht zulässig oder nicht im Maschinenkode darstellbar.

Argument:
keines

2030 Konstante kann nicht als Variable redefiniert werden

Type:
Fehler

Reason:
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

Type:
Fehler

Reason:
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

Type:
Fehler

Reason:
Bei einer Strukturdefinition fehlt der zugehörende Name für die Struktur.

Argument:
keines

2050 leeres Argument

Type:
Fehler

Reason:
In der Argumentenliste dieser Anweisung dürfen keine Leerstrings benutzt werden.

Argument:
keines

10001 Fehler bein Öffnen der Datei

Type:
fatal

Reason:
Beim Versuch, eine Datei zu öffnen, ist ein Fehler aufgetreten.

Argument:
Beschreibung des E/A-Fehlers

10002 Listingschreibfehler

Type:
fatal

Reason:
Beim Schreiben des Assemblerlistings ist ein Fehler aufgetreten.

Argument:
Beschreibung des E/A-Fehlers

10003 Dateilesefehler

Type:
fatal

Reason:
Beim Lesen aus einer Quelldatei ist ein Fehler aufgetreten.

Argument:
Beschreibung des E/A-Fehlers

10004 Dateischreibfehler

Type:
fatal

Reason:
Beim Schreiben von Code- oder Share-Datei ist ein Fehler aufgetreten.

Argument:
Beschreibung des E/A-Fehlers

10006 Speicherüberlauf

Type:
fatal

Reason:
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

Type:
fatal

Reason:
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

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. Assemblerlistings 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

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 reicht der Filter von 0 bis 32 Kbyte, falls Ihr Programm Teile außerhalb dieses Bereiches besitzen sollte, werden diese nicht übernommen. Sollte Ihr Code komplett jenseits 32 Kbyte liegen (wie es bei 65er und 68er-Prozessoren üblich ist), dann erhalten Sie das von Ihnen geschilderte Ergebnis. Setzen Sie das Adreßfilter einfach auf einen passenden Bereich (s. das Kapitel zu P2BIN/P2HEX).

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-Programmes 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
```

Anhang D

Pseudobefehle 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
ERROR	EQU	EXITM	FATAL	FORWARD
FUNCTION	GLOBAL	IF	IFB	IFDEF
IFEXIST	IFNB	IFNDEF	IFNEXIST	IFNUSED
IFUSED	INCLUDE	IRP	LABEL	LISTING
MACEXP	MACRO	MESSAGE	NEWPAGE	ORG
PAGE	PHASE	POPV	PUSHV	PRTEXIT
PRTINIT	PUBLIC	READ	RELAXED	REPT
RESTORE	RORG	SAVE	SECTION	SEGMENT
SHARED	STRUC	STRUCT	SWITCH	TITLE
UNION	WARNING	WHILE		

Zusätzlich existiert SET bzw. EVAL, falls SET bereits ein Prozessorbefehl ist.

Motorola 680x0

DC[.<size>]	DS[.<size>]	FULLPMMU	FPU	PADDING
PMMU	SUPMODE			

Motorola 56xxx

DC	DS	XSFR	YSFR

PowerPC

BIGENDIAN	DB	DD	DQ	DS
DT	DW	REG	SUPMODE	

Motorola M-Core

DC[.<size>]	DS[.<size>]	REG	SUPMODE

Motorola XGATE

ADR	BYT	DC[.<size>]	DFS	DS[.<size>]
FCB	FCC	FDB	PADDING	RMB

Motorola 68xx/Hitachi 63xx

ADR	BYT	DC[.<size>]	DFS	DS[.<size>]
FCB	FCC	FDB	PADDING	RMB

Motorola/Freescale 6805/68HC(S)08

```
ADR             BYT             DC[.<size>]  DFS             DS[.<size>]
FCB             FCC             FDB          PADDING         RMB
```

Motorola 6809/Hitachi 6309

```
ADR             ASSUME          BYT          DC[.<size>]
DFS             DS[.<size>]     FCB          FCC
FDB             PADDING         RMB
```

Motorola 68HC12

