Alfred Arnold, Stefan Hilse, Stephan Kanthak, Oliver Sellke,
Vittorio De Tomasi

Macro Assembler AS V1.42

User's Manual

Edition Oktober 2011

IBM, PPC403Gx, OS/2, and PowerPC are registered trademarks of IBM Corporation.

Intel, MCS-48, MCS-51, MCS-251, MCS-96, MCS-196 und MCS-296 are registered trademarks of Intel Corp. .

Motorola and ColdFire are registered trademarks of Motorola Inc. .

PicoBlaze is a registered trademark of Xilinx Inc.

UNIX is a registered trademark of the The Open Group.

Linux is a registered trademark of Linus Thorvalds.

Microsoft, Windows, and MS-DOS are registered trademarks of Microsoft Corporation.

All other trademarks not explicitly mentioned in this section and used in this manual are properties of their respective owners.

This document has been processed with the LaTeX typesetting system, using the Linux operating system.

Contents

1 Introduction **11**

1.1 License Agreement 11
1.2 General Capabilities of the Assembler 13
1.3 Supported Platforms 19

2 Assembler Usage **21**

2.1 Hardware Requirements 21
2.2 Delivery 22
2.3 Installation 23
2.4 Start-Up Command, Parameters 28
2.5 Format of the Input Files 37
2.6 Format of the Listing 39
2.7 Symbol Conventions 42
2.8 Temporary Symbols 44
2.9 Named Temporary Symbols 45
2.9.1 Nameless Temporary Symbols 46
2.9.2 Composed Temporary Symbols 47
2.10 Formula Expressions 48
2.10.1 Integer Constants 48
2.10.2 Floating Point Constants 50
2.10.3 String Constants 50
2.10.4 Evaluation 52
2.10.5 Operators 52
2.10.6 Functions 54
2.11 Forward References and Other Disasters 57
2.12 Register Symbols 61
2.13 Sharefile 62
2.14 Processor Aliases 62

3 Pseudo Instructions **65**
3.1 Definitions . . . 65
3.1.1 SET, EQU, and CONSTANT . . . 65
3.1.2 SFR and SFRB . . . 67
3.1.3 XSFR and YSFR . . . 67
3.1.4 LABEL . . . 68
3.1.5 BIT . . . 68
3.1.6 DBIT . . . 69
3.1.7 PORT . . . 70
3.1.8 REG and NAMEREG . . . 70
3.1.9 LIV and RIV . . . 70
3.1.10 CHARSET . . . 71
3.1.11 CODEPAGE . . . 72
3.1.12 ENUM . . . 73
3.1.13 PUSHV and POPV . . . 73
3.2 Code Modification . . . 74
3.2.1 ORG . . . 74
3.2.2 CPU . . . 79
3.2.3 SUPMODE, FPU, PMMU . . . 92
3.2.4 FULLPMMU . . . 93
3.2.5 PADDING . . . 93
3.2.6 PACKING . . . 94
3.2.7 MAXMODE . . . 94
3.2.8 EXTMODE and LWORDMODE . . . 95
3.2.9 SRCMODE . . . 95
3.2.10 BIGENDIAN . . . 96
3.2.11 WRAPMODE . . . 96
3.2.12 SEGMENT . . . 97
3.2.13 PHASE and DEPHASE . . . 98
3.2.14 SAVE and RESTORE . . . 99
3.2.15 ASSUME . . . 100
3.2.16 EMULATED . . . 107
3.2.17 BRANCHEXT . . . 108
3.3 Data Definitions . . . 109
3.3.1 DC[.Size] . . . 109
3.3.2 DS[.Size] . . . 110
3.3.3 DB,DW,DD,DQ, and DT . . . 111
3.3.4 DS, DS8 . . . 112

3.3.5 BYT or FCB . . . 113
3.3.6 BYTE . . . 113
3.3.7 DC8 . . . 113
3.3.8 ADR or FDB . . . 113
3.3.9 WORD . . . 114
3.3.10 DW16 . . . 114
3.3.11 LONG . . . 114
3.3.12 SINGLE, DOUBLE, and EXTENDED . . . 114
3.3.13 FLOAT and DOUBLE . . . 115
3.3.14 EFLOAT, BFLOAT, and TFLOAT . . . 115
3.3.15 Qxx and LQxx . . . 115
3.3.16 DATA . . . 116
3.3.17 ZERO . . . 116
3.3.18 FB and FW . . . 116
3.3.19 ASCII and ASCIZ . . . 117
3.3.20 STRING and RSTRING . . . 117
3.3.21 FCC . . . 117
3.3.22 DFS or RMB . . . 117
3.3.23 BLOCK . . . 118
3.3.24 SPACE . . . 118
3.3.25 RES . . . 118
3.3.26 BSS . . . 118
3.3.27 DSB and DSW . . . 118
3.3.28 DS16 . . . 119
3.3.29 ALIGN . . . 119
3.3.30 LTORG . . . 119
3.4 Macro Instructions . . . 120
3.4.1 MACRO . . . 120
3.4.2 IRP . . . 126
3.4.3 IRPC . . . 126
3.4.4 REPT . . . 127
3.4.5 WHILE . . . 127
3.4.6 EXITM . . . 128
3.4.7 SHIFT . . . 128
3.4.8 MAXNEST . . . 129
3.4.9 FUNCTION . . . 129
3.5 Structures . . . 131
3.5.1 Definition . . . 131

3.5.2 Usage . . . 132
3.5.3 Nested Structures . . . 133
3.5.4 Unions . . . 133
3.5.5 Structures and Sections . . . 133
3.6 Conditional Assembly . . . 134
3.6.1 IF / ELSEIF / ENDIF . . . 134
3.6.2 SWITCH / CASE / ELSECASE / ENDCASE . . . 136
3.7 Listing Control . . . 137
3.7.1 PAGE . . . 137
3.7.2 NEWPAGE . . . 138
3.7.3 MACEXP . . . 139
3.7.4 LISTING . . . 139
3.7.5 PRTINIT and PRTEXIT . . . 140
3.7.6 TITLE . . . 141
3.7.7 RADIX . . . 141
3.7.8 OUTRADIX . . . 141
3.8 Local Symbols . . . 142
3.8.1 Basic Definition (SECTION/ENDSECTION) . . . 143
3.8.2 Nesting and Scope Rules . . . 144
3.8.3 PUBLIC and GLOBAL . . . 147
3.8.4 FORWARD . . . 148
3.8.5 Performance Aspects . . . 149
3.9 Miscellaneous . . . 150
3.9.1 SHARED . . . 150
3.9.2 INCLUDE . . . 150
3.9.3 BINCLUDE . . . 151
3.9.4 MESSAGE, WARNING, ERROR, and FATAL . . . 152
3.9.5 READ . . . 153
3.9.6 RELAXED . . . 154
3.9.7 END . . . 154

4 Processor-specific Hints **157**
4.1 6811 . . . 157
4.2 PowerPC . . . 159
4.3 DSP56xxx . . . 159
4.4 H8/300 . . . 160
4.5 SH7000/7600/7700 . . . 160
4.6 MELPS-4500 . . . 164
4.7 6502UNDOC . . . 164
4.8 MELPS-740 . . . 168
4.9 MELPS-7700/65816 . . . 168
4.10 M16 . . . 172
4.11 4004/4040 . . . 173
4.12 MCS-48 . . . 173
4.13 MCS-51 . . . 173
4.14 MCS-251 . . . 174
4.15 8085UNDOC . . . 177
4.16 8086..V35 . . . 179
4.17 8X30x . . . 182
4.18 XA . . . 183
4.19 AVR . . . 184
4.20 Z80UNDOC . . . 184
4.21 Z380 . . . 185
4.22 Z8 and eZ8 . . . 186
4.23 TLCS-900(L) . . . 188
4.24 TLCS-90 . . . 192
4.25 TLCS-870 . . . 192
4.26 TLCS-47 . . . 193
4.27 TLCS-9000 . . . 193
4.28 29xxx . . . 194
4.29 80C16x . . . 194
4.30 PIC16C5x/16C8x . . . 196
4.31 PIC 17C4x . . . 197
4.32 ST6 . . . 198
4.33 ST7 . . . 199
4.34 ST9 . . . 199
4.35 6804 . . . 200
4.36 TMS3201x . . . 201
4.37 TMS320C2x . . . 201

4.38 TMS320C3x . . . 202
4.39 TMS9900 . . . 202
4.40 TMS70Cxx . . . 203
4.41 TMS370xxx . . . 204
4.42 MSP430 . . . 205
4.43 COP8 & SC/MP . . . 205
4.44 SC144xxx . . . 205
4.45 75K0 . . . 206
4.46 78K0 . . . 208
4.47 78K2 . . . 208
4.48 uPD772x . . . 208
4.49 F2MC16L . . . 209

5 File Formats 211
5.1 Code Files . . . 211
5.2 Debug Files . . . 214

6 Utility Programs 219
6.1 PLIST . . . 220
6.2 BIND . . . 221
6.3 P2HEX . . . 222
6.4 P2BIN . . . 226
6.5 AS2MSG . . . 228

A Error Messages of AS 229

B I/O Error Messages 271

C Frequently Asked Questions 275

D Pseudo-Instructions Collected 279

E Predefined Symbols 293

F Shipped Include Files 297
F.1 BITFUNCS.INC . . . 297
F.2 CTYPE.INC . . . 298

G Acknowledgments 301

H Changes since Version 1.3 **303**

I Hints for the AS Source Code **319**
I.1 Language Preliminaries 319
I.2 Capsuling System dependencies 320
I.3 System-Independent Files 321
I.3.1 Modules Used by AS 321
I.3.2 Additional Modules for the Tools 326
I.4 Modules Needed During the Build of AS 327
I.5 Generation of Message Files 329
I.5.1 Format of the Source Files 329
I.6 Creation of Documentation 331
I.7 Test Suite 333
I.8 Adding a New Target Processor 333
I.9 Localization to a New Language 340

Chapter 1

Introduction

This instruction is meant for those people who are already very familiar with Assembler and who like to know how to work with AS. It is rather a reference than a user's manual and so it neither tries to explain the "language assembler" nor the processors. I have listed further literature in the bibliography which was substantial in the implementation of the different code generators. There is no book I know where you can learn Assembler from the start, so I generally learned this by "trial and error".

1.1 License Agreement

Before we can go "in medias res", first of all the inevitable prologue:

As in the present version is licensed according to the Gnu General Public License (GPL); the details of the GPL may be read in the file COPYING bundled with this distribution. If you did not get it with AS, complain to the one you got AS from!

Shortly said, the GPL covers the following points:

- Programs based upon AS must also be licensed according to the GPL;
- distribution is explicitly allowed;

- explicit disclaiming of all warranties for damages resulting from usage of this program.

...however, I really urge you to read the file COPYING for the details!

To accelerate the error diagnose and correction, please add the following details to the bug report:

- hardware:
  - processor type (with/without coprocessor)
  - amount of memory installed
  - video card
  - hard-disk type(s) and their interface(s)
- software:
  - operating system (MS-DOS, Novell-DOS, DR-DOS, OS/2, Windows) and version
  - resident (TSR) programs installed
  - version of AS including dates of the `EXE`-files
- if possible, the source file, in which the bug occurs

You can contact me as follows:

- by Surface Mail:

  Alfred Arnold

  Hirschgraben 29

  D-52062 Aachen

  Germany

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

If someone likes to meet me personally to ask questions and lives near Aachen (= Aix-la-Chapelle), you will be able to meet me there. You can do this most probably on thursdays from 8pm to 9pm at the computerclub inside the RWTH Aachen (Eilfschornsteinstrasse 16, cellar of philosophers' building, backdoor entry).

Please don't call me by phone. First, complex relations are extremely hard to discuss at phone. Secondly, the telephone companies are already rich enough...

The latest version of AS (DOS, DPMI, OS/2, C) is available from the following Server:

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

or shortly

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

The sources of the C version may also be fetched from the following server:

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

..and of course thereby from every Sunsite mirror in the world!

Whoever has no access to an FTP-Server can ask me to send the assembler by mail. Only requests containing a blank CD-R and a self-addressed, (correctly) stamped envelope will be answered. Don't send any money!

Now, after this inevitable introduction we can turn to the actual documentation:

1.2 General Capabilities of the Assembler

In contrast to ordinary assemblers, AS offers the possibility to generate code for totally different processors. At the moment, the following processor families have been implemented:

- Motorola 68000..68040,683xx incl. coprocessor and MMU

- Motorola ColdFire
- Motorola DSP5600x,DSP56300
- Motorola M-Core
- Motorola/IBM MPC601/MPC505/PPC403
- Motorola 6800, 68(HC)11(K4) and Hitachi 6301
- Motorola/Freescale 6805, 68HC(S)08
- Motorola 6809 / Hitachi 6309
- Motorola/Freescale 68HC12(X) including XGATE
- Motorola 68HC16
- Freescale 68RS08
- Hitachi H8/300(H)
- Hitachi H8/500
- Hitachi SH7000/7600/7700
- Rockwell 6502 and 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 μPD 78(C)1x
- NEC μPD 75xxx (alias 75K0)
- NEC 78K0
- NEC 78K2
- NEC μPD7720/7725

- NEC μPD77230
- Symbios Logic SYM53C8xx (yes, they are programmable!)
- Fujitsu F^{2}MC8L
- Fujitsu F^{2}MC16L
- Intersil CDP1802/1805

under work / planned / in consideration :

- NEC 78K4
- Intel 8008
- Analog Devices ADSP21xx
- SGS-Thomson ST20
- Texas Instruments TMS320C4x
- Texas Instruments TMS320C8x
- Toshiba TC9331

I'm currently searching for documentation about the following families:

- the complete set of OKI controllers

unloved, but now, however, present :

- Intel 80x86, 80186, Nec V30&V35 incl. coprocessor 8087

The switch to a different code generator is allowed even within one file, and as often as one wants!

The reason for this flexibility is that AS has a history, which may also be recognized by looking at the version number. AS was created as an extension of a macro assembler for the 68000 family. On special request, I extended the

original assembler so that it was able to translate 8051 mnemonics. On this way (decline ?!) from the 68000 to 8051, some other processors were created as by-products. All others were added over time due to user requests. So At least for the processor-independent core of AS, one may assume that it is well-tested and free of obvious bugs. However, I often do not have the chance to test a new code generator in practice (due to lack of appropriate hardware), so surprises are not impossible when working with new features. You see, the things stated in section 1.1 have a reason...

This flexibility implies a somewhat exotic code format, therefore I added some tools to work with it. Their description can be found in chapter 6.

AS is a macro assembler, which means that the programmer has the possibility to define new "commands" by means of macros. Additionally it masters conditional assembling. Labels inside macros are automatically processed as being local.

For the assembler, symbols may have either integer, string or floating point values. These will be stored - like interim values in formulas - with a width of 32 bits for integer values, 80 or 64 bits for floating point values, and 255 characters for strings. For a couple of micro controllers, there is the possibility to classify symbols by segmentation. So the assembler has a (limited) possibility to recognize accesses to wrong address spaces.

The assembler does not know explicit limits in the nesting depth of include files or macros; a limit is only given by the program stack restricting the recursion depth. Nor is there a limit for the symbol length, which is only restricted by the maximum line length.

From version 1.38 on, AS is a multipass-assembler. This pompous term means no more than the fact that the number of passes through the source code need not be exactly two. If the source code does not contain any forward references, AS needs only one pass. In case AS recognizes in the second pass that it must use a shorter or longer instruction coding, it needs a third (fourth, fifth...) pass to process all symbol references correctly. There is nothing more behind the term "multipass", so it will not be used further more in this documentation.

After so much praise a bitter pill: AS cannot generate linkable code. An extension with a linker needs considerable effort and is not planned at the moment.

Those who want to take a look at the sources of AS can simply get the Unix version of AS, which comes as source for self-compiling. The sources are definitely not in a format that is targeted at easy understanding - the original Pascal version still raises its head at a couple of places, and I do not share a couple of common opinions about 'good' C coding.

1.3 Supported Platforms

DOS

Though AS started as a pure DOS program, there are a couple of versions available that are able to exploit a bit more than the Real Mode of an Intel CPU. Their usage is kept as compatible to the DOS version as possible, but there are of course differences concerning installation and embedding into the operating system in question. Sections in this manual that are only valid for a specific version of AS are marked with a corresponding sidemark (at this paragraph for the DOS version) aheaded to the paragraph. In detail, the following further versions exist (distributed as separate packages):

DPMI

In case you run into memory problems when assembling large and complex programs under DOS, there is a DOS version that runs in protected mode via a DOS extender and can therefore make use of the whole extended memory of an AT. The assembly becomes significantly slower by the extender, but at least it works...

OS/2

There is a native OS/2 version of AS for friends of IBM's OS/2 operating system. Since version 1.41r8, this is a full 32-bit OS/2 application, which of course means that OS/2 2.x and at least an 80386 CPU are mandatory.

UNIX

You can leave the area of PCs-only with the C version of AS that was designed to be compilable on a large number of UNIX systems (this includes OS/2 with the emx compiler) without too much of tweaking. In contrast to the previously mentioned versions, the C version is delivered in source code, i.e. one has to create the binaries by oneself using a C compiler. This is by far the simpler way (for me) than providing a dozen of precompiled binaries for machines I sometimes only have limited access to...

???

People who have read this enumeration up to this point will notice that world's best-selling operating system coming from Redmont is missing in this enumeration. People who know me personally will know that I do not

regard Windows to be a pat solution (regardless if its 3.X, 95, or NT). Frankly said, I am a 'windows hater'. A large number of people will now regard this to be somewhere between obsolete and ridiculous, and they will tell me that I withhold AS from a large part of potential users, but they will have to live with it: I primarily continue to improve AS because I have fun doing it; AS is a non-commercial project and I therefore take the freedom not to look at potential market shares. I select platforms for me where I have fun programming, and I definitely do not have any fun when programming for Windows! By the way, there was a time when I had to write Windows programs so I do not simply jabber without having an idea what I am talking about. If someone wants to port AS into this direction, I will not stand in his way, but (s)he should not expect anything more from me than providing sources (which is why (s)he will have to deal with questions like 'why does AS not work any more after I changed the JUNK-CAD 18.53 registry entry from upper to lower case?').

Chapter 2

Assembler Usage

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 Requirements

The hardware requirements of AS vary substantially from version to version:

DOS

The DOS version will principally run on any IBM-compatible PC, ranging from a PC/XT with 4-dot-little megahertz up to a Pentium. However, similar to other programs, the fun using AS increases the better your hardware is. An XT user without a hard drive will probably have significant trouble placing the overlay file on a floppy because it is larger than 500 Kbytes...the PC should therefore have at least a hard drive, allowing acceptable loading times. AS is not very advanced in its main memory needs: the program itself allocates less than 300 Kbytes main memory, AS should therefore work on machines with at least 512 Kbytes of memory.

DPMI

The version of AS compiled for the DOS Protected Mode Interface (DPMI)

requires at least 1 Mbyte of free extended memory. A total memory capacity of at least 2 Mbytes is therefore the absolute minimum given one does not have other tools in the XMS (like disk caches, RAM disks, or a hi-loaded DOS); the needs will rise then appropriately. If one uses the DPMI version in a DOS box of OS/2, one has to assure that DPMI has been enabled via the box's DOS settings (set to `on` or `auto`) and that a sufficient amount of XMS memory has been assigned to the box. The virtual memory management of OS/2 will free you from thinking about the amount of free real memory.

OS/2

The hardware requirements of the OS/2 version mainly result from the needs of the underlying operating system, i.e. at minimum an 80386SX processor, 8 Mbytes of RAM (resp. 4 Mbytes without the graphical user interface) and 100..150 Mbytes of hard disk space. AS2 is only a 16-bit application and therefore it should also work on older OS/2 versions (thereby reducing the processor needs to at least an 80286 processor); I had however no chance to test this.

UNIX

The C version of AS is delivered as source code and therefore requires a UNIX or OS/2 system equipped with a C compiler. The compiler has to fulfill the ANSI standard (GNU-C for example is ANSI-compliant). You can look up in the `README` file whether your UNIX system has already been tested so that the necessary definitions have been made. You should reserve about 15 Mbytes of free hard disk space for compilation; this value (and the amount needed after compilation to store the compiled programs) strongly differs from system to system, so you should take this value only as a rough approximation.

2.2 Delivery

Principally, you can obtain AS in one of two forms: as a *binary* or a *source* distribution. In case of a binary distribution, one gets AS, the accomanying tools and auxiliary files readily compiled, so you can immediately start to use it after unpacking the archive to the desired destination on your hard drive. Binary distibutions are made for widespread platforms, where either the majority of users does not have a compiler or the compilation is tricky (currently, this includes DOS and OS/2). A source distribution in contrast contains the complete set of C sources to generate AS; it is ultimately a

snapshot of the source tree I use for development on AS. The generation of AS from the sources and their structure is described in detail in appendix I, which is why at this place, only the contents and installation of a binary distribution will be described:

The contents of the archive is separated into several subdirectories, therefore you get a directory subtree immediately after unpacking without having to sort out things manually. The individual directories contain the following groups of files:

- `BIN`: executable programs, text resources;
- `INCLUDE`: include files for assembler programs, e.g. register definitions or standard macros;
- `MAN`: quick references for the individual programs in Unix 'man' format.

A list of the files found in every binary distribution is given in tables 2.1 to 2.3. In case a file listed in one of these (or the following) tables is missing, someone took a nap during copying (probably me)...