```
ADR             BYT             DC[.<size>]  DFS             DS[.<size>]
FCB             FCC             FDB          PADDING         RMB
```

Motorola 68HC16

```
ADR             ASSUME          BYT          DC[.<size>]  DFS
DS[.<size>]  FCB             FCC          FDB          PADDING
RMB
```

Freescale 68RS08

```
ADR             ASSUME          BYT          DC[.<size>]  DFS
DS[.<size>]  FCB             FCC          FDB          PADDING
```

Hitachi H8/300(L/H)

```
DC[.<size>]  DS[.<size>]  MAXMODE       PADDING
```

Hitachi H8/500

```
ASSUME      DC[.<size>]  DS[.<size>]  MAXMODE      PADDING
```

Hitachi SH7x00

```
COMPLITERALS  DC[.<size>]  DS[.<size>]  LTORG        PADDING
SUPMODE
```

65xx/MELPS-740

```
ADR           ASSUME        BYT           DFS           FCB
FCC           FDB           RMB
```

65816/MELPS-7700

```
ADR           ASSUME        BYT           DB            DD
DQ            DS            DT            DW            DFS
FCB           FCC           FDB           RMB
```

Mitsubishi MELPS-4500

```
DATA          RES           SFR
```

Mitsubishi M16

```
DB            DD            DQ            DS            DT
DW
```

Mitsubishi M16C

```
DB              DD              DQ              DS              DT
DW
```

Intel 4004

```
DATA            DS              REG
```

Intel 8008

```
DB              DD              DQ              DS              DT
DW
```

Intel MCS-48

```
DB              DD              DQ              DS              DT
DW
```

Intel MCS-(2)51

```
BIGENDIAN       BIT             DB              DD              DQ
DS              DT              DW              PORT            SFR
SFRB            SRCMODE
```

Intel MCS-96

```
ASSUME          DB              DD              DQ              DS
DT              DW
```

Intel 8080/8085

```
DATA        DS
```

Intel 8080/8085

```
DB          DD          DQ          DS          DT
DW          PORT
```

Intel i960

```
DB          DD          DQ          DS          DT
DW                      FPU         SPACE       SUPMODE
WORD
```

Signetics 8X30x

```
LIV         RIV
```

Signetics 2650

```
DB          DD          DQ          DS          DT
DW
```

Philips XA

```
ASSUME      BIT         DB          DC[.<size>] DD
DQ          DS[.<size>] DT          DW          PADDING
PORT        SUPMODE
```

Atmel AVR

DATA	PACKING	PORT	REG	RES

AMD 29K

ASSUME	DB	DD	DQ	DS
DT	DW	EMULATED	SUPMODE	

Siemens 80C166/167

ASSUME	BIT	DB	DD	DQ
DS	DT	DW	REG	

Zilog Zx80

DB	DD	DEFB	DEFW	DQ
DS	DT	DW	EXTMODE	LWORDMODE

Zilog Z8

DB	DD	DQ	DS	DT
DW	REG	SFR		

Xilinx KCPSM

CONSTANT	NAMEREG	REG

Xilinx KCPSM3

```
CONSTANT        DB              DD              DQ              DS
DT              DW              NAMEREG         PORT            REG
```

LatticeMico8

```
DB              DD              DQ              DS              DT
DW              PORT            REG
```

Toshiba TLCS-900

```
DB              DD              DQ              DS              DT
DW              MAXIMUM         SUPMODE
```

Toshiba TLCS-90

```
DB              DD              DQ              DS              DT
DW
```

Toshiba TLCS-870

```
DB              DD              DQ              DS              DT
DW
```

Toshiba TLCS-47(0(A))

```
ASSUME          DB              DD              DQ              DS
DT              DW              PORT
```

Toshiba TLCS-9000

```
DB              DD              DQ              DS              DT
DW
```

Microchip PIC16C5x

```
DATA            RES             SFR             ZERO
```

Microchip PIC16C8x

```
DATA            RES             SFR             ZERO
```

Microchip PIC17C42

```
DATA            RES             SFR             ZERO
```

SGS-Thomson ST6

```
ASCII           ASCIZ           ASSUME          BYTE            BLOCK
SFR             WORD
```

SGS-Thomson ST7

```
DC[.<size>]  DS[.<size>]  PADDING
```

SGS-Thomson ST9

```
ASSUME          BIT             DB              DD              DQ
DS              DT              DW              REG
```

6804

```
ADR             BYT             DFS             FCB             FCC
FDB             RMB             SFR
```

Texas TM3201x

```
DATA            PORT            RES
```

Texas TM32C02x

```
BFLOAT          BSS             BYTE            DATA            DOUBLE
EFLOAT          TFLOAT          LONG            LQxx            PORT
Qxx             RES             RSTRING         STRING          WORD
```

Texas TMS320C3x

```
ASSUME          BSS             DATA            EXTENDED        SINGLE
WORD
```

Texas TM32C020x/TM32C05x/TM32C054x

```
BFLOAT          BSS             BYTE            DATA            DOUBLE
EFLOAT          TFLOAT          LONG            LQxx            PORT
Qxx             RES             RSTRING         STRING          WORD
```

Texas TMS320C6x

```
BSS             DATA            DOUBLE          SINGLE
WORD
```

Texas TMS9900

```
BSS             BYTE            PADDING         WORD
```

Texas TMS70Cxx

```
DB              DD              DQ              DS              DT
DW
```

Texas TMS370

```
DB              DBIT            DD              DQ              DS
DT              DW
```

Texas MSP430

```
BSS             BYTE            PADDING         WORD
```

National SC/MP

```
DB              DD              DQ              DS              DT
DW
```

National INS807x

```
DB              DD              DQ              DS              DT
DW
```

National COP4

```
ADDR    ADDRW   BYTE    DB      DD
DQ      DS      DSB     DSW     DT
FB      FW      SFR     WORD
```

National COP8

```
ADDR    ADDRW   BYTE    DB      DD
DQ      DS      DSB     DSW     DT
FB      FW      SFR     WORD
```

National COP8

```
DC      DC8     DS      DS8     DS16
DW      DW16
```

Fairchild ACE

```
DB      DD      DQ      DS      DT
DW
```

NEC μPD78(C)1x

```
ASSUME  DB      DD      DQ      DS
DT      DW
```

NEC 75xx

```
DB      DD      DQ      DS      DT
DW
```

NEC 75K0

```
ASSUME          BIT             DB              DD              DQ
DS              DT              DW              SFR
```

NEC 78K0