Depending on the platform, a binary distribution however may contain more files to allow operation, like files necessary for DOS extenders. In case of the DOS DPMI version , the extensions listed in table 2.4 result. Just to mention it: it is perfectly O.K. to replace the tools with their counterparts from a DOS binary distribution; on the on hand, they execute significantly faster without the extender's overhead, and on the other hand, they do not need the extended memory provided by the extender. *DPMI*

An OS/2 binary distribution contains in addition to the base files a set of DLLs belonging to the runtime environment of the emx compiler used to build AS (table 2.5). In case you already have these DLLs (or newer versions of them), you may delete these and use your ones insted. *OS/2*

2.3 Installation

There is no need for a special installation prior to usage of AS. It is sufficient to unpack the archive in a fitting place and to add a few minor settings. For *DOS*

File	function
Directory BIN	
AS.EXE	executable of assembler
PLIST.EXE	lists contents of code files
PBIND.EXE	merges code files
P2HEX.EXE	converts code files to hex files
P2BIN.EXE	converts code files to binary files
AS.MSG	text resources for AS
PLIST.MSG	text resources for PLIST
PBIND.MSG	text resources for PBIND
P2HEX.MSG	text resources for P2HEX
P2BIN.MSG	text resources for P2BIN
TOOLS.MSG	common text resources for all tools
CMDARG.MSG	common text resources for all programs
IOERRS.MSG	
Directory DOC	
AS_DE.DOC	german documentation, ASCII format
AS_DE.HTML	german documentation, HTML format
AS_DE.TEX	german documentation, LaTeX format
AS_EN.DOC	english documentation, ASCII format
AS_EN.HTML	english documentation, HTML format
AS_EN.TEX	english documentation, LaTeX format
Directory INCLUDE	
BITFUNCS.INC	functions for bit manipulation
CTYPE.INC	functions for classification of characters
80C50X.INC	register addresses SAB C50x
80C552.INC	register addresses 80C552
H8_3048.INC	register addresses H8/3048
REG166.INC	addresses and instruction macros 80C166/167
REG251.INC	addresses and bits 80C251
REG29K.INC	peripheral addresses AMD 2924x

Table 2.1: Standard Contents of a Binary Distribution - Part 1

File	Function
Directory INCLUDE	
REG53X.INC	register addresses H8/53x
REG683XX.INC	register addresses 68332/68340/68360
REG7000.INC	register addresses TMS70Cxx
REG78K0.INC	register addresses 78K0
REG96.INC	register addresses MCS-96
REGACE.INC	register addresses ACE
REGAVR.INC	register and bit addresses AVR family
REGCOP8.INC	register addresses COP8
REGGP32.INC	register addresses 68HC908GP32
REGHC12.INC	register addresses 68HC12
REGM16C.INC	register addresses Mitsubishi M16C
REGMSP.INC	register addresses TI MSP430
REGST9.INC	register and Makrodefinitionen ST9
REGZ380.INC	register addresses Z380
STDDEF04.INC	register addresses 6804
STDDEF16.INC	instruction macros and register addresses PIC16C5x
STDDEF17.INC	register addresses PIC17C4x
STDDEF18.INC	register addresses PIC16C8x
STDDEF2X.INC	register addresses TMS3202x
STDDEF37.INC	register and bit addresses TMS370xxx
STDDEF3X.INC	peripheral addresses TMS320C3x
STDDEF47.INC	instruction macros TLCS-47
STDDEF51.INC	definition of SFRs and bits for 8051/8052/80515
STDDEF56K.INC	register addresses DSP56000
STDDEF5X.INC	peripheral addresses TMS320C5x
STDDEF60.INC	instruction macros and register addresses PowerPC
STDDEF62.INC	register addresses and Makros ST6
STDDEF75.INC	register addresses 75K0

Table 2.2: Standard Contents of a Binary Distribution - Part 2

File	Function
Directory INCLUDE	
STDDEF87.INC	register and memory addresses TLCS-870
STDDEF90.INC	register and memory addresses TLCS-90
STDDEF96.INC	register and memory addresses TLCS-900
STDDEFXA.INC	SFR and bit addresses Philips XA
STDDEFZ8.INC	register addresses Z8 family
Directory LIB	
Directory MAN	
ASL.1	quick reference for AS
PLIST.1	quick reference for PLIST
PBIND.1	quick reference for PBIND
P2HEX.1	quick reference for P2HEX
P2BIN.1	quick reference for P2BIN

Table 2.3: Standard Contents of a Binary Distribution - Part 3

File	Function
Directory BIN	
DPMI16BI.OVL	DPMI server for the assembler
RTM.EXE	runtime module of the extender

Table 2.4: Additional Files in a DPMI Binary Distribution

File	function
Directory BIN	
EMX.DLL	runtime libraries for AS and
EMXIO.DLL	its tools
EMXLIBC.DLL	
EMXWRAP.DLL	

Table 2.5: Additional Files in an OS/2 binary distribution

example, this is an installation a user used to UNIX-like operating systems might choose:

Create a directory `c:\as` an (I will assume in the following that you are going to install AS on drive C), change to this directory and unpack the archiv, keeping the path names stored in the archive (when using PKUNZIP, the command line option `-d` is necessary for that). You now should have the following directory tree:

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

Now, append the directory `c:\as\bin` to the `PATH` statement in your `AUTOEXEC.BAT`, which allows the system to find AS and its tools. With your favourite text editor, create a file named `AS.RC` in the `lib` directory with the following contents:

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

This so-called *key file* tells AS where to search for its include files. The following statement must be added to your `AUTOEXEC.BAT` to tell AS to read this file:

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

There are many more things you can preset via the key file; they are listed in the following section.

DPMI

The installation of the DPMI version should principally take the same course as for the pure DOS version; as soon as the PATH contains the `bin` directory, the DOS extender's files will be found automatically and you should not notice anything of this mechanism (except for the longer startup time...). When working on an 80286-based computer, it is theoretically possible tha you get confronted with the following message upon the first start:

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

Since the DPMIINST tool ins not any more included in newer versions of Borland's DOS extender, I suppose that this is not an item any more...in case you run into this, contact me!

OS/2 The installation of the OS/2 version can generally be done just like for the DOS version, with the addition that the DLLs have to be made visible for the operating system. In case you do not want to extend the `LIBPATH` entry in your `CONFIG.SYS`, it is of course also valid to move the DLLs into a directory already listed in `LIBPATH`.

UNIX As already mentioned, the installation instructions in this section limit themselves to binary distributions. Since an installation under Unix is currently alway a source-based installation, the only hint I can give here is a reference to appendix I.

2.4 Start-Up Command, Parameters

AS is a command line driven program, i.e. all parameters and file options are to be given in the command line.

A couple of message files belongs to AS (recognizable by their suffix `MSG`) AS accesses to dynamically load the messages appropriate for the national language. AS searches the following directories for these files:

- the current directory;
- the EXE-file's directory;
- the directory named in the `AS_MSGPATH` environment variable, or alternitavely the directories listed in the `PATH` environment variable;
- the directory compiled into AS via the `LIBDIR` macro.

These files are *indispensable* for a proper operation of AS, i.e. AS will terminate immediately if these files are not found.

The language selection (currently only German and English) is based on the `COUNTRY` setting under DOS and OS/2 respectively on the `LANG` environment variable under Unix.

OS

In order to fulfill AS's memory requirements under DOS, the various code generator modules of the DOS version were moved into an overlay which is part of the EXE file. A separate OVR file like in earlier versions of AS therefore dose not exist any more, AS will however still attempt to reduce the overlaying delays by using eventually available EMS or XMS memory. In case this results in trouble, you may suppress usage of EMS or XMS by setting the environment variable `USEXMS` or `USEEMS` to `n`. E.g., it is possible to suppress the using of XMS by the command:

```
SET USEXMS=n
```

Since AS performs all in- and output via the operating system (and therefore it should run also on not 100% compatible DOS-PC's) and needs some basic display control, it emits ANSI control sequences during the assembly. In case you should see strange characters in the messages displayed by AS, your *DOS/* `CONFIG.SYS` is obviously lacking a line like this:

```
device=ansi.sys
```

but the further functions of AS will not be influenced hereby. Alternatively *DPMI* you are able to suppress the output of ANSI sequences completely by setting the environment variable `USEANSI` to `n`.

The DOS extender of the DPMI version can be influenced in its memory *DPMI* allocation strategies by a couple of environment variables; if you need to know their settings, you may look up them in the file `DPMIUSER.DOC`. ASX is additionally able to extend the available memory by a swap file. To do this, set up an environment variable `ASXSWAP` in the following way:

```
SET ASXSWAP=<size>[,file name]
```

The size specification has to be done in megabytes and **has** to be done. The file name in contrast is optional; if it is missing, the file is named `ASX.TMP` and placed in the current directory. In any case, the swap file is deleted after program end.

The command line parameters can roughly be divided into three categories: switches, key file references (see below) and file specifications. Parameters of these two categories may be arbitrarily mixed in the command line. The assembler evaluates at first all parameters and then assembles the specified files. From this follow two things:

- the specified switches affect all specified source files. If several source files shall be assembled with different switches, this has to be done in separate runs.

- it is possible to assemble more than one file in one shot and to bring it to the top, it is allowed that the file specs contain wildcards.

Parameter switches are recognized by AS by starting with a slash (/) or hyphen (-). There are switches that are only one character long and additionally switches composed of a whole word. Whenever AS cannot interpret a switch as a whole word, it tries to interprete every letter as an individual switch. For example, if you write

```
-queit
```

instead of

```
-quiet
```

AS will take the letters `q, u, e, i`, and `t` as individual switches. Multiple-letter switches additionally have the difference to single-letter switches that AS will accept an arbitrary mixture of upper and lower casing, whereas single-letter switches may have a different meaning depending on whether upper or lower case is used.

At the moment, the following switches are defined:

- `l`: sends assembler listing to console terminal (mostly screen). In case several passes have to be done, the listing of all passes will be send to the console (in opposite to the next option).

- `L`: writes assembler listing into a file. The list file will get the same name as the source file, only the extension is replaced by `LST`. Except one uses...

- `OLIST`: with a fiel name as argument allows to redirect the listing to a different file or a different path. This option may be used multiple times in case multiple files are assembled with one execution.

- `o`: Sets the new name of the code file generated by AS. If this option is used multiple times, the names will be assigned, one after the other, to the source files which have to be assembled. A negation (see below) of this option in connection with a name erases this name from the list. A negation without a name erases the whole list.

- `SHAREOUT`:ditto for a SHARE file eventually to be created.

- `c`: SHARED-variables will be written in a format which permits an easy integration into a C-source file. The extension of the file is `H`.

- `p`: SHARED-variables will be written in a format which permits easy integration into the CONST-block of a Pascal program. The extension of the file is `INC`.

- `a`: SHARED-variables will be written in a format which permits easy integration into an assembler source file. The extension of the file is `INC`.

Concerning effect and function of the SHARED-symbols please see chapters 2.13 resp. 3.9.1.

- `g [format]`: This switch instructs AS to create an additional file that contains debug information for the program. Allowed formats are the AS-specific MAP format (`format=MAP`), a NoICE-compatible command file (`format=NOICE`), and the Atmel format used by the AVR tools (`format=ATMEL`). The information stored in the MAP format is comprised of a symbol table and a table describing the assignment of source lines to machine addresses. A more detailed description of the MAP format can be found in section 5.2 The file's extension is `MAP`, `NOI`, resp. `OBJ`, depending on the chosen format. If no explicit format specification is done, the MAP format is chosen.

- `noicemask [value]`: By default, AS lists only symbols from the CODE segment in NoICE debug info files. With this option and an integer value interpreted as a bit mask, symbols fom other segments may be added. The assignment of segments to bit positions may be taken from table 5.3.

- `w`: suppress issue of warnings;

- `E [file]`: error messages and warnings produced by AS will be redirected to a file. Instead of a file, the 5 standard handles (STDIN..STDPRN) can also be specified as `!0` to `!4` . Default is `!2`, meaning STDERR. If the file option is left out, the name of the error file is the same as of the source file, but with the extension `LOG`.

- `q`: This switch suppresses all messages of AS, the exceptions are error messages and outputs which are are forced from the source file. The time needed for assembly is slightly reduced hereby and if you call AS from a shell there is no redirection required. The disadvantage is that you may "stay in the dark" for several minutes ... It is valid to write `quiet` instead of `q`.

- `h`: write hexadecimal numbers in lowercase instead of capital letters. This option is primarily a question of personal taste.

- `i <path list>`: issues a list of directories where the assembler shall automatically search for include files, in case it didn't find a file in the current directory. The different directories have to be separated by semicolons.

- `u`: calculate a list of areas which are occupied in the segments. This option is effective only in case a listing is produced. This option requires considerable additional memory and computing performance. In normal operation it should be switched off.

- `C`: generates a list of cross references. It lists which (global) symbols are used in files and lines. This list will also be generated only in case a listing is produced. This option occupies, too, additional memory capacity during assembly.

- `s`: issues a list of all sections (see chapter 3.8). The nesting is indicated by indentations (Pascal like).

- `t`: by means of this switch it is possible to separate single components of the standard issued assembler-listing. The assignment of bits to parts can be found in the next section, where the exact format of the assembly listing is explained.

- `D`: defines symbols. The symbols which are specified behind this option and separated by commas are written to the global symbol table

before starting the assembly. As default these symbols are written as integer numbers with the value TRUE, by means of an appended equal sign, however, you can select other values. The expression following the equals sign may include operators or internal functions, but **not** any further symbols, even if these should have been defined before in the list! Together with the commands for conditional assembly (see there) you may produce different program versions out of one source file by command line inputs. **CAUTION!** If the case-sensitive mode is used, this has to be specified in the command line *before* any symbol definitions, otherwise symbol names will be converted to upper case at this place!

- `A`: stores the list of global symbols in another, more compact form. Use this option if the assembler crashes with a stack overflow because of too long symbol tables. Sometimes this option can increase the processing speed of the assembler, but this depends on the sources.

- `x`: Sets the level of detail for error messages. The level is increased resp. decreased by one each time this option is given. While on level 0 (default) only the error message itself is printed, an extended message is added beginning at level 1 that should simplify the identification of the error's cause. Appendix A lists which error messages carry which extended messages. At level 2 (maximum), the source line containing the error is additionally printed.

- `n`: If this option is set, the error messages will be issued additionally with their error number (see appendix A). This is primarily intended for use with shells or IDE's to make the identification of errors easier by those numbers.

- `U`: This option switches AS to the case-sensitive mode, i.e. upper and lower case in the names of symbols, sections, macros, character sets, and user-defined functions will be distinguished. This is not the case by default.

- `P`: Instructs AS to write the source text processed by macro processor and conditional assembly into a file. Additional blank and pure comment lines are missing in this file. The extension of this file is `I`.

- `M`: If this switch is given, AS generates a file, that contains definitions of macros defined in the source file that did not use the `NOEXPORT` option. This new file has the same name as the source file, only the extension is modified into `MAC`.

- `G`: this switch defines whether AS should produce code or not. If switched off, the processing will be stopped after the macro processor. This switch is activated by default (logically, otherwise you would not get a code file). This switch can be used in conjunction with the `P` switch, if only the macro processor of AS shall be used.

- `r [n]`: issue warnings if situations occur that force a further pass. This information can be used to reduce the number of passes. You may optionally specify the number of the first pass where issuing of such messages shall start. Without this argument, warnings will come starting with the first pass. Be prepared for a bunch of messages!!

- `cpu <name>`: this switch allows to set the target processor AS shall generate code for, in case the source file does not contain a `CPU` instruction and is not 68008 code.

- `alias <new>=<old>`: defines the processor type `<new>` to be an alias for the type `<old>`. See section 2.14 for the sense of processor aliases.

- `gnuerrors`: display messages about errors resp. warnings not in the As standard format, but instaed in a format similar to the GNU C compiler. This simplifies the integration of AS into environments tuned for this format, however also suppresses the display of precise error positions in macro bodies!

As long as switches require no arguments and their concatenation does not result in a multi-letter switch, it is possible to specify several switches at one time, as in the following example :

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

All files `TEST*.ASM` as well as the file `FIRSTPROG.ASM` will be assembled, whereby listings of all files are displayed on the console terminal. Additional

sharefiles will be generated in the C- format. The assembler should search for additional include files in the directory `C:\AS\8051\INCLUDE`.

This example shows that the assembler assumes `ASM` as the default extension for source files.

A bit of caution should be applied when using switches that have optional arguments: if a file specification immediately follows such aswitch without the optional argument, AS will try to interprete the file specification as argument - what of course fails:

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

The solution in this case would either be to move the -g option the end or to specify an explicit MAP argument.

Beside from specifying options in the command line, permanently needed options may be placed in the environment variable `ASCMD`. For example, if someone always wants to have assembly listings and has a fixed directory for include files, he can save a lot of typing with the following command:

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

The environment options are processed before the command line, so options in the command line can override contradicting ones in the environment variable.

In the case of very long path names, space in the `ASCMD` variable may become a problem. For such cases a key file may be the alternative, in which the options can be written in the same way as in the command line or the `ASCMD`-variable. But this file may contain several lines each with a maximum length of 255 characters. In a key file it is important, that for options which require an argument, switches and argument have to be written in the **same** line. AS gets informed of the name of the key file by a `@` aheaded in the `ASCMD` variable, e.g.

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

In order to neutralize options in the `ASCMD` variable (or in the key file), prefix the option with a plus sign. For example, if you do not want to generate an assembly listing in an individual case, the option can be retracted in this way:

```
as +L <file>
```

Naturally it is not consequently logical to deny an option by a plus sign.... UNIX soit qui mal y pense.

References to key files may not only come from the `ASCMD` variable, but also directly from the command line. Similarly to the `ASCMD` variable, prepend the file's name with a character:

```
as @<file> ....
```

The options read from a key file in this situation are processed as if they had been written out in the command line in place of the reference, *not* like the key file referenced by the `ASCMD` variable that is processed prior to the command line options.

Referencing a key file from a key file itself is not allowed and will be answered wit an error message by AS.

In case that you like to start AS from another program or a shell and this shell hands over only lower-case or capital letters in the command line, the following workaround exists: if a tilde (`~`) is put in front of an option letter, the following letter is always interpreted as a lower-case letter. Similarly a `#` demands the interpretation as a capital letter. For example, the following transformations result for:

```
/~I ---> /i
-#u ---> -U
```

In dependence of the assembly's outcome, the assembler ends with the following return codes:

0 error free run, at maximum warnings occurred

1 The assembler displayed only its command-line parameters and terminated immediately afterwards.

2 Errors occurred during assembly, no code file has been produced.

3 A fatal error occurred what led to immediate termination of the run.

4 An error occurred already while starting the assembler. This may be a parameter error or a faulty overlay file.

255 An internal error occurred during initialization that should not occur in any case...reboot, try again, and contact me if the problem is reproducible!

Similar to UNIX, OS/2 extends an application's data segment on demand when the application really needs the memory. Therefore, an output like *OS/2*

```
511 KByte available memory
```

does not indicate a shortly to come system crash due to memory lack, it simply shows the distance to the limit when OS/2 will push up the data segment's size again...

As there is no compatible way in C under different operating systens to find out the amount of available memory resp. stack, both lines are missing completely from the statistics the C version prints. *UNIX*

2.5 Format of the Input Files

Like most assemblers, AS expects exactly one instruction per line (blank lines are naturally allowed as well). The lines must not be longer than 255 characters, additional characters are discarded.

A single line has following format:

```
[label[:]] <mnemonic>[.attr] [param[,param..]] [;comment]
```

A line may also be split over several lines in the source file, continuation characters chain these parts together to a single line. One must however consider that, due to the internal buffer structure, the total line must not be longer than 256 characters. Line references in error messages always relate to the last line of such a composed source line.

The colon for the label is optional, in case the label starts in the first column (the consequence is that a mnemonic must not start in column 1). It is necessary to set the colon in case the label does not start in the first column

so that AS is able to distinguish it from a mnemonic. In the latter case, there must be at least one space between colon and mnemonic if the processor belongs to a family that supports an attribute that denotes an instruction format and is separated from the mnemonic by a colon. This restriction is necessary to avoid ambiguities: a distinction between a mnemonic with format and a label with mnemonic would otherwise be impossible.

Some signal processor families from Texas Instruments optionally use a double line (||) in place of the label to signify the prallel execution with the previous instruction(s). If these two assembler instructions become a single instruction word at machine level (C3x), an additional label in front of the second instruction of course does not make sense and is not allowed. The situation is different for the C6x with its instruction packets of variable length: If someone wants to jump into the middle of an instruction packet (bad style, if you ask me...), he has to place the necessary label *before* into a separate line. The same is valid for conditions, which however may be combined with the double line in a single source line.

The attribute is used by a couple of processors to specify variations or different codings of a certain instruction. The most prominent usage of the attibute is is the specification of the operand size, for example in the case of the 680x0 family (table 2.6).

attribute	arithmetic-logic instruction	jump instruction
B	byte (8 bits)	———
W	word (16 bits)	———
L	long word (32 bits)	16-bit-displacement
Q	quad word (64 bits)	———
S	single precision (32 bits)	8-bit-displacement
D	double precision (64 bits)	———
X	extended precision (80/96 bits)	32-bit-displacement
P	decimal floating point (80/96 bits)	———

Table 2.6: Allowed Attributes (Example 680x0)

Since this manual is not also meant as a user's manual for the processor families supported by AS, this is unfortunately not the place to enumerate all possible attributes for all families. It should however be mentioned that

in general, not all instructions of a given instruction set allow all attributes and that the omission of an attribute generally leads to the usage of the "natural" operand size of a processor family. For more thorough studies, consult a reasonable programmer's manual, e.g. [1] for the 68K's.

In the case of TLCS-9000, H8/500, and M16(C), the attribute serves both as an operand size specifier (if it is not obvious from the operands) and as a description of the instruction format to be used. A colon has to be used to separate the format from the operand size, e.g. like this:

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

This example does not show that there may be a format specification without an operand size. In contrast, if an operand size is used without a format specification, AS will automatically use the shortest possible format. The allowed formats and operand sizes again depend on the machine instruction and may be looked up e.g. in [106], [24], [44], resp. [45].