```
DB              DD              DQ              DS              DT
DW
```

NEC 78K2

```
BIT             DB              DD              DQ              DS
DT              DW
```

NEC μPD772x

```
DATA            RES
```

NEC μPD772x

```
DS              DW
```

Symbios Logic SYM53C8xx

Fujitsu F^{2}MC8L

```
DB              DD              DQ              DS              DT
DW
```

Fujitsu F^{2}MC16L

DB	DD	DQ	DS	DT
DW				

Mitsubishi M16C

DB	DD	DQ	DS	DT
DW				

XMOS XS1

DB	DD	DQ	DS	DT
DW	REG			

Anhang E

Vordefinierte Symbole

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...

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-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 Makrokon-strukten im Listing freigeschaltet ?

Tabelle E.1: Vordefinierte Symbole - Teil 1

Name	Datentyp	Definition	Bedeutung
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?
*	Integer	Spezial	mom. Programmzähler (Motorola, Rockwell, Microchip, Hitachi)
$	Integer	Spezial	mom. Programmzähler (Intel, Zilog, Texas, Toshiba, NEC, Siemens, AMD)

Tabelle E.2: Vordefinierte Symbole - Teil 2

Anhang F

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:

F.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;

F.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 G

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

Wenn man sich entschließt, ein solches Kapitel neu zu schreiben, nachdem es eigentlich schon zwei Jahre veraltet ist, läuft man automatisch Gefahr, daß dabei der eine oder andere gute Geist, der etwas zum bisherigen Gelingen dieses Projektes beigetragen hat, vergessen wird. Der allererste Dank gebührt daher allen Personen, die ich in der folgenden Aufzählung unfreiwillig unterschlagen habe!

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 ist er aber mein Chef und wirft ständig vier Augen auf das, was ich tue. Bei AS scheint zumindest ein lachendes dabei zu sein...

Ohne Datenbücher und Unterlagen zu Prozessoren ist ein Programm wie AS nicht zu machen. Ich habe von einer enormen Anzahl von Leuten Informationen bekommen, die von einem kleinen Tip bis zu ganzen Datenbüchern reichen. Hier eine Aufzählung (wie oben gesagt, ohne Garantie auf Vollständigkeit!):

Ernst Ahlers, Charles Altmann, Marco Awater, Len Bayles, Andreas Bolsch, Rolf Buchholz, Bernd Casimir, Nils Eilers, Gunther Ewald, Stephan Hruschka, Peter Kliegelhöfer, Ulf Meinke, Matthias Paul, Norbert Rosch, Steffen Schmid, Leonhard Schneider, Ernst Schwab, Michael Schwingen, Oliver Sellke, Christian Stelter, Patrik Strömdahl, Oliver Thamm, Thorsten Thiele, Andreas Wassatsch, John Weinrich.

...und ein gehässiger Dank an Rolf-Dieter-Klein und Tobias Thiel, die mit ihren ASM68K demonstrierten, wie man es **nicht** machen sollte und mich damit indirekt dazu angeregt haben, etwas besseres zu schreiben!

So ganz allein habe ich AS nicht verzapft. AS enthält 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...

Anhang H

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;
  - Selbstdefinierte 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 Assemblerlistings 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 Integerkonstanten 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 case-sensitiven Modus umzuschalten, in dem Namen von Symbolen, selbstdefinierten 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 Adreängaben 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^{2}MC8L-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^{2}MC16L-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;

Anhang I

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.

I.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:

I.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 Ungebung 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...).

I.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:

I.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 darüberliegende 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..DT`-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.

I.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.

I.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 Prográmmchen 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.

I.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.

I.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 I.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 I.1: Sonderzeichenschreibweise des *rescomp*

I.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...

I.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.

I.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 Großbuchtaben 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 Assemblerlisting 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 Integerkonstanten 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.

- `SetIsOccupied` : Einige Prozessoren verwenden `SET` als Maschinenbefehl. Ist diese Variable gesetzt, so gibt AS `SET` -Befehle an den Codegenerator weiter und verwendet stattdessen `EVAL`.

- `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 zurückgeliefert 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!

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 den Funktionszeiger `InitPassProc` angebotene Haken bietet die Möglichkeit, derartige Initialisierungen vorzunehmen. Das verwendete Prinzip ähnelt dabei dem Einhängen in einen Interruptvektor: In der Initialisierung der Unit wird der alte Wert von `InitPassProc` gesichert. Danach kann `InitPassProc` auf die hinzuzufügende Funktion (parameterlos, kein Rückgabewert) umgebogen werden. Die neue Routine ruft dann zuerst die alte Initialisierungsroutine auf und führt danach ihre eigenen Operationen durch.

Analog zu `InitPassProc` funktioniert die über `CleanUpProc` aufgebaute Funktionskette, 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...

I.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 solle 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] Atmel Corp.:
AVR Enhanced RISC Microcontroller Data Book.
May 1996

[4] Atmel Corp.:
8-Bit AVR Assembler and Simulator Object File Formats (Preliminary).
(Teil der AVR-Tools-Dokumentation)

[5] CMD Microcircuits:
G65SC802 / G65SC816 CMOS 8/16-Bit Microprocessor Family Data Sheet.

[6] National Semiconductor:
COP410L/COP411L/COP310L/COP311L Single-Chip N-Channel Microcontrollers. RRD-B30M105, March 1992

[7] National Semiconductor:
COPS Family User's Guide.

[8] Digital Research :
CP/M 68K Operating System User's Guide.
1983

[9] Cyrix Corp. :
FasMath 83D87 User's Manual.
1990

[10] Dallas Semiconductor:
DS80C320 High-Speed Micro User's Guide.
Version 1.30, 1/94

[11] Fairchild Semiconductor:
ACE1101 Data Sheet.
Preliminary, May 1999

[12] Fairchild Semiconductor:
ACE1202 Data Sheet.
Preliminary, May 1999

[13] Fairchild Semiconductor:
ACEx Guide to Developer Tools. AN-8004, Version 1.3 September 1998

[14] Freescale Semiconductor:
S12XCPUV1 Reference Manual. S12XCPUV1, v01.01, 03/2005

[15] Freescale Semiconductor:
RS08 Core Reference Manual. RS08RM, Rev. 1.0, 04/2006

[16] Freescale Semiconductor:
MC9S12XDP512 Data Sheet. MC9S12XDP512, Rev. 2.11, 5/2005