The number of instruction parameters depends on the mnemonic and is principally located between 0 and 20. The separation of the parameters from each other is to be performed only by commas (exception: DSP56xxx, its parallel data transfers are separated with blanks). Commas that are included in brackets or quotes, of course, are not taken into consideration.

Instead of a comment at the end, the whole line can consist of comment if it starts in the first column with a semicolon.

To separate the individual components you may also use tabulators instead of spaces.

2.6 Format of the Listing

The listing produced by AS using the command line options i or I is roughly divisible into the following parts :

1. issue of the source code assembled;
2. symbol list;

3. usage list;

4. cross reference list.

The two last ones are only generated if they have been demanded by additional command line options.

In the first part, AS lists the complete contents of all source files including the produced code. A line of this listing has the following form:

```
[<n>] <line>/<address> <code> <source>
```

In the field `n`, AS displays the include nesting level. The main file (the file where assembly was started) has the depth 0, an included file from there has depth 1 etc.. Depth 0 is not displayed.

In the field `line`, the source line number of the referenced file is issued. The first line of a file has the number 1. The address at which the code generated from this line is written follows after the slash in the field `address`.

The code produced is written behind `address` in the field `code`, in hexadecimal notation. Depending on the processor type and actual segment the values are formatted either as bytes or 16/32-bit-words. If more code is generated than the field can take, additional lines will be generated, in which case only this field is used.

Finally, in the field `source`, the line of the source file is issued in its original form.

The symbol table was designed in a way that it can be displayed on an 80-column display whenever possible. For symbols of "normal length", a double column output is used. If symbols exceed (with their name and value) the limit of 40 columns (characters), they will be issued in a separate line. The output is done in alphabetical order. Symbols that have been defined but were never used are marked with a star (*) as prefix.

The parts mentioned so far as well as the list of all macros/functions defined can be selectively masked out from the listing. This can be done by the already mentioned command line switch `-t`. There is an internal byte inside AS whose bits represent which parts are to be written. The assignment of bits to parts of the listing is listed in table 2.7.

All bits are set to 1 by default, when using the switch

bit	part
0	source file(s) + produced code
1	symbol table
2	macro list
3	function list
4	line numbering
5	register symbol list
7	character set table

Table 2.7: Assignment of Bits to Listing Components

`-t <mask>`

Bits set in <mask> are cleared, so that the respective listing parts are suppressed. Accordingly it is possible to switch on single parts again with a plus sign, in case you had switched off too much with the `ASCMD` variable... If someone wants to have, for example, only the symbol table, it is enough to write:

```
-t 2
```

The usage list issues the occupied areas hexadecimally for every single segment. If the area has only one address, only this is written, otherwise the first and last address.

The cross reference list issues any defined symbol in alphabetical order and has the following form:

```
symbol <symbol name> (=<value>,<file>/<line>):
 file <file 1>:
 <n1>[(m1)]  ..... <nk>[(mk)]
 .
 .
 file <file l>:
 <n1>[(m1)]  ..... <nk>[(mk)]
```

The cross reference list lists for every symbol in which files and lines it has been used. If a symbol was used several times in the same line, this would be indicated by a number in brackets behind the line number. If a symbol

was never used, it would not appear in the list; The same is true for a file that does not contain any references for the symbol in question.

CAUTION! AS can only print the listing correctly if it was previously informed about the output media's page length and width! This has to be done with the `PAGE` instruction (see there). The preset default is a length of 60 lines and an unlimited line width.

2.7 Symbol Conventions

Symbols are allowed to be up to 255 characters long (as hinted already in the introduction) and are being distinguished on the whole length, but the symbol names have to meet some conventions:

Symbol names are allowed to consist of a random combination of letters, digits, underlines and dots, whereby the first character must not be a digit. The dot is only allowed to meet the MCS-51 notation of register bits and should - as far as possible - not be used in own symbol names. To separate symbol names in any case the underline (`_`) and not the dot (`.`) should be used .

AS is by default not case-sensitive, i.e. it does not matter whether one uses upper or lower case characters. The command line switch `U` however allows to switch AS into a mode where upper and lower case makes a difference. The predefined symbol `CASESENSITIVE` signifies whether AS has been switched to this mode: TRUE means case-sensitiveness, and FALSE its absence.

Table 2.8 shows the most important symbols which are predefined by AS. **CAUTION!** While it does not matter in case-sensitive mode which combination of upper and lower case to use to reference predefined symbols, one has to use exactly the version given above (only upper case) when AS is in case-sensitive mode!

Additionally some pseudo instructions define symbols that reflect the value that has been set with these instructions. Their descriptions are explained at the individual commands belonging to them.

A hidden feature (that has to be used with care) is that symbol names may be assembled from the contents of string symbols. This can be achieved by

name	meaning
TRUE	logically "true"
FALSE	logically "false"
CONSTPI	Pi (3.1415.....)
VERSION	version of AS in BCD-coding, e.g. 1331 hex for version 1.33p1
ARCHITECTURE	target platform AS was compiled for, in the style processor-manufacturer-operating system
DATE	date and
TIME	time of the assembly (start)
MOMCPU	current target CPU (see the CPU instruction)
MOMFILE	current source file
MOMLINE	line number in source file
MOMPASS	number of the currently running pass
MOMSECTION	name of the current section or an empty string
*, $ resp. PC	current value of program counter

Table 2.8: Predefined Symbols

framing the string symbol's name with braces and inserting it into the new symbol's name. This allows for example to define a symbol's name based on the value of another symbol:

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

CAUTION: The programmer has to assure that only valid symbol names are generated!

A complete list of all symbols predefined by AS can be found in appendix E.

Apart from its value, every symbol also owns a marker which signifies to which *segment* it belongs. Such a distinction is mainly needed for processors

that have more than one address space. The additional information allows AS to issue a warning when a wrong instruction is used to access a symbol from a certain address space. A segment attribute is automatically added to a symbol when is gets defined via a label or a special instruction like `BIT`; a symbol defined via the "allround instructions" `SET` resp. `EQU` is however "typeless", i.e. its usage will never trigger warnings. A symbol's segment attribute may be queried via the buit-in function `SYMTYPE`, e.g.:

```
Label:
        .
        .
Attr    equ     symtype(Label)  ; results in 1
```

The individual segment types have the assigned numbers listed in table 2.9. Register symbols which do not really fit into the order of normal symbols are explained in section 2.12. The `SYMTYPE` function delivers -1 as result when called with an undefined symbol as argument.

segment	return value
<none>	0
CODE	1
DATA	2
IDATA	3
XDATA	4
YDATA	5
BITDATA	6
IO	7
REG	8
ROMDATA	9
<register symbol>	128

Table 2.9: return values of the `SYMTYPE` function

2.8 Temporary Symbols

Especially when dealing with programs that contain sequences of loops of if-like statements, one is continuously faced with the problem of inventing new

names for labels - labels of which you know exactly that you will never need to reference them again afterwards and you really would like to get 'rid' of them somehow. A simple solution if you don't want to swing the large hammer of sections (see chapter 3.8) are *temporary* symbols which remain valid as long as a new, non-temporary symbol gets defined. Other assemblers offer a similar mechanism which is commonly referred as 'local symbols'; however, for the sake of a better distinction, I want to stay with the term 'temporary symbols'. AS knows three different types of temporary symbols, in the hope to offer everyone 'switching' to AS a solution that makes conversion as easy as possible. However, practically every assembler has its own interpretation of this feature, so there will be only few cases where a 1:1 solution for existing code:

2.9 Named Temporary Symbols

A symbol whose name starts with two dollar signs (something that is neither allowed for non-temporary symbols nor for constants) is a named temporary symbol. AS keeps an internal counter which is reset to 0 before assembly begins and which gets incremented upon every definition of a non-temporary symbol. When a temporary symbol is defined or referenced, both leading dollar signs are discarded and the counter's current value is appended. This way, one regains the used symbol names with every definition of a non-temporary symbol - but you also cannot reach the previously symbols any more! Temporary symbols are therefore especially suited for usage in small instruction blocks, typically a dozen of machine instructions, definitely not more than one screen. Otherwise, one easily gets confused...

Here is a small example:

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

split:

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

Without the non-temporary label between the loops, of course an error message about a double-defined symbol would be the result.

2.9.1 Nameless Temporary Symbols

For all those who regard named temporary symbols still as too complicated, there is an even simpler variant: If one places a single puls or minus sign as a label, this is converted to symbol names of `__forwnn` respectively `__backmm`, with `nn` respectively `mm` being counters that start counting at zero. Those symbols are referenced via the special names `- -- ---` respectively `+ ++ +++`, which refer to the three last 'minus symbols' and the next three 'plus symbols'. Therefore, the selection between these two variants depends on whether one wants to forward- or backward-reference a symbol.

Apart from plus and minus, *defining* nameless temporary symbols also exists in a third variant, namely a slash (/). A temporary symbol defined in this way may be referenced both backward and forward, i.e. it is treated either as a plus or a minus, depending on the way it is being referenced.

Nameless temporary symbols are usually used in constructs that fit on one screen page, like skipping a few machine instructions or tight loops - things would become to puzzling otherwise (this only a good advice, however...). An example for this is the following piece of code, this time as 65xx code:

```
        cpu     6502

-       ldx     #00
-       dex
        bne     -           ; branch to 'dex'
        lda     RealSymbol
        beq     +           ; branch to 'bne --'
        jsr     SomeRtn
        iny
+       bne     --          ; branch to 'ldx #00'

SomeRtn:
        rts

RealSymbol:
        dfs     1

   inc ptr
```

```
    bne  +           ; branch to 'tax'
    inc  ptr+1
+  tax

  bpl  ++          ; branch to 'dex'
    beq  +           ; branch forward to 'rts'
    lda  #0
/   rts                 ; slash used as wildcard.
+  dex
    beq  -           ; branch backward to 'rts'

ptr: dfs 2
```

2.9.2 Composed Temporary Symbols

This is maybe the type of temporary symbols that is nearest to the concept of local symbols and sections. Whenever a symbol's name begins with a dot (.), the symbol is not directly stored with this name in the symbol table. Instead, the name of the most recently-defined symbol not beginning with a dot is prepended to the symbols name. This way, 'non-dotted' symbols take the role of section separators and 'dotted' symbol names may be reused after a 'non-dotted' symbol has been defined. Take a look at the following little example:

```
proc1: ; non-temporary symbol 'proc1'

.loop moveq #20,d0 ; actually defines 'proc1.loop'
dbra d0,.loop
rts

proc2: ; non-temporary symbol 'proc2'

.loop moveq #10,d1 ; actually defines 'proc2.loop'
jsr proc1
dbra d1,.loop
rts
```

Note that it is still possible to access all temporary symbols, even without being in the same 'area', by simply using the composed name (like 'proc2.loop' in the previous example).

It is principally possible to combine composed temporary symbols with sections, which makes them also to local symbols. Take however into account that the most recent non-temporary symbol is not stored per-section, but simply globally. This may change however in a future version, so one shouldn't rely on the current behaviour.

2.10 Formula Expressions

In most places where the assembler expects numeric inputs, it is possible to specify not only simple symbols or constants, but also complete formula expressions. The components of these formula expressions can be either single symbols and constants. Constants may be either integer, floating point, or string constants.

2.10.1 Integer Constants

Integer constants describe non-fractional numbers. They may either be written as a sequence of digits or as a sequence of characters enclosed in *single* quotation marks. In case they are written as a sequence of digits, this may be done in different numbering systems (table 2.10).

In case the numbering system has not been explicitly stated by adding the special control characters listed in the table, AS assumes the base given with the `RADIX` statement (which has itself 10 as default). This statement allows to set up 'unusual' numbering systems, i.e. others than 2, 8, 10, or 16.

Valid digits are numbers from 0 to 9 and letters from A to Z (value 10 to 35) up to the numbering system's base minus one. The usage of letters in integer constants however brings along some ambiguities since symbol names also are sequences of numbers and letters: a symbol name must not start with a character from 0 to 9. This means that an integer constant which is not clearly marked a such with a special prefix character never mav begin with

	Intel mode (Intel, Zilog, Thomson Texas, Toshiba, NEC, Siemens, Philips, Fujitsu, Fairchild, Intersil)	Motorola mode (Rockwell, Motorola, Microchip, Thomson, Hitachi)	C mode (PowerPC, AMD 29K, National, Symbios, Atmel)
decimal	direct	direct	direct
hexadecimal	followed by H	prepended $	prepended 0x
binary	followed by B	prepended %	prepended 0b
octal	followed by O followed by Q	prepended @	prepended 0

Table 2.10: Possible Numbering Systems

a letter. One has to add an additional, otherwise superfluous zero in front in such cases. The most prominent case is the writing of hexadecimal constants in Intel mode: If the leftmost digit is between A and F, the trailing H doesn't help anything, an additional 0 has to be prefixed (e.g. 0F0H instead of F0H). The Motorola and C syntaxes whis both mark the numbering system at the front of a constant do not have this problem (*hehehe..*).

Quite tricky is furthermore that the higher the default numbering system set via `RADIX` becomes, the more letters used to denote numbering systems in Intel and C syntax become 'eaten'. For example, you cannot write binary constants anymore after a `RADIX 16`, and starting at `RADIX 18`, the Intel syntax even doesn't allow to write hexadecimal constants any more. Therefore **CAUTION!**

With the help of the `RELAXED` instruction (see section 3.9.6), the strict assignment of a syntax to a certain target processor can be removed. The result is that an arbitrary syntax may be used (loosing compatibility to standard assemblers). This option is however turned off by default. This option also opens the opportunity for a fourth way of writing integer constants, a way that is sometimes found on other assemblers: This way puts the actual value into apostrophes and prepends the numbering system ('x' or 'h' for hexadecimal, 'o' for octal and 'b' for binary). So, the integer constant 305419896 can be written in the following ways:

```
x'12345678'
h'12345678'
o'2215053170'
b'00010010000110100010101100111000'
```

This syntax is noth the default for any processor architecture and only available in `RELAXED` mode. Its main purpose is the easier porting of existing sources and is not recommended for new programs.

Integer constants may also be written as ASCII values, like in the following examples:

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

It is important to write the characters in *single quotes*, to distinguish them from string constants (discussed somewhat later).

2.10.2 Floating Point Constants

Floating point constants are to be written in the usual scientific notation, which is known in the most general form:

```
[-]<integer digits>[.post decimal positions][E[-]exponent]
```

CAUTION! The assembler first tries to interprete a constant as an integer constant and makes a floating-point format try only in case the first one failed. If someone wants to enforce the evaluation as a floating point number, this can be done by dummy post decimal positions, e.g. `2.0` instead of `2`.

2.10.3 String Constants

String constants have to be included in *double quotation* marks (to distinguish them from the abovementioned ASCII-integers). In order to make it possible to write quotation marks and special characters without trouble in string constants, an "escape mechanism" has been implemented, which should sound familiar for C programmers:

The assembler understands a backslash (\) with a following decimal number of three digits maximum in the string as a character with the according decimal ASCII value. The numerical value may alternitavely be written in hexadecimal or octal notation if it is prefixed with an x resp. a 0. In case of hexadecimal notation, the maximum number of digits is limited to 2. For example, it is possible to include an ETC character by writing `\3`. But be careful with the definition of NUL characters! The C version currently uses *UNIX* C strings to store strings internally. As C strings use a NUL character for termination, the usage of NUL characters in strings is currently not portable!

Some frequently used control characters can also be reached with the following abbreviations:

```
\b : Backspace          \a : Bell         \e : Escape
\t : Tabulator          \n : Linefeed     \r : Carriage Return
\\ : Backslash          \' or \H : Apostrophe
\" or \I : Quotation marks
```

Both upper and lower case characters may be used for the identification letters.

By means of this escape character, you can even work formula expressions into a string, if they are enclosed by braces: e.g.

```
message "root of 81 : \{sqrt(81)}"
```

results in

```
root of 81 : 9
```

AS chooses with the help of the formula result type the correct output format, further string constants, however, are to be avoided in the expression. Otherwise the assembler will get mixed up at the transformation of capitals into lower case letters. Integer results will by default be written in hexadecimal notation, which may be changed via the `OUTRADIX` instruction.

Except for the insertion of formula expressions, you can use this "escape-mechanism" as well in ASCII defined integer constants, like this:

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

However, everything has its limits, because the parser with higher priority, which disassembles a line into op-code and parameters, does not know what it is actually working with, e.g. here:

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

After the third apostrophe, it will not find the comma any more, because it presumes that it is the start of a further character constant. An error message about a wrong parameter number is the result. A workaround would be to write e.g., `\i` instead of `\'`.

2.10.4 Evaluation

The calculation of intermediary results within formula expressions is always done with the highest available resolution, i.e. 32 bits for integer numbers, 80 bit for floating point numbers and 255 characters for strings. An possible test of value range overflows is done only on the final result.

UNIX The portable C version only supports floating point values up to 64 bits (resulting in a maximum value of roughly 10^{308}), but in turn features integer lengths of 64 bits on some platforms.

2.10.5 Operators

The assembler provides the operands listed in table 2.11 for combination. "Rank" is the priority of an operator at the separation of expressions into subexpressions. The operator with the highest rank will be evaluated at the very end. The order of evaluation can be defined by new bracketing.

The compare operators deliver TRUE in case the condition fits, and FALSE in case it doesn't. For the logical operators an expression is TRUE in case it is not 0, otherwise it is FALSE.

The mirroring of bits probably needs a little bit of explanation: the operator mirrors the lowest bits in the first operand and leaves the higher priority bits unchanged. The number of bits which is to be mirrored is given by the right operand and may be between 1 and 32 .

A small pitfall is hidden in the binary complement: As the computation is always done with 32 resp. 64 bits, its application on e.g. 8-bit masks usually results in values taht do not fit into 8-bit numbers any more due to the leading ones. A binary AND with a fitting mask is therefore unavoidable!

operand	function	#operands	integer	float	string	rank
<>	inequality	2	yes	yes	yes	14
>=	greater or equal	2	yes	yes	yes	14
<=	less or equal	2	yes	yes	yes	14
<	truly smaller	2	yes	yes	yes	14
>	truly greater	2	yes	yes	yes	14
=	equality	2	yes	yes	yes	14
==	alias for =					
!!	log. XOR	2	yes	no	no	13
||	log. OR	2	yes	no	no	12
&&	log. AND	2	yes	no	no	11
~~	log. NOT	1	yes	no	no	2
-	difference	2	yes	yes	no	10
+	sum	2	yes	yes	yes	10
#	modulo division	2	yes	no	no	9
/	quotient	2	yes*)	yes	no	9
*	product	2	yes	yes	no	9
^	power	2	yes	yes	no	8
!	binary XOR	2	yes	no	no	7
|	binary OR	2	yes	no	no	6
&	binary AND	2	yes	no	no	5
><	mirror of bits	2	yes	no	no	4
>>	log. shift right	2	yes	no	no	3
<<	log. shift left	2	yes	no	no	3
~	binary NOT	1	yes	no	no	1
*) remainder will be discarded						

Table 2.11: Operators Predefined by AS

2.10.6 Functions

In addition to the operators, the assembler defines another line of primarily transcendental functions with floating point arguments which are listed in tables 2.12 and 2.13. The functions `FIRSTBIT`, `LASTBIT`, and `BITPOS` return -1 as result if no resp. not exactly one bit is set. `BITPOS` additionally issues an error message in such a case.

The string function `SUBSTR` expects the source string as first parameter, the start position as second and the number of characters to be extracted as third parameter (a 0 means to extract all characters up to the end). Similarly, `CHARFROMSTR` expects the source string as first argument and the character position as second argument. In case the position argument is larger or equal to the source string's length, `SUBSTR` returns an empty string while `CHARFROMSTR` returns -1. A position argument smaller than zero is treated as zero by `SUBSTR`, while `CHARFROMSTR` will return -1 also in this case.

Here is an example how to use these both functions. The task is to put a string into memory, with the string end being signified by a set MSB in the last character:

```
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` returns the first occurence of the second string within the first one resp. -1 if the search pattern was not found. Similarly to `SUBSTR` and `CHARFROMSTR`, the first character has the position 0.

If a function expects floating point arguments, this does not mean it is impossible to write e.g.