[17] Fujitsu Limited:
June 1998 Semiconductor Data Book.
CD00-00981-1E

[18] Fujitsu Semiconductor:
F^2MC16LX 16-Bit Microcontroller MB90500 Series Programming Manual.
CM44-00201-1E, 1998

[19] Hitachi Ltd. :
8-/16-Bit Microprocessor Data Book.
1986

[20] Trevor J.Terrel & Robert J. Simpson :
Understanding HD6301X/03X CMOS Microprocessor Systems.
erschienen bei Hitachi

[21] Hitachi Microcomputer:
H8/300H Series Programming Manual.
(21-032, keine Jahresangabe)

[22] Hitachi Semiconductor Design & Development Center:
SH Microcomputer Hardware Manual (Preliminary).

[23] Hitachi Semiconductor and IC Div.:
SH7700 Series Programming Manual.
1st Edition, September 1995

[24] Hitachi Semiconductor and IC Div.:
H8/500 Series Programming Manual.
(21-20, 1st Edition Feb. 1989)

[25] Hitachi Ltd.:
H8/532 Hardware Manual.
(21-30, keine Jahresangabe)

[26] Hitachi Ltd.:
H8/534, H8/536 Hardware Manual.
(21-19A, keine Jahresangabe)

[27] IBM Corp.:
PPC403GA Embedded Controller User's Manual.
First Edition, September 1994

[28] Intel Corp. : *Embedded Controller Handbook.*
1987

[29] Intel Corp. :
Microprocessor and Peripheral Handbook.
Volume I Microprocessor, 1988

[30] Intel Corp. :
80960SA/SB Reference Manual.
1991

[31] Intel Corp.:
8XC196NT Microcontroller User's Manual.
June 1995

[32] Intel Corp.:
8XC251SB High Performance CHMOS Single-Chip Microcontroller.
Sept. 1995, Order Number 272616-003

[33] Intel Corp.:
80296SA Microcontroller User's Manual.
Sept. 1996

[34] Intel Corp.:
4040: Single-Chip 4-Bit P-Channel Microprocessor.
(keine Jahresangabe)

[35] Intersil:
CDP1802A, CDP1802AC, CDP1802BC CMOS 8-Bit Microprocessors.
March 1997

[36] Intersil:
CDP1805AC, CDP1806AC CMOS 8-Bit Microprocessor with On-Chip RAM and Counter/Timer.
March 1997

[37] Hirotsugu Kakugawa:
A memo on the secret features of 6309.
(erhältlich über WWW:
http://www.cs.umd.edu/users/fms/comp/CPUs/6309.txt)

[38] Lattice Semiconductor Corporation:
LatticeMico8 Microcontroller Users Guide.
Reference Design RD1026, February 2008

[39] Microchip Technology Inc.:
Microchip Data Book.
1993 Edition

[40] Mitsubishi Electric:
Single-Chip 8-Bit Microcomputers
Vol.2, 1987

[41] Mitsubishi Electric:
Single-Chip 16-Bit Microcomputers.
Enlarged edition, 1991

[42] Mitsubishi Electric:
Single-Chip 8 Bit Microcomputers.
Vol.2, 1992

[43] Mitsubishi Electric:
M34550Mx-XXXFP Users's Manual.
Jan. 1994

[44] Mitsubishi Electric:
M16 Family Software Manual.
First Edition, Sept. 1994

[45] Mitsubishi Electric:
M16C Software Manual.
First Edition, Rev. C, 1996

[46] Mitsubishi Electric:
M30600-XXXFP Data Sheet.
First Edition, April 1996

[47] Dokumentation zum M16/M32-Entwicklungspaket von Green Hills Software

[48] Motorola Inc. :
Microprocessor, Microcontroller and Peripheral Data.
Vol. I+II, 1988

[49] Motorola Inc. :
MC68881/882 Floating Point Coprocessor User's Manual.
Prentice-Hall, Englewood Cliffs, Second Edition 1989

[50] Motorola Inc. :
MC68851 Paged Memory Management Unit User's Manual.
Prentice-Hall, Englewood Cliffs, Second Edition 1989

[51] Motorola Inc.:
CPU32 Reference Manual.
Rev. 1, 1990

[52] Motorola Inc.:
DSP56000/DSP56001 Digital Signal Processor User's Manual.
Rev. 2, 1990

[53] Motorola Inc.:
MC68340 Technical Summary.
Rev. 2, 1991

[54] Motorola Inc.:
CPU16 Reference Manual.
Rev. 1, 1991

[55] Motorola Inc.:
Motorola M68000 Family Programmer's Reference Manual.
1992

[56] Motorola Inc.:
MC68332 Technical Summary.
Rev. 2, 1993

[57] Motorola Inc.:
PowerPC 601 RISC Microprocessor User's Manual.
1993

[58] Motorola Inc.:
PowerPC(tm) MPC505 RISC Microcontroller Technical Summary.
1994

[59] Motorola Inc.:
PowerPC(tm) MPC821 Portable Microprocessor User's Manual.
1996

[60] Motorola Inc.:
CPU12 Reference Manual.
1st. edition, 1996

[61] Motorola Inc.:
CPU08 Reference Manual.
Rev. 1 (keine Jahresangabe im PDF-File)

[62] Motorola Inc.:
MC68360 User's Manual.

[63] Motorola Inc.:
MCF 5200 ColdFire Family Programmer's Reference Manual.
1995

[64] Motorola Inc.:
*M*Core Programmer's Reference Manual.*
1997

[65] Motorola Inc.:
DSP56300 24-Bit Digital Signal Processor Family Manual.
Rev. 0 (keine Jahresangabe im PDF-File)

[66] Motorola Inc.:
MC68HC11K4 Technical Data. 1992

[67] National Semiconductor:
SC/MP Programmier- und Assembler-Handbuch.
Publication Number 4200094A, Aug. 1976

[68] National Semiconductor:
COP800 Assembler/Linker/Librarian User's Manual.

Customer Order Number COP8-ASMLNK-MAN, NSC Publication Number 424421632-001B, August 1993

[69] National Semiconductor:
COP87L84BC microCMOS One-Time-Programmable (OTP) Microcontroller.
Preliminary, March 1996

[70] National Semiconductor:
SC14xxx DIP commands Reference guide.
Application Note AN-D-031, Version 0.4, 28.12.1998

[71] National Semiconductor:
INS8070-Series Microprocessor Family. October 1980

[72] NEC Corp.:
µpD70108 / µpD70116 / µpD70208 / µpD70216 / µpD72091 Data Book.
(keine Jahresangabe)

[73] NEC Electronics Europe GmbH:
User's Manual µCOM-87 AD Family.
(keine Jahresangabe)

[74] NEC Corp.:
µCOM-75x Family 4-bit CMOS Microcomputer User's Manual.
Vol. I+II (keine Jahresangabe)

[75] NEC Corp.:
Digital Signal Processor Product Description.
PDDSP.....067V20 (keine Jahresangabe)

[76] NEC Corp.:
µPD78070A, 78070AY 8-Bit Single-Chip Microcontroller User's Manual.
Document No. U10200EJ1V0UM00 (1st edition), August 1995

[77] NEC Corp.:
Data Sheet µPD78014.

[78] NEC Corp.:
78K/II Series 8-Bit Single-Chip Microcontroller User's Manual - Instructions.
Document No. U10228EJ6V0UM00 (6th edition), December 1995

[79] Philips Semiconductor:
16-bit 80C51XA Microcontrollers (eXtended Architecture).
Data Handbook IC25, 1996

[80] SGS-Thomson Microelectronics:
8 Bit MCU Families EF6801/04/05 Databook.
1st edition, 1989

[81] SGS-Thomson Microelectronics:
ST6210/ST6215/ST6220/ST6225 Databook.
1st edition, 1991

[82] SGS-Thomson Microelectronics:
ST7 Family Programming Manual.
June 1995

[83] SGS-Thomson Microelectronics:
ST9 Programming Manual.
3rd edition, 1993

[84] Siemens AG:
SAB80C166/83C166 User's Manual.
Edition 6.90

[85] Siemens AG:
SAB C167 Preliminary User's Manual.
Revision 1.0, July 1992

[86] Siemens AG:
SAB-C502 8-Bit Single-Chip Microcontroller User's Manual.
Edition 8.94

[87] Siemens AG:
SAB-C501 8-Bit Single-Chip Microcontroller User's Manual.
Edition 2.96

[88] Siemens AG:
C504 8-Bit CMOS Microcontroller User's Manual.
Edition 5.96

[89] C.Vieillefond:
Programmierung des 68000.
Sybex-Verlag Düsseldorf, 1985

[90] Symbios Logic Inc:
Symbios Logic PCI-SCSI-I/O Processors Programming Guide.
Version 2.0, 1995/96

[91] Texas Instruments:
Model 990 Computer/TMS9900 Microprocessor Assembly Language Programmer's Guide.
1977, Manual No. 943441-9701

[92] Texas Instruments:
TMS9995 16-Bit Microcomputer Preliminary Data Manual.
1981

[93] Texas Instruments:
First-Generation TMS320 User's Guide.
1988, ISBN 2-86886-024-9

[94] Texas Instruments:
TMS7000 family Data Manual.
1991, DB103

[95] Texas Instruments:
TMS320C3x User's Guide.
Revision E, 1991

[96] Texas Instruments:
TMS320C2x User's Guide.
Revision C, Jan. 1993

[97] Texas Instruments:
TMS370 Family Data Manual.
1994, SPNS014B

[98] Texas Instruments:
MSP430 Family Software User's Guide.
1994, SLAUE11

[99] Texas Instruments:
MSP430 Metering Application.
1996, SLAAE10A

[100] Texas Instruments:
MSP430 Family Architecture User's Guide.
1995, SLAUE10A

[101] Texas Instruments:
TMS320C62xx CPU and Instruction Set Reference Manual.
Jan. 1997, SPRU189A

[102] Texas Instruments:
TMS320C20x User's Guide.
April 1999, SPRU127C

[103] Texas Instruments:
TMS320C54x DSP Reference Set; Volume 1: CPU and Peripherals.
March 2001, SPRU172C

[104] Texas Instruments:
TMS320C54x DSP; Volume 2: Mnemonic Instruction Set.
March 2001, SPRU172C

[105] Toshiba Corp.:
8-Bit Microcontroller TLCS-90 Development System Manual.
1990

[106] Toshiba Corp.:
8-Bit Microcontroller TLCS-870 Series Data Book.
1992

[107] Toshiba Corp.:
16-Bit Microcontroller TLCS-900 Series Users Manual.
1992

[108] Toshiba Corp.:
16-Bit Microcontroller TLCS-900 Series Data Book: TMP93CM40F/TMP93CM41F.
1993

[109] Toshiba Corp.:
4-Bit Microcontroller TLCS-47E/47/470/470A Development System Manual.
1993

[110] Toshiba Corp.:
TLCS-9000/16 Instruction Set Manual Version 2.2.
10. Feb 1994

[111] Valvo GmbH:
Bipolare Mikroprozessoren und bipolare LSI-Schaltungen.
Datenbuch, 1985, ISBN 3-87095-186-9

[112] Ken Chapman (Xilinx Inc.):
PicoBlaze 8-Bit Microcontroller for Virtex-E and Spartan-II/IIE Devices.
Application Note XAPP213, Version 2.1, February 2003

[113] 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

[114] David May::
The XMOS XS1 Architecture.
Publication Date: 2009/10/19, Copyright 2009 XMOS Ltd.

[115] Datenblätter der Firma Zilog zur Z80-Familie

[116] Zilog Inc.:
Z8 Microcontrollers Databook.
1992

[117] Zilog Inc.:
Discrete Z8 Microcontrollers Databook.
(keine Jahresangabe)

[118] Zilog Inc.:
Z380 CPU Central Processing Unit User's Manual.
(keine Jahresangabe)

[119] Zilog Inc.:
eZ8 CPU User Manual.
UM01285-0503

"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

ADDR, 106
ADDRW, 106
ADR, 108
ALIGN, 113
ASCII, 111
ASCIZ, 111
ASSUME, 95

BFLOAT, 109
BIGENDIAN, 91
BINCLUDE, 144
BIT, 64
BLOCK, 112
BRANCHEXT, 103
BSS, 112
BYT, 107
BYTE, 106, 107

CASE, 129
CHARSET, 66
CODEPAGE, 67
CONSTANT, 61
CPU, 74

DATA, 110
DB, 105
DBIT, 65
DC, 104
DC8, 108
DD, 105
DEFB, 106
DEFW, 106
DEPHASE, 93
DFS, 112
DOTTEDSTRUCTS, 125
DOUBLE, 109
DQ, 105
DS, 105, 107
DS16, 113
DS8, 107
DSB, 113
DSW, 113
DT, 105
DUP, 106
DW, 105
DW16, 108

EFLOAT, 109
ELSE, 128
ELSECASE, 129
ELSEIF, 128
EMULATED, 102
END, 147
ENDCASE, 129
ENDIF, 128
ENDM, 114
ENDS, 125
ENDSTRUCT, 125
ENUM, 68
EQU, 61
ERROR, 144
EXITM, 122
EXTENDED, 109
EXTMODE, 90

FATAL, 144
FB, 111
FCB, 107

FCC, 111
FDB, 108
FLOAT, 109
FPU, 88
FULLPMMU, 88
FUNCTION, 123
FW, 111

IF, 128
IFB, 129
IFDEF, 129
IFEXIST, 129
IFNB, 129
IFNDEF, 129
IFNEXIST, 129
IFNUSED, 129
IFUSED, 129
INCLUDE, 143
IRP, 120
IRPC, 120

LABEL, 63
LISTING, 133
LIV, 66
LONG, 108
LQxx, 110
LTORG, 113
LWORDMODE, 90

MACEXP, 132
MACRO, 114
MAXMODE, 90
MAXNEST, 123
MESSAGE, 144

NAMEREG, 65
NEWPAGE, 132

ORG, 69
OUTRADIX, 135

PACKING, 89
PADDING, 89
PAGE, 131
PHASE, 93
PMMU, 88
POPV, 68
PORT, 65
PRTEXIT, 133
PRTINIT, 133
PUSHV, 68

Qxx, 110

RADIX, 135
READ, 145
REG, 65
Registersymbole, 56
RELAXED, 146
REPT, 121
RES, 112
RESTORE, 94
RIV, 66
RMB, 112
RORG, 74
RSTRING, 111

SAVE, 94
SEGMENT, 92
SET, 61
SFR, 62
SFRB, 62
SHARED, 28, 57, 142
SHIFT, 122
SINGLE, 109
SPACE, 112
SRCMODE, 90
STRING, 111
STRUC, 125
STRUCT, 125
SUPMODE, 88
SWITCH, 129

TFLOAT, 109
TITLE, 134

UNION, 125

WARNING, 144
WHILE, 121
WORD, 106, 108
WRAPMODE, 91

XSFR, 63

YSFR, 63

ZERO, 110