```
    sqr2 equ sqrt(2)
```

name	meaning	argument	result
SQRT	square root	$arg \geq 0$	floating point
SIN	sine	$arg \in \mathbb{R}$	floating point
COS	cosine	$arg \in \mathbb{R}$	floating point
TAN	tangent	$arg \neq (2n+1) * \frac{\pi}{2}$	floating point
COT	cotangent	$arg \neq n * \pi$	floating point
ASIN	inverse sine	$\mid arg \mid \leq 1$	floating point
ACOS	inverse cosine	$\mid arg \mid \leq 1$	floating point
ATAN	inverse tangent	$arg \in \mathbb{R}$	floating point
ACOT	inverse cotangent	$arg \in \mathbb{R}$	floating point
EXP	exponential function	$arg \in \mathbb{R}$	floating point
ALOG	10 power of argument	$arg \in \mathbb{R}$	floating point
ALD	2 power of argument	$arg \in \mathbb{R}$	floating point
SINH	hyp. sine	$arg \in \mathbb{R}$	floating point
COSH	hyp. cosine	$arg \in \mathbb{R}$	floating point
TANH	hyp. tangent	$arg \in \mathbb{R}$	floating point
COTH	hyp. cotangent	$arg \neq 0$	floating point
LN	nat. logarithm	$arg > 0$	floating point
LOG	dec. logarithm	$arg > 0$	floating point
LD	bin. logarithm	$arg > 0$	floating point
ASINH	inv. hyp. Sine	$arg \in \mathbb{R}$	floating point
ACOSH	inv. hyp. Cosine	$arg \geq 1$	floating point
ATANH	inv. hyp. Tangent	$arg < 1$	floating point
ACOTH	inv. hyp. Cotangent	$arg > 1$	floating point
INT	integer part	$arg \in \mathbb{R}$	floating point
BITCNT	number of one's	integer	integer
FIRSTBIT	lowest 1-bit	integer	integer

Table 2.12: Functions Predefined by AS - Part 1 (Integer and Floating Point Functions

name	meaning	argument	result
LASTBIT	highest 1-bit	integer	integer
BITPOS	unique 1-bit	integer	integer
SGN	sign (0/1/-1)	floating point or integer	integer
ABS	absolute value	integer or floating point	integer or floating point
TOUPPER	matching capital	integer	integer
TOLOWER	matching lower case	integer	integer
UPSTRING	changes all characters into capitals	string	string
LOWSTRING	changes all characters into to lower case	string	string
STRLEN	returns the length of a string	string	integer
SUBSTR	extracts parts of a string	string, integer, integer	string
CHARFROMSTR	extracts a character from a string	string, integer	integer
STRSTR	searches a substring in a string	string, string	integer
VAL	evaluates contents as expression	string	depends on argument
EXPRTYPE	delivers type of argument	integer, float, string	0 1 2

Table 2.13: Functions Predefined by AS - Part 2 (Integer and String Functions

In such cases an automatic type conversion is engaged. In the reverse case the `INT`-function has to be applied to convert a floating point number to an integer. When using this function, you have to pay attention that the result produced always is a signed integer and therefore has a value range of approximately +/-2.0E9.

When AS is switched to case-sensitive mode, predefined functions may be accessed with an arbitrary combination of upper and lower case (in contrast to predefined symbols). However, in the case of user-defined functions (see section 3.4.9), a distinction between upper and lower case is made. This has e.g. the result that if one defines a function `Sin`, one can afterwards access this function via `Sin`, but all other combinations of upper and lower case will lead to the predefined function.

For a correct conversion of lower case letters into capital letters a DOS version $\geq$ 3.30 is required. *DOS/DPMI*

2.11 Forward References and Other Disasters

This section is the result of a significant amount of hate on the (legal) way some people program. This way can lead to trouble in conjunction with AS in some cases. The section will deal with so-called 'forward references'. What makes a forward reference different from a usual reference? To understand the difference, take a look at the following programming example (please excuse my bias for the 68000 family that is also present in the rest of this manual):

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

If one overlooks the loop body with its branch statement, a program remains that is extremely simple to assemble: the only reference is the branch back

to the body's beginning, and as an assembler processes a program from the beginning to the end, the symbol's value is already known before it is needed the first time. If one has a program that only contains such backward references, one has the nice situation that only one pass through the source code is needed to generate a correct and optimal machine code. Some high level languages like Pascal with their strict rule that everything has to be defined before it is used exploit exactly this property to speed up the compilation.

Unfortunately, things are not that simple in the case of assembler, because one sometimes has to jump forward in the code or there are reasons why one has to move variable definitions behind the code. For our example, this is the case for the conditional branch that is used to skip over another instruction. When the assembler hits the branch instruction in the first pass, it is confronted with the situation of either leaving blank all instruction fields related to the target address or offering a value that "hurts noone" via the formula parser (which has to evaluate the address argument). In case of a "simple" assembler that supports only one target architecture with a relatively small number of instructions to treat, one will surely prefer the first solution, but the effort for AS with its dozens of target architectures would have become extremely high. Only the second way was possible: If an unknown symbol is detected in the first pass, the formula parser delivers the program counter's current value as result! This is the only value suitable to offer an address to a branch instruction with unknown distance length that will not lead to errors. This answers also a frequently asked question why a first-pass listing (it will not be erased e.g. when AS does not start a second pass due to additional errors) partially shows wrong addresses in the generated binary code - they are the result of unresolved forward references.

The example listed above however uncovers an additional difficulty of forward references: Depending on the distance of branch instruction and target in the source code, the branch may be either long or short. The decision however about the code length - and therefore about the addresses of following labels - cannot be made in the first pass due to missing knowledge about the target address. In case the programmer did not explicitly mark whether a long or short branch shall be used, genuine 2-pass assemblers like older versions of MASM from Microsoft "solve" the problem by reserving space for the longest version in the first pass (all label addresses have to be fixed after the first pass) and filling the remaining space with `NOP`s in the second pass. AS versions up to 1.37 did the same before I switched to the multipass principle

that removes the strict separation into two passes and allows an arbitrary number of passes. Said in detail, the optimal code for the assumed values is generated in the first pass. In case AS detects that values of symbols changed in the second pass due to changes in code lengths, simply a third pass is done, and as the second pass'es new symbol values might again shorten or lengthen the code, a further pass is not impossible. I have seen 8086 programs that needed 12 passes to get everything correct and optimal. Unfortunately, this mechanism does not allow to specify a maximum number passes; I can only advise that the number of passes goes down when one makes more use of explicit length specifications.

Especially for large programs, another situation might arise: the position of a forward directed branch has moved so much in the second pass relative to the first pass that the old label value still valid is out of the allowed branch distance. AS knows of such situations and suppresses all error messages about too long branches when it is clear that another pass is needed. This works for 99% of all cases, but there are also constructs where the first critical instruction appears so early that AS had no chance up to now to recognize that another pass is needed. The following example constructs such a situation with the help of a forward reference (and was the reason for this section's heading...):

```
        cpu   6811

        org     $8000
        beq     skip
        rept    60
         ldd    Var
        endm
skip:   nop

Var     equ     $10
```

Due to the address position, AS assumes long addresses in the first pass for the `LDD` instructions, what results in a code length of 180 bytes and an out of branch error message in the second pass (at the point of the `BEQ` instruction, the old value of `skip` is still valid, i.e. AS does not know at this point that the code is only 120 bytes long in reality) is the result. The error can be avoided in three different ways:

1. Explicitly tell AS to use short addressing for the `LDD` instructions (`ldd <Var`)

2. Remove this damned, rotten forward reference and place the `EQU` statement at the beginning where it has to be (all right, I'm already calming down...)

3. For real die-hards: use the `-Y` command line option. This option tells AS to forget the error message when the address change has been detected. Not pretty, but...

Another tip regarding the `EQU` instruction: AS cannot know in which context a symbol defined with `EQU` will be used, so an `EQU` containing forward references will not be done at all in the first pass. Thus, if the symbol defined with `EQU` gets forward-referenced in the second pass:

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

one gets an error message due to an undefined symbol in the second pass...but why on earth do people do such things?

Admittedly, this was quite a lengthy excursion, but I thought it was necessary. Which is the essence you should learn from this section?

1. AS always tries to generate the shortest code possible. A finite number of passes is needed for this. If you do not tweak AS extremely, AS will know no mercy...

2. Whenever sensible and possible, explicitly specify branch and address lengths. There is a chance of significantly reducing the number of passes by this.

3. Limit forward references to what is absolutely needed. You make your and AS's live much easier this way!

2.12 Register Symbols

valid for: PowerPC, M-Core, 4004/4040, 80C16x, AVR

Sometimes it is desirable not only to assign symbolic names to memory addresses or constants, but also to a register, to emphasize its function in a certain program section. This is no problem for processors that treat registers simply as another address space, as this allows to use numeric expressions and one can use simple `EQU`s to define such symbols. (e.g. for the MCS-96 or TMS70000). However, for most processors, register identifiers are fixed literals which are seperately treated by AS for speed reasons. A special mechanism is therefore necessary to define symbolic register names. A register symbol is usually defined via the `REG` instruction, which has otherwise the same syntax as an `EQU` definition. This however has a couple of restrictions: A register symbol is a pure character string stored 'as is' which may exclusively be used this way. For example, no arithmetic is allowed to calculate a register's successor, like in the following example:

```
myreg   reg     r17         ; definition of register symbol
        addi    myreg+1,3   ; does not work!
```

Additionally, a register symbol has to be defined prior to ist first usage; a forward reference would have the result that AS suspects a forward reference to a memory location in case a register symbol is not found. Since the usage of memory operands is far more limited on most processors, a bunch of errors would be the result...

Analogous to ordinary symbols, register symbols are local to sections and it is possible to access a register symbol from a specific section by appending the section's name enclosed in brackets. Due to the missing ability to do forward references, there is nothing like a `FORWARD` directive, and an export by something comparable to `PUBLIC` or `GLOBAL` is also not possible since register symbols generally have their meaning in a small context.

If there is both an ordinary and a register symbol of same name present in a context, the register symbol will be preferred. This is however not the case when the name is embedded into a complex expression (parentheses are sufficient!), the normal symbol will be used then.

2.13 Sharefile

This function is a by-product from the old pure-68000 predecessors of AS, I have kept them in case someone really needs it. The basic problem is to access certain symbols produced during assembly, because possibly someone would like to access the memory of the target system via this address information. The assembler allows to export symbol values by means of `SHARED` pseudo commands (see there). For this purpose, the assembler produces a text file with the required symbols and its values in the second pass. This file may be included into a higher-level language or another assembler program. The format of the text file (C, Pascal or Assembler) can be set by the command line switches `p, c` or, `a`.

CAUTION! If none of the switches is given, no file will be generated and it makes no difference if `SHARED`-commands are in the source text or not!

When creating a Sharefile, AS does not check if a file with the same name already exists, such a file will be simply overwritten. In my opinion a request does not make sense, because AS would ask at each run if it should overwrite the old version of the Sharefile, and that would be really annoying...

2.14 Processor Aliases

Common microcontroller families are like rabbits: They become more at a higher speed than you can provide support for them. Especially the development of processor cores as building blocks for ASICs and of microcontroller families with user-definable peripherals has led to a steeply rising number of controllers that only deviate from a well-known type by a slightly modified peripheral set. But the distinction among them is still important, e.g. for the design of include files that only define the appropriate subset of peripherals. I have struggled up to now to integrate the most important reperesentatives of a processor family into AS (and I will continue to do this), but sometimes I just cannot keep pace with the development...there was an urgent need for a mechanism to extend the list of processors by the user.

The result are processor aliases: the alias command line option allows to define a new processor type, whose instruction set is equal to another processor

built into AS. After switching to this processor via the `CPU` instruction, AS behaves exactly as if the original processor had been used, with a single difference: the variables `MOMCPU` resp. `MOMCPUNAME` are set to the alias name, which allows to use the new name for differentiation, e.g. in include files.

There were two reasons to realize the definition of aliases by the command line and not by pseudo instructions: first, it would anyway be difficult to put the alias definitions together with register definitions into a single include file, because a program that wants to use such a file would have to include it before and after the CPU instruction - an imagination that lies somewhere between inelegant and impossible. Second, the definition in the command line allows to put the definitions in a key file that is executed automatically at startup via the `ASCMD` variable, without a need for the program to take any further care about this.

Chapter 3

Pseudo Instructions

Not all pseudo instructions are defined for all processors. A note that shows the range of validity is therefore prepended to every individual description.

3.1 Definitions

3.1.1 SET, EQU, and CONSTANT

valid for: all processors, `CONSTANT` *only for KCPSM(3)*

`SET` and `EQU` allow the definition of typeless constants, i.e. they will not be assigned to a segment and their usage will not generate warnings because of segment mixing. `EQU` defines constants which can not be modified (by `EQU`) again, but `SET` permits the definition of variables, which can be modified during the assembly. This is useful e.g. for the allocation of resources like interrupt vectors, as shown in the following example:

```
VecCnt  set     0       ; somewhere at the beginning
        .
        .
        .
DefVec  macro   Name    ; allocate a new vector
Name    equ     VecCnt
```

```
VecCnt  set     VecCnt+4
        endm
        .
        .
        .
        DefVec  Vec1    ; results in Vec1=0
        DefVec  Vec2    ; results in Vec2=4
```

constants and variables are internally stored in the same way, the only difference is that they are marked as unchangeable if defined via `EQU`. Trying to change a constant with `SET` will result in an error message.

`EQU/SET` allow to define constants of all possible types, e.g.

```
IntTwo   equ    2
FloatTwo equ    2.0
```

Some processors unfortunately have already a `SET` instruction. For these targets, `EVAL` must be used instead of `SET`.

A simple equation sign may be used instead of `EQU`. Similarly, one may simply write `:=` instead of `SET` resp. `EVAL`.

For compatibility reasons to the original assembler, the KCPSM target also knows the `CONSTANT` statement, which - in contrast to `EQU` - takes both name and value as argument. For example:

```
      CONSTANT  const1, 2
```

`CONSTANT` is however limited to integer constants.

Symbols defined with `SET` or `EQU` are typeless by default, but optionally a segment name (`CODE, DATA, IDATA, XDATA, YDATA, BITDATA, IO`, or `REG`) or `MOMSEGMENT` for the currently active segment may be given as a second parameter, allowing to assign the symbol to a specific address space. AS does not check at this point if the used address space exists on the currently active target processor!

3.1.2 SFR and SFRB

valid for: various, `SFRB` *only MCS-51*

These instructions act like `EQU`, but symbols defined with them are assigned to the directly addressable data segment, i.e. they serve preferential for the definition of RAM-cells and (as the name lets guess) hardware registers mapped into the data area. The allowed range of values is equal to the range allowed for `ORG` in the data segment (see section 3.2.1). The difference between `SFR` and `SFRB` is that `SFRB` marks the register as bit addressable, which is why AS generates 8 additional symbols which will be assigned to the bit segment and carry the names xx.0 to xx.7, e.g.

```
PSW     sfr     0d0h    ; results in PSW = D0H (data segment)

PSW     sfrb    0d0h    ; results in extra PSW.0 = D0H (bit)
                        ;               to PSW.7 = D7H (bit)
```

The `SFRB` instruction is not any more defined for the 80C251 as it allows direct bit access to all SFRs without special bit symbols; bits like `PSW.0` to `PSW.7` are automatically present.

Whenever a bit-addressable register is defined via `SFRB`, AS checks if the memory address is bit addressable (range 20h..3fh resp. 80h, 88h, 90h, 98h...0f8h). If it is not bit-addressable, a warning is issued and the generated bit symbols are undefined.

3.1.3 XSFR and YSFR

valid for: DSP56xxx

Also the DSP56000 has a few peripheral registers memory-mapped to the RAM, but the affair becomes complicated because there are two data areas, the X- and Y-area. This architecture allows on the one hand a higher parallelism, but forces on the other hand to divide the normal `SFR` instruction into the two above mentioned variations. They works identically to `SFR`, just that `XSFR` defines a symbol in the X- addressing space and YSFR a corresponding one in the Y-addressing space. The allowed value range is 0..$ffff.

3.1.4 LABEL

valid for: all processors

The function of the `LABEL` instruction is identical to `EQU`, but the symbol does not become typeless, it gets the attribute "code". `LABEL` is needed exactly for one purpose: Labels are normally local in macros, that means they are not accessible outside of a macro. With an `EQU` instruction you could get out of it nicely, but the phrasing

```
<name>  label   $
```

generates a symbol with correct attributes.

3.1.5 BIT

valid for: MCS/(2)51, XA, 80C166, 75K0, ST9

`BIT` serves to equate a single bit of a memory cell with a symbolic name. This instruction varies from target platform to target platform due to the different ways in which processors handle bit manipulation and addressing:

The MCS/51 family has an own address space for bit operands. The function of `BIT` is therefore quite similar to `SFR`, i.e. a simple integer symbol with the specified value is generated and assigned to the `BDATA` segment. For all other processors, bit addressing is done in a two-dimensional fashion with address and bit position. In these cases, AS packs both parts into an integer symbol in a way that depends on the currently active target processor and separates both parts again when the symbol is used. The latter is is also valid for the 80C251: While an instruction like

```
My_Carry bit    PSW.7
```

would assign the value 0d7h to `My_Carry` on an 8051, a value of 070000d0h would be generated on an 80C251, i.e. the address is located in bits 0..7 and the bit position in bits 24..26. This procedure is equal to the way the `DBIT` instruction handles things on a TMS370 and is also used on the 80C166, with the only difference that bit positions may range from 0..15:

```
MSB     BIT     r5.15
```

On a Philips XA, the bit's address is located in bits 0..9 just with the same coding as used in machine instructions, and the 64K bank of bits in RAM memory is placed in bits 16..23.

The `BIT` instruction of the 75K0 family even goes further: As bit expressions may not only use absolute base addresses, even expressions like

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

are allowed.

The ST9 in turn allows to invert bits, what is also allowed in the `BIT` instruction:

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

More about the ST9's `BIT` instruction can be found in the processor specific hints.

3.1.6 DBIT

valid for: TMS 370xxx

Though the TMS370 series does not have an explicit bit segment, single bit symbols may be simulated with this instruction. `DBIT` requires two operands, the address of the memory cell that contains the bit and the exact position of the bit in the byte. For example,

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

defines the bit that enables interrupts via the INT3 pin. Bits defined this way may be used in the instructions `SBIT0, SBIT1, CMPBIT, JBIT0`, and `JBIT`.

3.1.7 PORT

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

`PORT` works similar to `EQU`, just the symbol becomes assigned to the I/O-address range. Allowed values are 0..7 for the 3201x, 0..15 for the 320C2x, 0..65535 for the 8086 and 320C5x, 0..63 for the AVR, and 0..255 for the rest.

Example : an 8255 PIO is located at address 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 and NAMEREG

*valid for: AVR, M*Core, ST9, 80C16x, KCPSM (*`NAMEREG` *valid only for KCPSM(3)), LatticeMico8*

Though it always has the same syntax, this instruction has a slightly different meaning from processor to processor: If the processor uses a separate addressing space for registers, `REG` has the same effect as a simple `EQU` for this address space (e.g. for the ST9). `REG` defines register symbols for all other processors whose function is described in section 2.12.

`NAMEREG` exists for compatibility reasons to the original KCPSM assembler. It has an identical function, however both register and symbolic name are given as arguments, for example:

```
    NAMEREG  s08, treg
```

3.1.9 LIV and RIV

valid for: 8X30x

`LIV` and `RIV` allow to define so-called "IV bus objects". These are groups of bits located in a peripheral memory cell with a length of 1 up to 8 bits, which can afterwards be referenced symbolically. The result is that one does

not anymore have to specify address, position, and length separately for instructions that can refer to peripheral bit groups. As the 8X30x processors feature two peripheral address spaces (a "left" and a "right" one), there are two separate pseudo instructions. The parameters of these instructions are however equal: three parameters have to be given that specify address, start position and length. Further hints for the usage of bus objects can be found in section 4.17 .

3.1.10 CHARSET

valid for: all processors

Single board systems, especially when driving LCDs, frequently use character sets different to ASCII. So it is probably purely coincidental that the umlaut coding corresponds with the one used by the PC. To avoid error-prone manual encoding, the assembler contains a translation table for characters which assigns a target character to each source-code. To modify this table (which initial translates 1:1), one has to use the `CHARSET` instruction. `CHARSET` may be used with different numbers and types of parameters. If there is only a single parameter, it has to be a string expression which is interpreted as a file name by AS. AS reads the first 256 bytes from this table and copies them into the translation table. This allows to activate complex, externally generated tables with a single statement. For all other variants, the first parameter has to be an integer in the range of 0 to 255 which designates the start index of the entries to be modified in the translation table. One or two parameters follow, giving the type of modification:

A single additional integer modies exactly one entry. For example,

```
CHARSET 'ä',128
```

means that the target system codes the 'ä' into the number 128 (80H). If however two more integers are given, the first one describes the last entry to be modified, and the second the new value of the first table entry. All entries up to the index end are loaded sequentially. For example, in case that the target system does not support lower-case characters, a simple

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

translates all lower-case characters automatically into the matching capital letters.

For the last variant, a string follows the start index and contains the characters to be placed in the table. The last example therefore may also be written as

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

CHARSET may also be called without any parameters, which however has a drastical effect: the translation table is reinitialized to its initial state, i.e. all character translations are removed.

CAUTION! CHARSET not only affects string constants stored in memory, but also integer constants written as "ASCII". This means that an already modified translation table can lead to other results in the above mentioned examples!

3.1.11 CODEPAGE

valid for: all processors

Though the CHARSET statement gives unlimited freedom in the character assignment between host and target platform, switching among different character *sets* can become quite tedious if several character sets have to be supported on the target platform. The CODEPAGE instruction however allows to define and keep different character sets and to switch with a single statement among them. CODEPAGE expects one or two arguments: the name of the set to be used hereafter and optionally the name of another table that defines its initial contents (the second parameter therefore only has a meaning for the first switch to the table when AS automatically creates it). If the second parameter is missing, the initial contents of the new table are copied from the previously active set. All subsequent CHARSET statements *only* modify the new set.

At the beginning of a pass, AS automatically creates a single character set with the name STANDARD with a one-to-one translation. If no CODEPAGE instructions are used, all settings made via CHARSET refer to this table.

3.1.12 ENUM

valid for: all processors

Similar to the same-named instruction known from C, `ENUM` is used to define enumeration types, i.e. a sequence of integer constants that are assigned sequential values starting at 0. The parameters are the names of the symbols, like in the following example:

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

This instruction will assign the values 0, 1, and 2 to the symbols `SymA`, `SymB`, and `SymC`.

`ENUM` instructions are always single-line instructions, i.e. the enumeration will again start at zero when a new `ENUM` instruction is found. Multi-line enumerations may however be achieved with a small trick that exploits the fact that the internal counter can be set to a new value with an explicit assignment, like in the following case:

```
ENUM    January=1,February,March,April,May,June
```

The numeric values 1..6 are assigned to month names. One can continue the enumeration in the following way:

```
ENUM    July=June+1,August,September,October
ENUM    November=October+1,December
```

A definition of a symbol with `ENUM` is equal to a definition with `EQU`, i.e. it is not possible to assign a new value to a symbol that already exists.

3.1.13 PUSHV and POPV

valid for: all processors

`PUSHV` and `POPV` allow to temporarily save the value of a symbol (that is not macro-local) and to restore it at a later point of time. The storage is done on stacks, i.e. Last-In-First-Out memory structures. A stack has a name that has to fulfill the general rules for symbol names and it exists as long as it contains at least one element: a stack that did not exist before is

automatically created upon PUSHV, and a stack becoming empty upon a POPV is deleted automatically. The name of the stack that shall be used to save or restore symbols is the first parameter of PUSH resp. POPV, followed by a list of symbols as further parameters. All symbols referenced in the list already have to exist, it is therefore **not** possible to implicitly define symbols with a POPV instruction.

Stacks are a global resource, i.e. their names are not local to sections.

It is important to note that symbol lists are **always** processed from left to right. Someone who wants to pop several variables from a stack with a POPV therefore has to use the exact reverse order used in the corresponding PUSHV!

The name of the stack may be left blank, like this:

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

AS will then use a predefined internal default stack.

AS checks at the end of a pass if there are stacks that are not empty and issues their names together with their "filling level". This allows to find out if there are any unpaired PUSHVs or POPVs. However, it is in no case possible to save values in a stack beyond the end of a pass: all stacks are cleared at the beginning of a pass!

3.2 Code Modification

3.2.1 ORG

valid for: all processors

ORG allows to load the internal address counter (of the assembler) with a new value. The value range depends on the currently selected segment and on the processor type (tables 3.1 to 3.4). The lower bound is always zero, and the upper bound is the given value minus 1:

target	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	—	—	—	—	—	—	—	—

Table 3.1: Address Ranges for `ORG` — Part 1

target	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	—	—	—	—	—	—	—
* As the 8051 does not have any RAM beyond 80h, this value has to be adapted with ORG for the 8051 as target processor!!									
+ As the Z180 still can address only 64K logically, the whole address space can only be reached via `PHASE` instructions!									

Table 3.2: Address Ranges for `ORG` — Part 2

target	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	—	—	—	—	—	—	—	—

Table 3.3: Address Ranges for `ORG` — Part 3

target	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	—	—	—	—	—	—	—	—
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	—	—	—	—	—	—	—	—

Table 3.4: Address Ranges for `ORG` — Part 4

In case that different variations in a processor family have address spaces of different size, the maximum range is listed for each.

`ORG` is mostly needed to give the code a new starting address or to put different, non-continuous code parts into one source file. In case there is no explicit other value listet in a table entry, the initial address for this segment (i.e. the start address used without `ORG`) is 0.

3.2.2 CPU

valid for: all processors

This command rules for which processor the further code shall be generated. Instructions of other processor families are not accessible afterwards and will produce error messages!

The processors can roughly be distinguished in families, inside the families different types additionally serve for a detailed distinction:

a) 68008 → 68000 → 68010 → 68012 →
MCF5200 → 68332 → 68340 → 68360 →
68020 → 68030 → 68040

The differences in this family lie in additional instructions and addressing modes (starting from the 68020). A small exception is the step to the 68030 that misses two instructions: `CALLM` and `RTM`. The three representors of the 683xx family have the same processor core (a slightly reduced 68020 CPU), however completely different peripherals. MCF5200 represents the ColdFire family from Motorola, RISC processors downwardly binary compatible to the 680x0. For the 68040, additional control registers (reachable via `MOVEC`) and instructions for control of the on-chip MMU and caches were added.

b) 56000 ⟶ 56002 ⟶ 56300

While the 56002 only adds instructions for incrementing and decrementing the accumulators, the 56300 core is almost a new processor: all address spaces are enlarged from 64K words to 16M and the number of instructions almost has been doubled.

c) PPC403 → MPPC403 → MPC505 → MPC601 → RS6000

The PPC403 is a reduced version of the PowerPC line without a floating point unit, which is why all floating point instructions are disabled for him; in turn, some microcontroller-specific instructions have been added which are unique in this family. The GC variant of the PPC403 incorporates an additional MMU and has therefore some additional instructions for its control. The MPC505 (a microcontroller variant without a FPU) only differ in its peripheral registers from the 601 as long as I do not know it better - [58] is a bit reluctant in this respect... The RS6000 line knows a few instructions more (that are emulated on many 601-based systems), IBM additionally uses different mnemonics for their pure workstation processors, as a reminiscence of 370 mainframes...

d) MCORE

e) XGATE

f) 6800 → 6301 → 6811

While the 6301 only offers a few additional instructions, the 6811 delivers a second index register and much more instructions.

g) 6809/6309 and 6805/68HC(S)08

These processors are partially source-code compatible to the other 68xx processors, but they have a different binary code format and a significantly reduced (6805) resp. enhanced (6809) instruction set. The 6309 is a CMOS version of the 6809 which is officially only compatible to the 6809, but inofficially offers more registers and a lot of new instructions (see [37]).

h) 68HC12 ⟶ 68HC12X

The 12X core offers a couple of new instructions, and existing instructions were were enriched with new addressing modes.

i) 68HC16

j) HD6413308 → HD6413309

These both names represent the 300 and 300H variants of the H8 family; the H version owns a larger address space (16Mbytes instead of 64Kbytes), double-width registers (32 bits), and knows a few more instructions and addressing modes. It is still binary upward compatible.

k) HD6475328 → HD6475348 → HD6475368 → HD6475388

These processors all share the same CPU core; the different types are only needed to include the correct subset of registers in the file `REG53X.INC`.

l) SH7000 → SH7600 ⟶ SH7700

The processor core of the 7600 offers a few more instructions that close gaps in the 7000's instruction set (delayed conditional and relative and indirect jumps, multiplications with 32-bit operands and multiply/add instructions). The 7700 series (also known as SH3) furthermore offers a second register bank, better shift instructions, and instructions to control the cache.

m) 6502 → 65(S)C02 / MELPS740 / 6502UNDOC

The CMOS version defines some additional instructions, as well as a number of some instruction/addressing mode combinations were added which were not possible on the 6502. The Mitsubishi micro controllers in opposite expand the 6502 instruction set primarily to bit operations and multiplication / division instructions. Except for the unconditional jump and instructions to increment/decrement the accumulator, the instruction extensions are orthogonal. The 65SC02 lacks the bit manipulation instructions of the 65C02. The 6502UNDOC processor type enables access to the "undocumented" 6502 instructions, i.e. the operations that result from the usage of bit combinations in the opcode that are not defined as instructions. The variants supported by AS are listed in the appendix containing processor-specific hints.

n) MELPS7700, 65816

Apart from a '16-bit-version' of the 6502's instruction set, these processors both offer some instruction set extensions. These are however orthogonal as they are oriented along their 8-bit predecessors (65C02 resp. MELPS-740). Partially, different mnemonics are used for the same operations.

o) MELPS4500

p) M16

q) M16C

r) 4004 → 4040

Opposed to its predecessor, the 4040 features about a dozen additional machine instructions.

s) 8008 → 8008NEW Intel redefined the mnemonics around 1975, the second variant reflects this new instruction set. A simultaneous support of both sets was not possible due to mnemonic conflicts.

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

For the ROM-less versions 8039 and 80C39, the commands which are using the BUS (port 0) are forbidden. The 8021 and 8022 are special versions with a strongly shrinked instruction set, for which the 8022 has two A/D-converters and the necessary control-commands. It is possible to transfer the CMOS-versions with the `IDL`-command into a stop mode with lower current consumption. The 8041 and 8042 have some additional instructions for controlling the bus interface, but in turn a few other commands were omitted. Moreover, the code address space of these processors is not externally extendable, and so AS limits the code segment of these processors to 1 resp. 2 Kbytes.

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

The 87C750 can only access a maximum of 2 Kbytes program memory which is why it lacks the `LCALL` and `LJMP` instructions. AS does not make any distinction among the processors in the middle, instead it only stores the different names in the `MOMCPU` variable (see below), which allows to query the setting with `IF` instructions. An exception is the 80C504 that has a mask flaw in its current versions. This flaw shows up when an `AJMP` or `ACALL` instruction starts at the second last address of a 2K page. AS will automatically use long instructions or issues an error message in such situations. The 80C251 in contrast represents a drastic progress in the the direction 16/32 bits, larger address spaces, and a more orthogonal instruction set. One might call the 80C390 the 'small solution': Dallas Semiconductor modified instruction set and architecture only as far as it was necessary for the 16 Mbytes large address spaces.

v) 8096 → 80196 → 80196N → 80296

Apart from a different set of SFRs (which however strongly vary from version to version), the 80196 knows several new instructions and supports a 'windowing' mechanism to access the larger internal RAM. The 80196N family extends the address space to 16 Mbytes and introduces a set of instructions to access addresses beyond 64Kbytes. The 80296 extends the CPU core by instructions for signal processing and a second windowing register, however removes the Peripheral Transaction Server (PTS) and therefore looses again two machine instructions.

w) 8080 → 8085 → 8085UNDOC

The 8085 knows the additional commands `RIM` and `SIM` for controlling the interrupt mask and the two I/O-pins. The type `8085UNDOC` enables additional instructions that are not documented by Intel. These instructions are documented in section 4.15.

x) 8086 → 80186 → V30 → V35

Only new instructions are added in this family. The corresponding 8-bit versions are not mentioned due to their instruction compatibility, so one e.g. has to choose 8086 for an 8088-based system.

y) 80960

z) 8X300 → 8X305

The 8X305 features a couple of additional registers that miss on the 8X300. Additionally, it can do new operations with these registers (like direct writing of 8 bit values to peripheral addresses).

aa) XAG1, XAG2, XAG3

These processors only differ in the size of their internal ROM which is defined in `STDDEFXA.INC`.

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

The first member of the AVR series represents a minimum configuration without RAM memory and therefore lacks load/store instructions. The next three processors only differ in their memory equipment and on-chip peripherals, what is differentiated in `REGAVR.INC`. The same is true for the MEGA AVRs, which also offer new machine instructions compared to their predecessors.

ac) AM29245 → AM29243 → AM29240 → AM29000

The further one moves to the right in this list, the fewer the instructions become that have to be emulated in software. While e.g. the 29245 not even owns a hardware multiplier, the two representors in the middle only lack the floating point instructions. The 29000 serves as a 'generic' type that understands all instructions in hardware.

ad) 80C166 → 80C167,80C165,80C163

80C167 and 80C165/163 have an address space of 16 Mbytes instead of 256 Kbytes, and furthermore they know some additional instructions for extended addressing modes and atomic instruction sequences. They are 'second generation' processors and differ from each other only in the amount of on-chip peripherals.

ae) Z80 → Z80UNDOC → Z180 → Z380

While there are only a few additional instructions for the Z180, the Z380 owns 32-bit registers, a linear address space of 4 Gbytes, a couple of instruction set extensions that make the overall instruction set considerably more orthogonal, and new addressing modes (referring to index register halves, stack relative). These extensions partially already exist on the Z80 as undocumented extensions and may be switched on via the Z80UNDOC variant. A list with the additional instructions can be found in the chapter with processor specific hints.

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

The variants with Z8 core only differ in internal memory size and on-chip peripherals, i.e. the choice does not have an effect on the supported instruction set. This is substantially different with the eZ8, which brings along a strongly extended instruction set that is in wide parts only source-level compatible.

ag) KCPSM

Both processor cores are not available as standalone components, they are provided as logic cores for gate arrays made by Xilinx The -3 variant offers a larger address space and some additional instructions. Note that it is not binary upward-compatible!

ah) MICO8_05, MICO8_V3, MICO8_V31

Lattice unfortunately changed the machine instructions more than once, so different targets became necessary to provide continued support for older projects. The first variant is the one described in the 2005 manual, the two other ones represent versions 3.0 resp. 3.1.

ai) 96C141, 93C141

These two processors represent the two variations of the processor family: TLCS-900 and TLCS-900L. The differences of these two variations will be discussed in detail in section 4.23.

aj) 90C141

ak) 87C00, 87C20, 87C40, 87C70

The processors of the TLCS-870 series have an identical CPU core, but different peripherals depending on the type. In part registers with the same name are located at different addresses. The file `STDDEF87.INC` uses, similar to the MCS-51-family, the distinction possible by different types to provide the correct symbol set automatically.

al) 47C00 → 470C00 → 470AC00

These three variations of the TLCS-47-family have on-chip RAM and ROM of different size, which leads to several bank switching instructions being added or suppressed.

am) 97C241

an) 16C54 → 16C55 → 16C56 → 16C57

These processors differ by the available code area, i.e. by the address limit after which AS reports overruns.

ao) 16C84, 16C64

Analog to the MCS-51 family, no distinction is made in the code generator, the different numbers only serve to include the correct SFRs in `STDDEF18.INC`.

ap) 17C42

aq) ST6210/ST6215→ST6220/ST6225

The only distinction AS makes between the two pairs is the smaller addressing space (2K instead 4K) of the first ones. The detailed distinction serves to provide an automatic distinction in the source file which hardware is available (analog to the 8051/52/515).

ar) ST7

as) ST9020, ST9030, ST9040, ST9050

These 4 names represent the four "sub-families" of the ST9 family, which only differ in their on-chip peripherals. Their processor cores are identical, which is why this distinction is again only used in the include file containing the peripheral addresses.

at) 6804

au) 32010→32015

The TMS32010 owns just 144 bytes of internal RAM, and so AS limits addresses in the data segment just up to this amount. This restriction does not apply for the 32015, the full range from 0..255 can be used.

av) 320C25 → 320C26 → 320C28

These processors only differ slightly in their on-chip peripherals and in their configuration instructions.

aw) 320C30, 320C31

The 320C31 is a reduced version with the same instruction set, however fewer peripherals. The distinction is exploited in `STDDEF3X.INC`.

ax) 320C203 → 320C50, 320C51, 320C53

The first one represents the C20x family of signal processors which implement a subset of the C5x instruction set. The distinction among the C5x processors is currently not used by AS.

ay) 320C541

This one at the moment represents the TMS320C54x family...

az) TMS9900

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

All members of this family share the same CPU core, they therefore do not differ in their instruction set. The differences manifest only in the file `REG7000.INC` where address ranges and peripheral addresses are defined. Types listed in the same row have the same amount of internal RAM and the same on-chip peripherals, they differ only in the amount of integrated ROM.

bb) 370C010, 370C020, 370C030, 370C040 and 370C050

Similar to the MCS-51 family, the different types are only used to differentiate the peripheral equipment in `STDDEF37.INC`; the instruction set is always the same.

bc) MSP430

bd) SC/MP

be) 8070

This processor represents the whole 807x family (which consists at least of the 8070, 8072, and 8073), which however shares identical CPU cores.

bf) COP87L84

This is the only member of National Semiconductor's COP8 family that is currently supported. I know that the family is substantially larger and that there are representors with differently large instruction sets which will be added when a need occurs. It is a beginning, and National's documentation is quite extensive...

bg) COP410 → COP420 The COP42x derivates offer some additional instructions, plus other instructions have an extended operand range.

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

This series of DECT controllers differentiates itself by the amount of instructions, since each of them supports different B field formats and their architecture has been optimized over time.

bi) 7810→78C10

The NMOS version has no stop-mode; the respective command and the ZCM register are omitted. **CAUTION!** NMOS and CMOS version partially differ in the reset values of some registers!

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

This 'cornucopia' of processors differs only by the RAM size in one group; the groups themselves again differ by their on-chip peripherals on the one hand and by their instruction set's power on the other hand.

bk) 78070

This is currently the only member of NEC's 78K0 family I am familiar with. Similar remarks like for the COP8 family apply!

bl) 78214

This is currently the the representor of NEC's 78K2 family.

bm) 7720 → 7725

The μPD7725 offers larger address spaces and som more instructions compared to his predecessor. **CAUTION!** The processors are not binary compatible to each other!

bn) 77230

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

The simpler members of this family of SCSI processors lack some instruction variants, furthermore they are different in their set of internal registers.

bp) MB89190

This processor type represents Fujitsu's F^{2}MC8L series...

bq) MB9500

...just like this one does it currently for the 16-bit variants from Fujitsu!

The `CPU` instruction needs the processor type as a simple constant, a calculation like:

```
        CPU     68010+10
```

is not allowed. Valid calls are e.g.

```
        CPU     8051
```

or

```
        CPU     6800
```

Regardless of the processor type currently set, the integer variable `MOMCPU` contains the current status as a hexadecimal number. For example, `MOMCPU`=$68010 for the 68010 or `MOMCPU`=80C48H for the 80C48. As one cannot express all letters as hexadecimal digits (only A..F are possible), all other letters must must be omitted in the hex notation; for example, `MOMCPU`=80H for the Z80.

You can take advantage of this feature to generate different code depending on the processor type. For example, the 68000 does not have a machine instruction for a subroutine return with stack correction. With the variable `MOMCPU` you can define a macro that uses the machine instruction or emulates it depending on the processor type:

```
myrtd   macro   disp
        if      MOMCPU<$68010 ; emulate for 68008 & 68000
         move.l (sp),disp(sp)
         lea    disp(sp),sp
         rts
        elseif
```

```
 rtd    #disp          ; direct use on >=68010
endif
endm


cpu     68010
myrtd   12             ; results in RTD #12

cpu     68000
myrtd   12             ; results in MOVE../LEA../RTS
```

As not all processor names are built only out of numbers and letters from A..F, the full name is additionally stored in the string variable named `MOMCPUNAME`.

The assembler implicitly switches back to the `CODE` segment when a `CPU` instruction is executed. This is done because `CODE` is the only segment all processors support.

The default processor type is 68008, unless it has been changed via the command line option with same name.

3.2.3 SUPMODE, FPU, PMMU

*valid for: 680x0, FPU also for 80x86, i960, SUPMODE also for TLCS-900, SH7000, i960, 29K, XA, PowerPC, M*Core, and TMS9900*

These three switches allow to define which parts of the instruction set shall be disabled because the necessary preconditions are not valid for the following piece of code. The parameter for these instructions may be either `ON` or `OFF`, the current status can be read out of a variable which is either TRUE or FALSE.

The commands have the following meanings in detail:

- `SUPMODE`: allows or prohibits commands, for whose execution the processor has to be within the supervisor mode. The status variable is called `INSUPMODE`.

- `FPU`: allows or prohibits the commands of the numerical coprocessors 8087 resp. 68881 or 68882. The status variable is called `FPUAVAIL`.
- `PMMU`: allows or prohibits the commands of the memory management unit 68851 resp. of the built-in MMU of the 68030. **CAUTION!** The 68030-MMU supports only a relatively small subset of the 68851 instructions. The assembler cannot test this! The status variable is called `PMMUAVAIL`.

The usage of of instructions prohibited in this manner will generate a warning at `SUPMODE`, at `PMMU` and `FPU` a real error message.

3.2.4 FULLPMMU

valid for: 680x0

Motorola integrated the MMU into the processor starting with the 68030, but the built-in FPU is equipped only with a relatively small subset of the 68851 instruction set. AS will therefore disable all extended MMU instructions when the target processor is 68030 or higher. It is however possible that the internal MMU has been disabled in a 68030-based system and the processor operates with an external 68851. One can the use a `FULLPMMU ON` to tell AS that the complete MMU instruction set is allowed. Vice versa, one may use a `FULLPMMU OFF` to disable all additional instruction in spite of a 68020 target platform to assure that portable code is written. The switch between full and reduced instruction set may be done as often as needed, and the current setting may be read from a symbol with the same name. **CAUTION!** The `CPU` instruction implicitly sets or resets this switch when its argument is a 68xxx processor! `FULLPMMU` therefore has to be written after the `CPU` instruction!

3.2.5 PADDING

*valid for: 680x0, M*Core, XA, H8, SH7000, MSP430, TMS9900, ST7*

Processors of the 680x0 family are quite critical regarding odd addresses: instructions must not start on an odd address, and data accesses to odd

addresses are only allowed bytewise up to the 68010. The H8/300 family simply resets the lowest address bit to zero when accessing odd addresses, the 500 in contrast 'thanks' with an exception... AS therefore tries to round up data structures built with `DC` or `DS` to an even number of bytes. This however means for `DC.B` and `DS.B` that a padding byte may have to be added. This behaviour can be turned on and off via the `PADDING` instruction. Similar to the previous instructions, the argument may be either `ON` or `OFF`, and the current setting may be read from a symbol with the same name. `PADDING` is by default only enabled for the 680x0 family, it has to be turned on explicitly for all other families!

3.2.6 PACKING

valid for: AVR

In some way, `PACKING` is similar to `PADDING`, it just has a somewhat opposite effect: While `PADDING` extends the disposed data to get full words and keep a possible alignment, `PACKING` squeezes several values into a single word. This makes sense for the AVR's code segment since the CPU has a special instruction (`LPM`) to access single bytes within a 16-bit word. In case this option is turned on (argument `ON`), two byte values are packed into a single word by `DATA`, similar to the single characters of string arguments. The value range of course reduces to -128...+255. If this option is turned off (argument `OFF`), each integer argument obtains its own word and may take values from -32768...+65535.

This distinctin is only made for integer arguments of `DATA`, strings will always be packed.. Keep further in mind that packing of values only works within the arguments of a `DATA` statement; if one has subsequent `DATA` statements, there will still be half-filled words when the argument count is odd!

3.2.7 MAXMODE

valid for: TLCS-900, H8

The processors of the TLCS-900-family are able to work in 2 modes, the minimum and maximum mode. Depending on the actual mode, the execution

environment and the assembler are a little bit different. Along with this instruction and the parameter `ON` or `OFF`, AS is informed that the following code will run in maximum resp. minimum mode. The actual setting can be read from the variable `INMAXMODE`. Presetting is `OFF`, i.e. minimum mode.

Similarly, one uses this instruction to tell AS in H8 mode whether the address space is 64K or 16 Mbytes. This setting is always `OFF` for the 'small' 300 version and cannot be changed.

3.2.8 EXTMODE and LWORDMODE

valid for: Z380

The Z380 may operate in altogether 4 modes, which are the result of setting two flags: The XM flag rules whether the processor shall operate wit an address space of 64 Kbytes or 4 Gbytes and it may only be set to 1 (after a reset, it is set to 0 for compatibility with the Z80). The LW flag in turn rules whether word operations shall work with a word size of 16 or 32 bits. The setting of these two flags influences range checks of constants and addresses, which is why one has to tell AS the setting of these two flags via these instructions. The default assumption is that both flags are 0, the current setting (`ON` or `OFF`) may be read from the predefined symbols `INEXTMODE` resp. `INLWORDMODE`.

3.2.9 SRCMODE

valid for: MCS-251

Intel substantially extended the 8051 instruction set with the 80C251, but unfortunately there was only a single free opcode for all these new instructions. To avoid a processor that will be eternally crippled by a prefix, Intel provided two operating modes: the binary and the source mode. The new processor is fully binary compatible to the 8051 in binary mode, all new instructions require the free opcode as prefix. In source mode, the new instructions exchange their places in the code tables with the corresponding 8051 instructions, which in turn then need a prefix. One has to inform AS whether the processor operates in source mode (`ON`) or binary mode (`OFF`) to enable AS to add prefixes when required. The current setting may be read from the variable `INSRCMODE`. The default is `OFF`.

3.2.10 BIGENDIAN

valid for: MCS-51/251, PowerPC

Intel broke with its own principles when the 8051 series was designed: in contrast to all traditions, the processor uses big-endian ordering for all multi-byte values! While this was not a big deal for MCS-51 processors (the processor could access memory only in 8-bit portions, so everyone was free to use whichever endianess one wanted), it may be a problem for the 251 as it can fetch whole (long-)words from memory and expects the MSB to be first. As this is not the way of constant disposal earlier versions of AS used, one can use this instruction to toggle between big and little endian mode for the instructions `DB, DW, DD, DQ,` and `DT`. `BIGENDIAN OFF` (the default) puts the LSB first into memory as it used to be on earlier versions of AS, `BIGENDIAN ON` engages the big-endian mode compatible to the MCS-251. One may of course change this setting as often as one wants; the current setting can be read from the symbol with the same name.

3.2.11 WRAPMODE

valid for: Atmel AVR

After this switch has been set to `ON`, AS will assume that the processor's program counter does not have the full length of 16 bits given by the architecture, but instead a length that is exactly sufficient to address the internal ROM. For example, in case of the AT90S8515, this means 12 bits, corresponding to 4 Kwords or 8 Kbytes. This assumption allows relative branches from the ROM's beginning to the end and vice versa which would result in an out-of-branch error when using strict arithmetics. Here, they work because the carry bits resulting from the target address computation are discarded. Assure that the target processor you are using works in the outlined way before you enable this option! In case of the abovementioned AT90S8515, this option is even necessary because it is the only way to perform a direct jump through the complete address space...

This switch is set to `OFF` by default, and its current setting may be read from a symbol with same name.

3.2.12 SEGMENT

valid for: all processors

Some microcontrollers and signal processors know various address ranges, which do not overlap with each other and require also different instructions and addressing modes for access. To manage these ones also, the assembler provides various program counters, you can switch among them to and from by the use of the `SEGMENT` instruction. For subroutines included with `INCLUDE`, this e.g. allows to define data used by the main program or subroutines near to the place they are used. In detail, the following segments with the following names are supported:

- `CODE`: program code;
- `DATA`: directly addressable data (including SFRs);
- `XDATA`: data in externally connected RAM or X-addressing space of the DSP56xxx or ROM data for the μPD772x;
- `YDATA`: Y-addressing space of the DSP56xxx;
- `IDATA`: indirectly addressable (internal) data;
- `BITDATA`: the part of the 8051-internal RAM that is bitwise addressable;
- `IO`: I/O-address range;
- `REG`: register bank of the ST9;
- `ROMDATA`: constant ROM of the NEC signal processors.

See also section 3.2.1 (`ORG`) for detailed information about address ranges and initial values of the segments. Depending on the processor family, not all segment types will be permitted.

The bit segment is managed as if it would be a byte segment, i.e. the addresses will be incremented by 1 per bit.

Labels get the same type as attribute as the segment that was active when the label was defined. So the assembler has a limited ability to check whether you access symbols of a certain segment with wrong instructions. In such cases the assembler issues a warning.

Example:

```
        CPU     8051    ; MCS-51-code

        segment code    ; test code

        setb    flag    ; no warning
        setb    var     ; warning : wrong segment

        segment data

var     db      ?

        segment bitdata

flag    db      ?
```

3.2.13 PHASE and DEPHASE

valid for: all processors

For some applications (especially on Z80 systems), the code must be moved to another address range before execution. If the assembler didn't know about this, it would align all labels to the load address (not the start address). The programmer is then forced to write jumps within this area either independent of location or has to add the offset at each symbol manually. The first one is not possible for some processors, the last one is extremely error-prone. With the commands `PHASE` and `DEPHASE`, it is possible to inform the assembler at which address the code will really be executed on the target system:

```
        phase   <address>
```

informs the assembler that the following code shall be executed at the specified address. The assembler calculates thereupon the difference to the real program counter and adds this difference for the following operations:

- address values in the listing
- filing of label values

- program counter references in relative jumps and address expressions
- readout of the program counter via the symbols * or $

this "shifting" is switched off by the instruction

```
dephase
```

The assembler manages phase values for all defined segments, although this instruction pair only makes real sense in the code segment.

3.2.14 SAVE and RESTORE

valid for: all processors

The command `SAVE` forces the assembler to push the contents of following variables onto an internal stack:

- currently selected processor type (set by `CPU`);
- currently active memory area (set by `SEGMENT`);
- the flag whether listing is switched on or off (set by `LISTING`);
- the flag whether expansions of following macros shall be issued in the assembly listing (set by `MACEXP`).
- currently active character translation table (set by `CODEPAGE`).

The counterpart `RESTORE` pops the values saved last from this stack. These two commands were primarily designed for include files, to change the above mentioned variables in any way inside of these files, without loosing their original content. This may be helpful e.g. in include files with own, fully debugged subroutines, to switch the listing generation off:

```
        SAVE            ; save old status

        LISTING OFF     ; save paper

        .               ; the actual code
        .

        RESTORE         ; restore
```

In opposite to a simple `LISTING OFF .. ON`-pair, the correct status will be restored, in case the listing generation was switched off already before.

The assembler checks if the number of `SAVE`-and `RESTORE`-commands corresponds and issues error messages in the following cases:

- `RESTORE`, but the internal stack is empty;
- the stack not empty at the end of a pass.

3.2.15 ASSUME

valid for: various

This instruction allows to tell AS the current setting of certain registers whose contents cannot be described with a simple `ON` or `OFF`. These are typically registers that influence addressing modes and whose contents are important to know for AS in order to generate correct addressing. It is important to note that `ASSUME` only informs AS about these, **no** machine code is generated that actually loads these values into the appropriate registers!

6809

In contrast to its 'predecessors' like the 6800 and 6502, the position of the direct page, i.e. the page of memory that can be reached with single-byte addresses, can be set freely. This is done via the 'direct page register' that sets the page number. One has to assign a corresponding value to this register via `ASSUME` is the contents are different from the default of 0, otherwise wrong addresses will be generated!

68HC11K4

Also for the HC11, the designers finally weren't able to avoid the major sin: using a banking scheme to address more than 64 Kbytes with only 16 address lines. The registers `MMSIZ`, `MMWBR`, `MM1CR`, and `MM2CR` control whether and how the additional 512K address ranges are mapped into the physical address space. AS intially assumes the reset state of these registers, i.e. all are set to $00 and windowing is disabled.

68HC12X

Similar to its cousin without the appended 'X', the HC12X supports a short direct addressing mode. In this case however, it can be used to address more than just the first 256 bytes of the address space. The `DIRECT` register specifices which 256 byte page of the address space is addressed by this addressing mode. `ASSUME` is used to tell AS the current value of this register, so it is able to automatically select the most efficient address ing mode when absolute addresses are used. The default is 0, which corresponds to the reset state.

68HC16

The 68HC16 employs a set of bank registers to address a space of 1 Mbyte with its registers that are only 16 bits wide. These registers supply the upper 4 bits. Of these, the EK register is responsible for absolute data accesses (not jumps!). AS checks for each absolute address whether the upper 4 bits of the address are equal to the value of EK specified via `ASSUME`. AS issues a warning if they differ. The default for EK is 0.

H8/500

In maximum mode, the extended address space of these processors is addressed via a couple of bank registers. They carry the names DP (registers from 0..3, absolute addresses), EP (register 4 and 5), and TP (stack). AS needs the current value of DP to check if absolute addresses are within the

currently addressable bank; the other two registers are only used for indirect addressing and can therefore not be monitored; it is a question of personal taste whether one specifies their values or not. The BR register is in contrast important because it rules which 256-byte page may be accessed with short addresses. It is common for all registers that AS does not assume **any** default value for them as they are undefined after a CPU reset. Everyone who wants to use absolute addresses must therefore assign values to at least DR and DP!

MELPS740

Microcontrollers of this series know a "special page" addressing mode for the `JSR` instruction that allows a shorter coding for jumps into the last page of on-chip ROM. The size of this ROM depends of course on the exact processor type, and there are more derivatives than it would be meaningful to offer via the CPU instruction...we therefore have to rely on `ASSUME` to define the address of this page, e.g.

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

in case the internal ROM is 8K.

MELPS7700/65816

These processors contain a lot of registers whose contents AS has to know in order to generate correct machine code. These are the registers in question:

name	function	value range	default
DT	data bank	0-$ff	0
PG	code Bank	0-$ff	0
DPR	directly addr. page	0-$ffff	0
X	index register width	0 or 1	0
M	accumulator width	0 or 1	0

To avoid endless repetitions, see section 4.9 for instructions how to use these registers. The handling is otherwise similar to the 8086, i.e. multiple values may be set with one instruction and no code is generated that actually loads the registers with the given values. This is again up to the programmer!

MCS-196/296

Starting with the 80196, all processors of the MCS-96 family have a register 'WSR' that allows to map memory areas from the extended internal RAM or the SFR range into areas of the register file which may then be accessed with short addresses. If one informs AS about the value of the WSR register, it can automatically find out whether an absolute address can be addressed with a single-byte address via windowing; consequently, long addresses will be automatically generated for registers covered by windowing. The 80296 contains an additional register WSR1 to allow simultaneous mapping of two memory areas into the register file. In case it is possible to address a memory cell via both areas, AS will always choose the way via WSR!

8086

The 8086 is able to address data from all segments in all instructions, but it however needs so-called "segment prefixes" if another segment register than DS shall be used. In addition it is possible that the DS register is adjusted to another segment, e.g. to address data in the code segment for longer parts of the program. As AS cannot analyze the code's meaning, it has to informed via this instruction to what segments the segment registers point at the moment, e.g.:

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

It is possible to assign assumptions to all four segment registers in this way. This instruction produces **no** code, so the program itself has to do the actual load of the registers with the values.

The usage of this instruction has on the one hand the result that AS is able to automatically put ahead prefixes at sporadic accesses into the code segment, or on the other hand, one can inform AS that the DS-register was modified and you can save explicit `CS:`-instructions.

Valid arguments behind the colon are `CODE`, `DATA` and `NOTHING`. The latter value informs AS that a segment register contains no usable value (for AS). The following values are preinitialized:

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

XA

The XA family has a data address space of 16 Mbytes, a process however can always address within a 64K segment only that is given by the DS register. One has to inform AS about the current value of this register in order to enable it to check accesses to absolute addresses.

29K

The processors of the 29K family feature a register RBP that allows to protect banks of 16 registers against access from user mode. The corresponding bit has to be set to achieve the protection. `ASSUME` allows to tell AS which value RBP currently contains. AS can warn this way in case a try to access protected registers from user mode is made.

80C166/167

Though none of the 80C166/167's registers is longer than sixteen bits, this processor has 18/24 address lines and can therefore address up to 256Kbytes/16Mbytes. To resolve this contradiction, it neither uses the well-known (and ill-famed) Intel method of segmentation nor does it have inflexible bank registers...no, it uses paging! To accomplish this, the logical address space of 64 Kbytes is split into 4 pages of 16 Kbytes, and for each page there is a page register (named DPP0..DPP3) that rules which of the 16/1024 physical pages shall be mapped to this logical page. AS always tries to present the address space with a size of 256Kbytes/16MBytes in the sight of the programmer, i.e. the physical page is taken for absolute accesses and the setting of bits 14/15 of the logical address is deduced. If no page register fits, a warning is issued. AS assumes by default that the four registers linearly map the first 64 Kbytes of memory, in the following style:

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

The 80C167 knows some additional instructions that can override the page registers' function. The chapter with processor-specific hints describes how these instructions influence the address generation.

TLCS-47

The direct data address space of these processors (it makes no difference whether you address directly or via the HL register) has a size of only 256 nibbles. Because the "better" family members have up to 1024 nibbles of RAM on chip, Toshiba was forced to introduce a banking mechanism via the DMB register. AS manages the data segment as a continuous addressing space and checks at any direct addressing if the address is in the currently active bank. The bank AS currently expects can be set by means of

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

The default value is 0.

ST6

The microcontrollers of the ST62 family are able to map a part (64 bytes) of the code area into the data area, e.g. to load constants from the ROM. This means also that at one moment only one part of the ROM can be addressed. A special register rules which part it is. AS cannot check the contents of this register directly, but it can be informed by this instruction that a new value has been assigned to the register. AS then can test and warn if necessary, in case addresses of the code segment are accessed, which are not located in the "announced" window. If, for example, the variable `VARI` has the value 456h, so

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

sets the AS-internal variable to 11h, and an access to `VARI` generates an access to address 56h in the data segment.

It is possible to assign a simple `NOTHING` instead of a value, e.g. if the bank register is used temporarily as a memory cell. This value is also the default.

ST9

The ST9 family uses exactly the same instructions to address code and data area. It depends on the setting of the flag register's DP flag which address space is referenced. To enable AS to check if one works with symbols from the correct address space (this of course **only** works with absolute accesses!), one has to inform AS whether the DP flag is currently 0 (code) or 1 (data). The initial value of this assumption is 0.

78K2

78K2 is an 8/16 bit architecture, which has later been extended to a one-megabyte addres space via banking. Banking is realized with the registers PM6 (normal case) resp. P6 (alternate case with `&` as prefix) that supply the missing upper four address bits. At least for absolute addresses, AS can check whether the current, linear 20-bit address is within the given 64K window.

320C3x

As all instruction words of this processor family are only 32 bits long (of which only 16 bits were reserved for absolute addresses), the missing upper 8 bits have to be added from the DP register. It is however still possible to specify a full 24-bit address when addressing, AS will check then whether the upper 8 bits are equal to the DP register's assumed values. `ASSUME` is different to the `LDP` instruction in the sense that one cannot specify an arbitrary address out of the bank in question, one has to extract the upper bits by hand:

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

μPD78(C)10

These processors have a register (V) that allows to move the "zero page", i.e. page of memory that is addressable by just one byte, freely in the address space, within page limits. By reasons of comforts you don't want to work with expressions such as

```
inrw    Lo(counter)
```

so AS takes over this job, but only under the premise that it is informed via the `ASSUME`-command about the contents of the V register. If an instruction with short addressing is used, it will be checked if the upper half of the address expression corresponds to the expected content. A warning will be issued if both do not match.

75K0

As the whole address space of 12 bits could not be addressed even by the help of register pairs (8 bits), NEC had to introduce banking (like many others too...): the upper 4 address bits are fetched from the MBS register (which can be assigned values from 0 to 15 by the `ASSUME` instruction), which however will only be regarded if the MBE flag has been set to 1. If it is 0 (default), the lowest and highest 128 nibbles of the address space can be reached without banking. The `ASSUME` instruction is undefined for the 75402 as it contains neither a MBE flag nor an MBS register; the initial values cannot be changed therefore.

F^2MC16L

Similar to many other families of microcontrollers, this family suffers somewhat from its designers miserliness: registers of only 16 bits width are faced with an address space of 24 bits. Once again, bank registers had to fill the gap. In detail, these are PCB for the progam code, DTB for all data accesses, ADB for indirect accesses via RW2/RW6, and SSB/USB for the stacks. They may all take values from 0 to 255 and are by default assumed to be 0, with the exception of 0ffh for PCB.

Furthermore, a DPR register exists that specifies which memory page within the 64K bank given by DTB may be reached with 8 bit addresses. The default for DPR is 1, resulting in a default page of 0001xxh when ine takes DTB's default into account.

3.2.16 EMULATED

valid for: 29K

AMD defined the 29000's series exception handling for undefined instructions in a way that there is a separate exception vector for each instruction. This allows to extend the instruction set of a smaller member of this family by a software emulation. To avoid that AS quarrels about these instructions as being undefined, the `EMULATED` instruction allows to tell AS that certain instructions are allowed in this case. The check if the currently set processors

knows the instruction is then skipped. For example, if one has written a module that supports 32-bit IEEE numbers and the processor does not have a FPU, one writes

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

3.2.17 BRANCHEXT

valid for: XA

`BRANCHEXT` with either `ON` or `OFF` as argument tells AS whether short branches that are only available with an 8-bit displacement shall automatically be 'extended', for example by replacing a single instruction like

```
        bne     target
```

with a longer sequence of same functionality, in case the branc target is out of reach for the instruction's displacement. For example, the replacement sequence for `bne` would be

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

In case there is no fitting 'opposite' for an instruction, the sequence may become even longer, e.g. for `jbc`:

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

This feature however has the side effect that there is no unambigious assignment between machine and assembly code any more. Furthermore, additional passes may be the result if there are forward branches. One should therefore use this feature with caution!

3.3 Data Definitions

The instructions described in this section partially overlap in their functionality, but each processor family defines other names for the same function. To stay compatible with the standard assemblers, this way of implementation was chosen.

If not explicitly mentioned otherwise, all instructions for data deposition (not those for reservation of memory!) allow an arbitrary number of parameters which are being processed from left to right.

3.3.1 DC[.Size]

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

This instruction places one or several constants of the type specified by the attribute into memory. The attributes are the same ones as defined in section 2.5, and there is additionally the possibility for byte constants to place string constants in memory, like

```
String  dc.B "Hello world!\0"
```

The parameter count may be between 1 and 20. A repeat count enclosed in brackets may additionally be prefixed to each parameter; for example, one can for example fill the area up to the next page boundary with zeroes with a statement like

```
        dc.b    [(*+255)&$ffffff00-*]0
```

CAUTION! This function easily allows to reach the limit of 1 Kbyte of generated code per line!

The assembler can automatically add another byte of data in case the byte sum should become odd, to keep the word alignment. This behaviour may be turned on and off via the `PADDING` instruction.

Decimal floating point numbers stored with this instruction (`DC.P...`) can cover the whole range of extended precision, one however has to pay attention to the detail that the coprocessors currently available from Motorola

(68881/68882) ignore the thousands digit of the exponent at the read of such constants!

The default attribute is `W`, that means 16-bit-integer numbers.

For the DSP56xxx, the data type is fixed to integer numbers (an attribute is therefore neither necessary nor allowed), which may be in the range of -8M up to 16M-1. String constants are also allowed, whereby three characters are packed into each word.

Opposed to the standar Motorola ssembler, it is also valid to reserve memory space with this statement, by using a question mark as operand. This is an extension added by some third-party suppliers for 68K assemblers, similar to what Intel assemblers provide. However, it should be clear that usage of this feature may lead to portability problems. Furthermore, question marks as operands must not be mixed with 'normal' constants in a single statement.

3.3.2 DS[.Size]

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

On the one hand, this instruction enables to reserve memory space for the specified count of numbers of the type given by the attribute. Therefore,

```
DS.B    20
```

for example reserves 20 bytes of memory, but

```
DS.X    20
```

reserves 240 bytes!

The other purpose is the alignment of the program counter which is achieved by a count specification of 0. In this way, with a

```
DS.W    0  ,
```

the program counter will be rounded up to the next even address, with a

```
DS.D 0
```

in contrast to the next double word boundary. Memory cells possibly staying unused thereby are neither zeroed nor filled with NOPs, they simply stay undefined.

The default for the operand length is - as usual - `W`, i.e. 16 bits.

For the 56xxx, the operand length is fixed to words (of 24 bit), attributes therefore do not exist just as in the case of `DC`.

3.3.3 DB,DW,DD,DQ, and DT

valid for: Intel, Zilog, Toshiba, NEC, TMS370, Siemens, AMD, MELPS7700/65816, M16(C), National, ST9, TMS70Cxx, μPD77230, Fairchild, Intersil

These commands are - one could say - the Intel counterpart to `DS` and `DC`, and as expected, their logic is a little bit different: First, the specification of the operand length is moved into the mnemonic:

- `DB`: byte or ASCII string similar to `DC.B`
- `DW`: 16-bit integer
- `DD`: 32-bit integer or single precision
- `DQ`: double precision (64 bits)
- `DT`: extended precision (80 bits)

Second, the distinction between constant definition and memory reservation is done by the operand. A reservation of memory is marked by a `?` :

```
db      ?       ; reserves a byte
dw      ?,?     ; reserves memory for 2 words (=4 byte)
dd      -1      ; places the constant -1 (FFFFFFFFH) !
```

Reserved memory and constant definition **must not** be mixed within one instruction:

```
db      "hello",?       ; --> error message
```

Additionally, the `DUP` Operator permits the repeated placing of constant sequences or the reservation of whole memory blocks:

```
db      3 dup (1,2)     ; --> 1 2 1 2 1 2
dw      20 dup (?)      ; reserves 40 bytes of memory
```

As you can see, the DUP-argument must be enclosed in parentheses, which is also why it may consist of several components, that may themselves be DUPs...the stuff therefore works recursively. DUP is however also a place where one can get in touch with another limit of the assembler: a maximum of 1024 bytes of code or data may be generated in one line. This is not valid for the reservation of memory, only for the definition of constant arrays!

In order to be compatible to the M80, DEFB/DEFW may be used instead of DB/DW in Z80-mode.

Similarly, BYTE/ADDR resp. WORD/ADDRW in COP4/8 mode are an alias for DB resp. DW, with the pairs differing in byte order: instructions defined by National for address storage use big endian, BYTE resp. WORD in contrast use little endian.

The NEC 77230 is special with its DW instruction: It more works like the DATA statement of its smaller brothers, but apart from string and integer arguments, it also accepts floating point values (and stores them in the processor's proprietary 32-bit format). There is *no* DUP operator!

3.3.4 DS, DS8

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

With this instruction, you can reserve a memory area:

```
DS      <count>
```

It is an abbreviation of

```
DB      <count> DUP (?)
```

Although this could easily be made by a macro, some people grown up with Motorola CPUs (Hi Michael!) suggest DS to be a built-in instruction...I hope they are satisfied now ;-)

DS8 is defined as an alias for DS on the National SC14xxx. Beware that the code memory of these processors is organized in words of 16 bits, it is therefore impossible to reserve individual bytes. In case the argument of DS is odd, it will be rounded up to the next even number.

3.3.5 BYT or FCB

valid for: 6502, 68xx

By this instruction, byte constants or ASCII strings are placed in 65xx/68xx-mode, it therefore corresponds to `DC.B` on the 68000 or `DB` on Intel. Similarly to `DC`, a repetition factor enclosed in brackets ([..]) may be prepended to every single parameter.

3.3.6 BYTE

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

Ditto. Note that when in 320C2(0)x/5x mode, the assembler assumes that a label on the left side of this instruction has no type, i.e. it belongs to no address space. This behaviour is explained in the processor-specific hints.

The `PADDING` instruction allows to set whether odd counts of bytes shall be padded with a zero byte in MSP/TMS9900 mode.

3.3.7 DC8

valid for: SC144xx

This statement is an alias for `DB`, i.e. it may be used to dump byte constants or strings to memory.

3.3.8 ADR or FDB

valid for: 6502, 68xx

`ADR` resp. `FDB` stores word constants when in 65xx/68xx mode. It is therefore the equivalent to `DC.W` on the 68000 or `DW` on Intel platforms. Similarly to `DC`, a repetition factor enclosed in brackets ([..]) may be prepended to every single parameter.

3.3.9 WORD

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

If assembling for the 320C3x or i960, this command stores 32-bit words, 16-bit words for the other families. Note that when in 320C2(0)x/5x mode, the assembler assumes that a label on the left side of this instruction has no type, i.e. it belongs to no address space. This behaviour is explained at the discussion on processor-specific hints.

3.3.10 DW16

valid for: SC144xx

This instruction is for SC144xx targets a way to dump word (16 bit) constants to memory. `CAUTION!!` It is therefore an alias for `DW`.

3.3.11 LONG

valid for: 320C2(0)x, 320C5x

LONG stores a 32-bit integer to memory with the order LoWord-HiWord. Note that when in 320C2(0)x/5x mode, the assembler assumes that a label on the left side of this instruction has no type, i.e. it belongs to no address space. This behaviour is explained in the processor-specific hints.

3.3.12 SINGLE, DOUBLE, and EXTENDED

valid for: 320C3x (not `DOUBLE`*), 320C6x (not* `EXTENDED`*)*

Both commands store floating-point constants to memory. In case of the 320C3x, they are **not** stored in IEEE-format. Instead the processor-specific formats with 32 and 40 bit are used. In case of `EXTENDED` the resulting constant occupies two memory words. The most significant 8 bits (the exponent) are written to the first word while the other ones (the mantissa) are copied into the second word.

3.3.13 FLOAT and DOUBLE

valid for: 320C2(0)x, 320C5x

These two commands store floating-point constants in memory using the standard IEEE 32-bit and 64-bit IEEE formats. The least significant byte is copied to the first allocated memory location. Note that when in 320C2(0)x/5x mode the assembler assumes that all labels on the left side of an instruction have no type, i.e. they belong to no address space. This behaviour is explained in the processor-specific hints.

3.3.14 EFLOAT, BFLOAT, and TFLOAT

valid for: 320C2(0)x, 320C5x

Another three floating point commands. All of them support non-IEEE formats, which should be easily applicable on signal processors:

- `EFLOAT`: mantissa with 16 bits, exponent with 16 bits
- `BFLOAT`: mantissa with 32 bits, exponent with 16 bits
- `DFLOAT`: mantissa with 64 bits, exponent with 32 bits

The three commands share a common storage strategy. In all cases the mantissa precedes the exponent in memory, both are stored as 2's complement with the least significant byte first. Note that when in 320C2(0)x/5x mode the assembler assumes that all labels on the left side of an instruction have no type, i.e. they belong to no address space. This behaviour is explained in the processor-specific hints.

3.3.15 Qxx and LQxx

valid for: 320C2(0)x, 320C5x

`Qxx` and `LQxx` can be used to generate constants in a fixed point format. `xx` denotes a 2-digit number. The operand is first multiplied by 2^{xx} before converting it to binary notation. Thus `xx` can be viewed as the number of bits which should be reserved for the fractional part of the constant in fixed point format. `Qxx` stores only one word (16 bit) while `LQxx` stores two words (low word first):

```
q05     2.5     ; --> 0050h
lq20    ConstPI ; --> 43F7h 0032h
```

Please do not flame me in case I calculated something wrong on my HP28...

3.3.16 DATA

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

This command stores data in the current segment. Both integer values as well as character strings are supported. On 16C5x/16C8x, 17C4x in data segment and on the 4500, characters occupy one word. On AVR, 17C4x in code segment, μPD772x in the data segments, and on 3201x/3202x, in general two characters fit into one word (LSB first). The μPD77C25 can hold three bytees per word in the code segment. When in 320C3x, mode the assembler puts four characters into one word (MSB first). In contrast to this characters occupy two memory locations in the data segment of the 4500, similar in the 4004. The range of integer values corresponds to the word width of each processor in a specific segment. This means that `DATA` has the same result than `WORD` on a 320C3x (and that of `SINGLE` if AS recognizes the operand as a floating-point constant).

3.3.17 ZERO

valid for: PIC

Generates a continuous string of zero words in memory. The length is given by the argument and must not exceed 512.

3.3.18 FB and FW

valid for: COP4/8

These instruction allow to fill memory blocks with a byte or word constant. The first operand specifies the size of the memory block while the second one sets the filling constant itself. The maximum supported block size is 1024 elements for `FB` and 512 elements for `FW`.

3.3.19 ASCII and ASCIZ

valid for: ST6

Both commands store string constants to memory. While `ASCII` writes the character information only, `ASCIZ` additionally appends a zero to the end of the string.

3.3.20 STRING and RSTRING

valid for: 320C2(0)x, 320C5x

These commands are functionally equivalent to `DATA`, but integer values are limited to the range of byte values. This enables two characters or numbers to be packed together into one word. Both commands only differ in the order they use to write bytes: `STRING` stores the upper one first then the lower one, `RSTRING` does this vice versa. Note that when in 320C2(0)x/5x mode the assembler assumes that a label on the left side of this instruction has no type, i.e. it belongs to no address space. This behaviour is explained in the processor-specific hints.

3.3.21 FCC

valid for: 6502, 68xx

When in 65xx/68xx mode, string constants are generated using this instruction. In contrast to the original assembler AS11 from Motorola (this is the main reason why AS understands this command, the functionality is contained within the `BYT` instruction) you must enclose the string argument by double quotation marks instead of single quotation marks or slashes. Similarly to `DC`, a repetition factor enclosed in brackets ([..]) may be prepended to every single parameter.

3.3.22 DFS or RMB

valid for: 6502, 68xx

Reserves a memory block when in 6502/68xx mode. It is therefore the equivalent to `DS.B` on the 68000 or `DB ?` on Intel platforms.

3.3.23 BLOCK

valid for: ST6

Ditto.

3.3.24 SPACE

valid for: i960

Ditto.

3.3.25 RES

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

This command allocates memory. When used in code segments the argument counts words (10/12/14/16 bit). In data segments it counts bytes for PICs, nibbles for 4500's and words for the TI devices.

3.3.26 BSS

valid for: 320C2(0)x, 320C3x, 320C5x, 320C6x, MSP

`BSS` works like `RES`, but when in 320C2(0)x/5x mode, the assembler assumes that a label on the left side of this instruction has no type, i.e it belongs to no address space. This behaviour is explained in the processor-specific hints.

3.3.27 DSB and DSW

valid for: COP4/8

Both instructions allocate memory and ensure compatibility to ASMCOP from National. While `DSB` takes the argument as byte count, `DSW` uses it as word count (thus it allocates twice as much memory than `DSB`).

3.3.28 DS16

valid for: SC144xx

This instruction reserves memory in steps of full words, i.e. 16 bits. It is an alias for `DW`.

3.3.29 ALIGN

valid for: all processors

Takes the argument to align the program counter to a certain address boundary. AS increments the program counter to the next multiple of the argument. So, `ALIGN` corresponds to `DS.x` on 68000, but is much more flexible at the same time.

Example:

```
        align     2
```

aligns to an even address (PC mod 2 = 0). The contents of the skipped addresses is left undefined.

3.3.30 LTORG

valid for: SH7x00

Although the SH7000 processor can do an immediate register load with 8 bit only, AS shows up with no such restriction. This behaviour is instead simulated through constants in memory. Storing them in the code segment (not far away from the register load instruction) would require an additional jump. AS Therefore gathers the constants an stores them at an address specified by `LTORG`. Details are explained in the processor-specific section somewhat later.

3.4 Macro Instructions

valid for: all processors

Now we finally reach the things that make a macro assembler different from an ordinary assembler: the ability to define macros (guessed it !?).

When speaking about 'macros', I generally mean a sequence of (machine or pseudo) instructions which are united to a block by special statements and can then be treated in certain ways. The assembler knows the following statements to work with such blocks:

3.4.1 MACRO

is probably the most important instruction for macro programming. The instruction sequence

```
<name>  MACRO   [parameter list]
        <instructions>
        ENDM
```

defines the macro <name> to be the enclosed instruction sequence. This definition by itself does not generate any code! In turn, from now on the instruction sequence can simply be called by the name, the whole construct therefore shortens and simplifies programs. A parameter list may be added to the macro definition to make things even more useful. The parameters' names have to be separated by commas (as usual) and have to conform to the conventions for symbol names (see section 2.7) - like the macro name itself.

A switch to case-sensitive mode influences both macro names and parameters.

Similar to symbols, macros are local, i.e. they are only known in a section and its subsections when the definition is done from within a section. This behaviour however can be controlled in wide limits via the options `PUBLIC` and `GLOBAL` described below.

Apart from the macro parameters themselves, the parameter list may contain control parameters which influence the processing of the macro. These parameters are distinguished from normal parameters by being enclosed in braces. The following control parameters are defined:

- EXPAND/NOEXPAND: rule whether the enclosed code shall be written to the listing when the macro is expanded. The default is the value set by the pseudo instruction MACEXP.

- PUBLIC[:section name]: assigns the macro to a parent section instead of the current section. A section can make macros accessible for the outer code this way. If the section specification is missing, the macro becomes completely global, i.e. it may be referenced from everywhere.

- GLOBAL[:section name]: rules that in addition to the macro itself, another macro shall be generated that has the same contents but is assigned to the specified section. Its name is constructed by concatenating the current section's name to the macro name. The section specified must be a parent section of the current section; if the specification is missing, the additional macro becomes globally visible. For example, if a macro A is defined in a section B that is a child section of section C, an additional global macro named C_B_A would be generated. In contrast, if C had been specified as target section, the macro would be named B_A and be assigned to section C. This option is turned off by default and it only has an effect when it is used from within a section. The macro defined locally is not influenced by this option.

- EXPORT/NOEXPORT: rules whether the definition of this macro shall be written to a separate file in case the -M command line option was given. This way, definitions of 'private' macros may be mapped out selectively. The default is FALSE, i.e. the definition will not be written to the file. The macro will be written with the concatenated name if the GLOBAL option was additionally present.

- INTLABEL/NOINTLABEL : rules whether a label defined in a line that calls this macro may be used as an additional parameter inside the label or not, instead of simply 'labeling' the line.

The control parameters described above are removed from the parameter list by AS, i.e. they do not have a further influence on processing and usage.

When a macro is called, the parameters given for the call are textually inserted into the instruction block and the resulting assembler code is assembled as usual. Zero length parameters are inserted in case too few parameters are

specified. It is important to note that string constants are not protected from macro expansions. The old IBM rule:

It's not a bug, it's a feature!

applies for this detail. The gap was left to allow checking of parameters via string comparisons. For example, one can analyze a macro parameter in the following way:

```
mul     MACRO   para,parb
        IF      UpString("PARA")<>"A"
         MOV    a,para
        ENDIF
        IF      UpString("PARB")<>"B"
         MOV    b,parb
        ENDIF
        mul     ab
        ENDM
```

It is important for the example above that the assembler converts all parameter names to upper case when operating in case-insensitive mode, but this conversion never takes place inside of string constants. Macro parameter names therefore have to be written in upper case when they appear in string constants.

The same naming rules as for usual symbols also apply for macro parameters, with the exception that only letters and numbers are allowed, i.e. dots and underscores are forbidden. This constraint has its reason in a hidden feature: the underscore allows to concatenate macro parameter names to a symbol, like in the following example:

```
concat  macro   part1,part2
        call    part1_part2
        endm
```

The call

```
        concat  module,function
```

will therefore result in

```
        call    module_function
```

Apart from the parameters explicitly declared for a macro, four more 'implicitly' declared parameters exist. Since they are always present, they cannot not be redeclared as explicit parameters:

- `ATTRIBUTE` refers to the attribute appended to the macro call, in case the currently active architecture supports attributes for machine instructions. See below for an example!
- `ALLARGS` refers to a comma-separated list of all arguments passed to a macro, usable e.g. to pass them on to a IRP statement.
- `ARGCOUNT` refers to the actual count of parameters passed to a macro. Note however that this number is never lower than the formal parameter count of the macro, since AS will fill up missing arguments with empty strings!
- `__LABEL__` refers to a label present in a line that calls the macro. This replacement only takes place if the `INTLABEL` option was set for this macro!

The purpose of being able to 'internally' use a label in a macro is surely not immediately obvious. There might be cases where moving the macro's entry point into its body may be useful. The most important application however are TI signal processors that use a double pipe symbol in the label's column to mark parallelism, like this:

```
    instr1
||  instr2
```

(since both instructions merge into a single word of machine code, you cannot branch to the second instruction - so occupying the label's position doesn't hurt). The problem is however that some 'convenience instructions' are realized as macros. A prallelization symbol written in front of a macro call normally would be assigned to the macro itself, *not to the macro body's first instruction*. However, things work with this trick:

```
myinstr    macro {INTLABEL}
__LABEL__  instr2
           endm

           instr1
||         myinstr
```

The result after expanding `myinstr` is identical to the previous example without macro.

Recursion of macros, i.e. the repeated call of a macro from within its own body is completely legal. However, like for any other sort of recursion, one has to assure that there is an end at someplace. For cases where one forgot this, AS keeps an internal counter for every macro that is incremented when an expansion of this macro is begun and decremented again when the expansion is completed. In case of recursive calls, this counter reaches higher and higher values, and at a limit settable via `NESTMAX`, AS will refuse to expand. Be careful when you turn off this emergency brake: the memory consumption on the heap may go beyond all limits and even shut down a Unix system...

A small example to remove all clarities ;-)

A programmer braindamaged by years of programming Intel processors wants to have the instructions `PUSH/POP` also for the 68000. He solves the 'problem' in the following way:

```
push    macro   op
        move.ATTRIBUTE op,-(sp)
        endm

pop     macro   op
        move.ATTRIBUTE (sp)+,op
        endm
```

If one writes

```
        push    d0
        pop.l   a2    ,
```

this results in

```
        move.   d0,-(sp)
        move.l  (sp)+,a2
```

A macro definition must not cross include file boundaries.

Labels defined in macros always are regarded as being local, an explicit `LOCAL` instruction is therefore not necessary (it even does not exist). In case there is a reason to make a label global, one may define it with `LABEL` which always creates global symbols (similar to `BIT, SFR...`):

```
<Name>  label   $
```

When parsing a line, the assembler first checks the macro list afterwards looks for processor instructions, which is why macros allow to redefine processor instructions. However, the definition should appear previously to the first invocation of the instruction to avoid phase errors like in the following example:

```
        bsr     target

bsr     macro   targ
        jsr     targ
        endm

        bsr     target
```

In the first pass, the macro is not known when the first `BSR` instruction is assembled; an instruction with 4 bytes of length is generated. In the second pass however, the macro definition is immediately available (from the first pass), a `JSR` of 6 bytes length is therefore generated. As a result, all labels following are too low by 2 and phase errors occur for them. An additional pass is necessary to resolve this.

Because a machine or pseudo instruction becomes hidden when a macro of same name is defined, there is a backdoor to reach the original meaning: the search for macros is suppressed if the name is prefixed with an exclamation mark (!). This may come in handy if one wants to extend existing instructions in their functionality, e.g. the TLCS-90's shift instructions:

```
srl     macro   op,n           ; shift by n places
        rept    n               ; n simple instructions
         !srl   op
        endm
        endm
```

From now on, the `SRL` instruction has an additional parameter...

3.4.2 IRP

is a simplified macro definition for the case that an instruction sequence shall be applied to a couple of operands and the the code is not needed any more afterwards. `IRP` needs a symbol for the operand as its first parameter, and an (almost) arbitrary number of parameters that are sequentially inserted into the block of code. For example, one can write

```
        irp     op, acc,b,dpl,dph
        push    op
        endm
```

to push a couple of registers to the stack, what results in

```
        push    acc
        push    b
        push    dpl
        push    dph
```

Again, labels used are automatically local for every pass.

3.4.3 IRPC

`IRPC` is a variant of `IRP` where the first argument's occurences in the lines up to `ENDM` are successively replaced by the characters of a string instead of further parameters. For example, an especially complicated way of placing a string into memory would be:

```
irpc    char,"Hello World"
db      'CHAR'
endm
```

CAUTION! As the example already shows, IRPC only inserts the pure character; it is the programmer's task to assure that valid code results (in this example by inserting quotes, including the detail that no automatic conversion to uppercase characters is done).

3.4.4 REPT

is the simplest way to employ macro constructs. The code between REPT and ENDM is assembled as often as the integer argument of REPT specifies. This statement is commonly used in small loops to replace a programmed loop to save the loop overhead.

An example for the sake of completeness:

```
rept    3
rr      a
endm
```

rotates the accumulator to the right by three digits.

In case REPT's argument is equal to or smaller than 0, no expansion at all is done. This is different to older versions of AS which used to be a bit 'sloppy' in this respect and always made a single expansion.

3.4.5 WHILE

WHILE operates similarly to REPT, but the fixed number of repetitions given as an argument is replaced by a boolean expression. The code framed by WHILE and ENDM is assembled until the expression becomes logically false. This may mean in the extreme case that the enclosed code is not assembled at all in case the expression was already false when the construct was found. On the other hand, it may happen that the expression stays true forever and AS will run infinitely...one should apply therefore a bit of accuracy when one uses this construct, i.e. the code must contain a statement that influences the condition, e.g. like this:

```
cnt     set     1
sq      set     cnt*cnt
        while   sq<=1000
         dc.l    sq
cnt      set     cnt+1
sq       set     cnt*cnt
        endm
```

This example stores all square numbers up to 1000 to memory.

Currently there exists a little ugly detail for `WHILE`: an additional empty line that was not present in the code itself is added after the last expansion. This is a 'side effect' based on a weakness of the macro processor and it is unfortunately not that easy to fix. I hope noone minds...

3.4.6 EXITM

`EXITM` offers a way to terminate a macro expansion or one of the instructions `REPT, IRP,` or `WHILE` prematurely. Such an option helps for example to replace encapsulations with `IF-ENDIF`-ladders in macros by something more readable. Of course, an `EXITM` itself always has to be conditional, what leads us to an important detail: When an `EXITM` is executed, the stack of open `IF` and `SWITCH` constructs is reset to the state it had just before the macro expansion started. This is imperative for conditional `EXITM`'s as the `ENDIF` resp. `ENDCASE` that frames the `EXITM` statement will not be reached any more; AS would print an error message without this trick. Please keep also in mind that `EXITM` always only terminates the innermost construct if macro constructs are nested! If one want to completely break out of a nested construct, one has to use additional `EXITM`'s on the higher levels!

3.4.7 SHIFT

`SHIFT` is a tool to construct macros with variable argument lists: it discards the first parameter, with the result that the second parameter takes its place and so on. This way one could process a variable argument list...if you do it the right way. For example, the following does not work...

```
pushlist  macro reg
          rept  ARGCOUNT
          push  reg
          shift
          endm
          endm
```

...because the macro gets expanded once, its output is captured by REPT and then executed n times. Therefore, the first argument is saved n times...the following approach works better:

```
pushlist  macro reg
          if      "ARG"<>""
           push    arg
           shift
           pushlist ALLARGS
          endif
          endm
```

Effectively, this is a recursion that shortens the argument list once per step. The important trick is that a new macro expansion is started in each step...

3.4.8 MAXNEST

MAXNEST allows to adjust how often a mcro may be called recursively before AS terminates with an error message. The argument may be an arbitrary positive integer value, with the special value 0 turning the this security brake completely off (be careful with that...). The default value for the maximum nesting level is 256; its current value may be read from the integer symbol of same name.

3.4.9 FUNCTION

Though FUNCTION is not a macro statement in the inner sense, I will describe this instruction at this place because it uses similar principles like macro replacements.

This instruction is used to define new functions that may then be used in formula expressions like predefined functions. The definition must have the following form:

```
<name>  FUNCTION <arg>,..,<arg>,<expression>
```

The arguments are the values that are 'fed into' the function. The definition uses symbolic names for the arguments. The assembler knows by this that where to insert the actual values when the function is called. This can be seen from the following example:

```
isdigit FUNCTION ch,(ch>='0')&&(ch<='9')
```

This function checks whether the argument (interpreted as a character) is a number in the currently valid character set (the character set can be modified via `CHARSET`, therefore the careful wording).

The arguments' names (`CH` in this case) must conform to the stricter rules for macro parameter names, i.e. the special characters . and _ are not allowed.

User-defined functions can be used in the same way as builtin functions, i.e. with a list of parameters, separated by commas, enclosed in parentheses:

```
IF isdigit(char)
 message "\{char} is a number"
ELSEIF
 message "\{char} is not a number"
ENDIF
```

When the function is called, all parameters are calculated once and are then inserted into the function's formula. This is done to reduce calculation overhead and to avoid side effects. The individual arguments have to be separated by commas when a function has more than one parameter.

CAUTION! Similar to macros, one can use user-defined functions to override builtin functions. This is a possible source for phase errors. Such definitions therefore should be done before the first call!

The result's type may depend on the type of the input arguments as the arguments are textually inserted into the function's formula. For example, the function

```
double  function x,x+x
```

may have an integer, a float, or even a string as result, depending on the argument's type!

When AS operates in case-sensitive mode, the case matters when defining or referencing user-defined functions, in contrast to builtin functions!

3.5 Structures

valid for: all processors

Even in assembly language programs, there is sometimes the necessity to define composed data structures, similar to high-level languages. AS supports both the definition and usage of structures with a couple of statements. These statements shall be explained in the following section.

3.5.1 Definition

The definiton of a structure is begun with the statement `STRUCT` and ends with `ENDSTRUCT` (lazy people may also write `STRUC` resp. `ENDS` instead). A label preceding these instructions is taken as the name of the structure to be defined; it is optional at the end of the definition and may be used to redefine the length symbol's name (see below). The remaining procedure is simple: Together with `STRUCT`, the cuurent program counter is saved and reset to zero. All labels defined between `STRUCT` and `ENDSTRUCT` therefore are the offsets of the structure's data fields. Reserving space is done via the same instructions that are also otherwise used for reserving space, like e.g. `DS.x` for Motorola CPUs or `DB` & co. for Intel-style processors. The rules for rounding up lengths to assure certain alignments also apply here - if one wants to define 'packed' structures, a preceding `PADDING OFF` may be necessary. Vice versa, alignments may be forced with `ALIGN` or similar instructions.

Since such a definition only represents a sort of 'prototype', only instructions that reserve memory may be used, no instructions that dispose constants or generate code.

Labels defined inside structures (i.e. the elements' names) are not stored as-is. Instead, the structure's name is prepended to them, separated with a special character. By default, this is the underbar (_). This behaviour however may be modified with two arguments passed to the `STRUCT` statement:

- `NOEXTNAMES` suppressed the prepending of the structure's name. In this case, it is the programmer's responsibility to assure that field names are not used more than once.

- `DOTS` instructs AS to use the dot as connecting character instead of the underbar. It should however be pointed out that on certain target architectures, the dot has a special meaning for bit addressing, which may lead to problems!

It is futhermore possible to turn the usage of a dot on resp. off for all following structures:

```
dottedstructs <on|off>
```

Aside from the element names, AS also defines a further symbol with the structure's overall length when the definition has been finished. This symbol has the name `LEN`, which is being extended with the structure's name via the same rules - or alternitavely with the label name given with the `ENDSTRUCT` statement.

In practice, this may things may look like in this example:

```
Rec     STRUCT
Ident   db      ?
Pad     db      ?
Pointer dd      ?
Rec     ENDSTRUCT
```

In this example, the symbol `REC_LEN` would be assigned the value 6.

3.5.2 Usage

Once a structure has been assigned, usage is as simple as possible and similar to a macro: a simple

```
thisrec Rec
```

reserves as much memory as needed to hold an instance of the structure, and additionally defines a symbol for every element of the structure with its address, in this case `THISREC_IDENT, THISREC_PAD`, and `THISREC_POINTER`. A label naturally must not be omitted when calling a structure; if it is missing, an error will be emitted.

ATTENTION! Though AS does not expect any arguments when calling a structure, any arguments given will simply be ignored and not be reported as error. This is a precaution to allow the definition of pre-initialized structures in the future.

3.5.3 Nested Structures

Is is perfectly valid to call an already defined structure within the definition of another structure. The procedure that is taking place then is a combination of the definition and calling described in the previous two sections: elements of the substructure are being defined, the name of the instance is being prepended, and the name of the super-structure is once again geing prepended to this concatenated name. This may look like the following:

```
TreeRec struct
left    dd          ?
right   dd          ?
data    Rec
TreeRec endstruct
```

3.5.4 Unions

A union is a special form of a structure whose elements are not laid out sequentially in memory. Instead all elements occupy the *same* memory and are located at offset 0 in the structure. Naturally, suich a defnition basically does nothing more than to assign the value of zero to a couple of symbols. It may however be useful to clarify the overlap in a program and therefore to make it more 'readable'. The size of a union is the maximum of all elements' lengths.

3.5.5 Structures and Sections

Symbols that are created in the course of defining or usage of structures are treated just like normal symbols, i.e. when used within a section, these symbols are local to the section. The same is however also true for the structures themselves, i.e. a structure defined within a section cannot be used outside of the section.

3.6 Conditional Assembly

valid for: all processors

The assembler supports conditional assembly with the help of statements like `IF...` resp. `SWITCH...` . These statements work at assembly time allowing or disallowing the assembly of program parts based on conditions. They are therefore not to be compared with IF statements of high-level languages (though it would be tempting to extend assembly language with structurization statements of higher level languages...).

The following constructs may be nested arbitrarily (until a memory overflow occurs).

3.6.1 IF / ELSEIF / ENDIF

`IF` is the most common and most versatile construct. The general style of an `IF` statement is as follows:

```
IF      <expression 1>
.
.
<block 1>
.
.
ELSEIF  <expression 2>
.
.
<block 2>
.
.
(possibly more ELSEIFs)

.
.
ELSEIF
.
.
```

```
<block n>
.
.
ENDIF
```

IF serves as an entry, evaluates the first expression, and assembles block 1 if the expression is true (i.e. not 0). All further ELSEIF-blocks will then be skipped. However, if the expression is false, block 1 will be skipped and expression 2 is evaluated. If this expression turns out to be true, block 2 is assembled. The number of ELSEIF parts is variable and results in an IF-THEN-ELSE ladder of an arbitrary length. The block assigned to the last ELSEIF (without argument) only gets assembled if all previous expressions evaluated to false; it therefore forms a 'default' branch. It is important to note that only **one** of the blocks will be assembled: the first one whose IF/ELSEIF had a true expression as argument.

The ELSEIF parts are optional, i.e. IF may directly be followed by an ENDIF. An ELSEIF without parameters must be the last branch.

ELSEIF always refers to the innermost, unfinished IF construct in case IF's are nested.

In addition to IF, the following further conditional statements are defined:

- IFDEF <symbol>: true if the given symbol has been defined. The definition has to appear before IFDEF.
- IFNDEF <symbol>: counterpart to IFDEF.
- IFUSED <symbol>: true if if the given symbol has been referenced at least once up to now.
- IFNUSED <symbol>: counterpart to IFUSED.
- IFEXIST <name>: true if the given file exists. The same rules for search paths and syntax apply as for the INCLUDE instruction (see section 3.9.2).
- IFNEXIST <name>: counterpart to IFEXIST.
- IFB <arg-list>: true if all arguments of the parameter list are empty strings.

- IFNB <arg-list>: counterpart to IFB.

It is valid to write ELSE instead of ELSEIF since everybody seems to be used to it...

For every IF... statement, there has to be a corresponding ENDIF. 'Open' constructs will lead to an error message at the end of an assembly path. The way AS has 'paired' ENDIF statements with IFs may be deduced from the assembly listing: for ENDIF, the line number of the corresponding IF... will be shown.

3.6.2 SWITCH / CASE / ELSECASE / ENDCASE

CASE is a special case of IF and is designed for situations when an expression has to be compared with a couple of values. This could of course also be done with a series of ELSEIFs, but the following form

```
SWITCH  <expression>
.
.
CASE    <value 1>
.
<block 1>
.
CASE    <value 2>
.
<block 2>
.
(further CASE blocks)
.
CASE    <value n-1>
.
<block n-1>
.
ELSECASE
.
<block n>
.
ENDCASE
```

has the advantage that the expression is only written once and also only gets evaluated once. It is therefore less error-prone and slightly faster than an IF chain, but obviously not as flexible.

It is possible to specify multiple values separated by commas to a CASE statement in order to assemble the following block in multiple cases. The ELSECASE branch again serves as a 'trap' for the case that none of the CASE conditions was met. AS will issue a warning in case it is missing and all comparisons fail.

Even when value lists of CASE branches overlap, only **one** branch is executed, which is the first one in case of ambiguities.

SWITCH only serves to open the whole construct; an arbitrary number of statements may be between SWITCH and the first CASE (but don't leave other IFs open!), for the sake of better readability this should however not be done.

Similarly to IF constructs, there must be exactly one ENDCASE for every SWITCH. Analogous to ENDIF, for ENDCASE the line number of the corresponding SWITCH is shown in the listing.

3.7 Listing Control

valid for: all processors

3.7.1 PAGE

PAGE is used to tell AS the dimensions of the paper that is used to print the assembly listing. The first parameter is thereby the number of lines after which AS shall automatically output a form feed. One should however take into account that this value does **not** include heading lines including an eventual line specified with TITLE. The minimum number of lines is 5, and the maximum value is 255. A specification of 0 has the result that AS will not do any form feeds except those triggered by a NEWPAGE instruction or those implicitly engaged at the end of the assembly listing (e.g. prior to the symbol table).

The specification of the listing's length in characters is an optional second parameter and serves two purposes: on the one hand, the internal line counter of AS will continue to run correctly when a source line has to be split into several listing lines, and on the other hand there are printers (like some laser printers) that do not automatically wrap into a new line at line end but instead simply discard the rest. For this reason, AS does line breaks by itself, i.e. lines that are too long are split into chunks whose lengths are equal to or smaller than the specified width. This may lead to double line feeds on printers that can do line wraps on their own if one specifies the exact line width as listing width. The solution for such a case is to reduce the assembly listing's width by 1. The specified line width may lie between 5 and 255 characters; a line width of 0 means similarly to the page length that AS shall not do any splitting of listing lines; lines that are too long of course cannot be taken into account of the form feed then any more.

The default setting for the page length is 60 lines, the default for the line width is 0; the latter value is also assumed when `PAGE` is called with only one parameter.

CAUTION! There is no way for AS to check whether the specified listing length and width correspond to the reality!

3.7.2 NEWPAGE

`NEWPAGE` can be used to force a line feed though the current line is not full up to now. This might be useful to separate program parts in the listing that are logically different. The internal line counter is reset and the page counter is incremented by one. The optional parameter is in conjunction with a hierarchical page numbering AS supports up to a chapter depth of 4. 0 always refers to the lowest depth, and the maximum value may vary during the assembly run. This may look a bit puzzling, as the following example shows:

page 1,	instruction `NEWPAGE 0`	→ page 2
page 2,	instruction `NEWPAGE 1`	→ page 2.1
page 2.1,	instruction `NEWPAGE 1`	→ page 3.1
page 3.1,	instruction `NEWPAGE 0`	→ page 3.2
page 3.2,	instruction `NEWPAGE 2`	→ page 4.1.1

NEWPAGE <number> may therefore result in changes in different digits, depending on the current chapter depth. An automatic form feed due to a line counter overflow or a NEWPAGE without parameter is equal to NEWPAGE 0. Previous to the output of the symbol table, an implicit NEWPAGE <maximum up to now> is done to start a new 'main chapter'.

3.7.3 MACEXP

One can achieve by the statement

```
macexp  off
```

that only the macro call and not the expanded text is listed for macro expansions. This is sensible for macro intensive codes to avoid that the listing grows beyond all bounds. The full listing can be turned on again with a

```
macexp  on  .
```

This is also the default.

There is a subtle difference between the meaning of MACEXP for macros and for all other macro-like constructs (e.g. REPT): while a macro contain an internal flag that rules whether expansions of this macro shall be listed or not, MACEXP directly influences all other constructs that are resolved 'in place'. The reason for this differentiation is that there may be macros that are tested and their expansion is therefore unnecessary, but all other macros still shall be expanded. MACEXP serves as a default for the macro's internal flag when it is defined, and it may be overridden by the NOEXPAND resp. EXPAND directives.

The current setting may be read from the symbol MACEXP.

3.7.4 LISTING

works like MACEXP and accepts the same parameters, but is much more radical: After a

```
listing off   ,
```

nothing at all will be written to the listing. This directive makes sense for tested code parts or include files to avoid a paper consumption going beyond all bounds. **CAUTION!** If one forgets to issue the counterpart somewhere later, even the symbol table will not be written any more! In addition to `ON` and `OFF`, `LISTING` also accepts `NOSKIPPED` and `PURECODE` as arguments. Program parts that were not assembled due to conditional assembly will not be written to the listing when `NOSKIPPED` is set, while `PURECODE` - as the name indicates - even suppresses the `IF` directives themselves in the listing. These options are useful if one uses macros that act differently depending on parameters and one only wants to see the used parts in the listing.

The current setting may be read from the symbol `LISTING` (0=`OFF`, 1=`ON`, 2=`NOSKIPPED`, 3=`PURECODE`).

3.7.5 PRTINIT and PRTEXIT

Quite often it makes sense to switch to another printing mode (like compressed printing) when the listing is sent to a printer and to deactivate this mode again at the end of the listing. The output of the needed control sequences can be automated with these instructions if one specifies the sequence that shall be sent to the output device prior to the listing with `PRTINIT` <`string`> and similarly the deinitialization string with `PRTEXIT` <`string`>. <`string`> has to be a string expression in both cases. The syntax rules for string constants allow to insert control characters into the string without too much tweaking.

When writing the listing, the assembler does **not** differentiate where the listing actually goes, i.e. printer control characters are sent to the screen without mercy!

Example:

For Epson printers, it makes sense to switch them to compressed printing because listings are so wide. The lines

```
prtinit "\15"
prtexit "\18"
```

assure that the compressed mode is turned on at the beginning of the listing and turned off afterwards.

3.7.6 TITLE

The assembler normally adds a header line to each page of the listing that contains the source file's name, date, and time. This statement allows to extend the page header by an arbitrary additional line. The string that has to be specified is an arbitrary string expression.

Example:

For the Epson printer already mentioned above, a title line shall be written in wide mode, which makes it necessary to turn off the compressed mode before:

```
title   "\18\14Wide Title\15"
```

(Epson printers automatically turn off the wide mode at the end of a line.)

3.7.7 RADIX

`RADIX` with a numerical argument between 2 and 36 sets the default numbering system for integer constants, i.e. the numbering system used if nothing else has been stated explicitly. The default is 10, and there are some possible pitfalls to keep in mind which are described in section 2.10.1.

Independent of the current setting, the argument of `RADIX` is *always decimal*; furthermore, no symbolic or formula expressions may be used as argument. Only use simple constant numbers!

3.7.8 OUTRADIX

`OUTRADIX` can in a certain way be regarded as the opposite to `RADIX`: This statement allows to configure which numbering system to use for integer results when `\{...}` constructs are used in string constants (see section 2.10.3). Valid arguments range again from 2 to 36, while the default is 16.

3.8 Local Symbols

valid for: all processors

local symbols and the section concept introduced with them are a completely new function that was introduced with version 1.39. One could say that this part is version "1.0" and therefore probably not the optimum. Ideas and (constructive) criticism are therefore especially wanted. I admittedly described the usage of sections how I imagined it. It is therefore possible that the reality is not entirely equal to the model in my head. I promise that in case of discrepancies, changes will occur that the reality gets adapted to the documentation and not vice versa (I was told that the latter sometimes takes place in larger companies...).

AS does not generate linkable code (and this will probably not change in the near future `:-(`). This fact forces one to always assemble a program in a whole. In contrast to this technique, a separation into linkable modules would have several advantages:

- shorter assembly times as only the modified modules have to be re-assembled;
- the option to set up defined interfaces among modules by definition of private and public symbols;
- the smaller length of the individual modules reduces the number of symbols per module and therefore allows to use shorter symbol names that are still unique.

Especially the last item was something that always nagged me: once there was a label's name defined at the beginning of a 2000-lines program, there was no way to reuse it somehow - even not at the file's other end where routines with a completely different context were placed. I was forced to use concatenated names in the style of

```
<subprogram name>_<symbol name>
```

that had lengths ranging from 15 to 25 characters and made the program difficult to overlook. The concept of section described in detail in the following text was designed to cure at least the second and third item of the list above. It is completely optional: if you do not want to use sections, simply forget them and continue to work like you did with previous versions of AS.

3.8.1 Basic Definition (SECTION/ENDSECTION)

A section represents a part of the assembler program enclosed by special statements and has a unique name chosen by the programmer:

```
.
.
<other code>
.
.
SECTION <section's name>
.
.
<code inside of the section>
.
.
ENDSECTION [section's name]
.
.
<other code>
.
.
```

The name of a section must conform to the conventions for s symbol name; AS stores section and symbol names in separate tables which is the reason why a name may be used for a symbol and a section at the same time. Section names must be unique in a sense that there must not be more than one section on the same level with the same name (I will explain in the next part what "levels" mean). The argument of `ENDSECTION` is optional, it may also be omitted; if it is omitted, AS will show the section's name that has been closed with this `ENDSECTION`. Code inside a section will be processed by AS exactly as if it were outside, except for three decisive differences:

- Symbols defined within a section additionally get an internally generated number that corresponds to the section. These symbols are not accessible by code outside the section (this can be changed by pseudo instructions, later more about this).

- The additional attribute allows to define symbols of the same name inside and outside the section; the attribute makes it possible to use a symbol name multiple times without getting error messages from AS.
- If a symbol of a certain name has been defined inside and outside of a section, the "local" one will be preferred inside the section, i.e. AS first searches the symbol table for a symbol of the referenced name that also was assigned to the section. A search for a global symbol of this name only takes place if the first search fails.

This mechanism e.g. allows to split the code into modules as one might have done it with linkable code. A more fine-grained approach would be to pack every routine into a separate section. Depending on the individual routines' lengths, the symbols for internal use may obtain very short names.

AS will by default not differentiate between upper and lower case in section names; if one however switches to case-sensitive mode, the case will be regarded just like for symbols.

The organization described up to now roughly corresponds to what is possible in the C language that places all functions on the same level. However, as my "high-level" ideal was Pascal and not C, I went one step further:

3.8.2 Nesting and Scope Rules

It is valid to define further sections within a section. This is analog to the option given in Pascal to define procedures inside a procedure or function. The following example shows this:

```
sym     EQU        0

        SECTION    ModuleA

         SECTION    ProcA1

sym       EQU        5

         ENDSECTION ProcA1
```

```
          SECTION    ProcA2
sym        EQU        10
          ENDSECTION ProcA2
         ENDSECTION ModuleA

         SECTION    ModuleB
sym       EQU        15
          SECTION    ProcB
          ENDSECTION ProcB
         ENDSECTION ModuleB
```

When looking up a symbol, AS first searches for a symbol assigned to the current section, and afterwards traverses the list of parent sections until the global symbols are reached. In our example, the individual sections see the values given in table 3.5 for the symbol `sym`: This rule can be overridden by

section	value	from section...
Global	0	Global
`ModuleA`	0	Global
`ProcA1`	5	`ProcA1`
`ProcA2`	10	`ProcA2`
`ModuleB`	15	`ModuleB`
`ProcB`	15	`ModuleB`

Table 3.5: Valid values for the Individual Sections

explicitly appending a section's name to the symbol's name. The section's name has to be enclosed in brackets:

```
move.l  #sym[ModulB],d0
```

Only sections that are in the parent section path of the current section may be used. The special values PARENT0..PARENT9 are allowed to reference the n-th "parent" of the current section; PARENT0 is therefore equivalent to the current section itself, PARENT1 the direct parent and so on. PARENT1 may be abbreviated as PARENT. If no name is given between the brackets, like in this example:

```
        move.l  #sym[],d0 ,
```

one reaches the global symbol. **CAUTION!** If one explicitly references a symbol from a certain section, AS will only seek for symbols from this section, i.e. the traversal of the parent sections path is omitted!

Similar to Pascal, it is allowed that different sections have subsections of the same name; the principle of locality avoids irritations. One should IMHO still use this feature as seldom as possible: Symbols listed in the symbol resp. cross reference list are only marked with the section they are assigned to, not with the "section hierarchy" lying above them (this really would have busted the available space); a differentiation is made very difficult this way.

As a SECTION instruction does not define a label by itself, the section concept has an important difference to Pascal's concept of nested procedures: a pascal procedure can automatically "see" its subprocedures(functions), AS requires an explicit definition of an entry point. This can be done e.g. with the following macro pair:

```
proc    MACRO   name
        SECTION name
name    LABEL   $
        ENDM

endp    MACRO   name
        ENDSECTION name
        ENDM
```

This example also shows that the locality of labels inside macros is not influenced by sections. It makes the trick with the LABEL instruction necessary.

This does of course not solve the problem completely. The label is still local and not referencable from the outside. Those who think that it would suffice to place the label in front of the SECTION statement should be quiet because they would spoil the bridge to the next theme:

3.8.3 PUBLIC and GLOBAL

The `PUBLIC` statement allows to change the assignment of a symbol to a certain section. It is possible to treat multiple symbols with one statement, but I will use an example with only one symbol in the following (not hurting the generality of this discussion). In the simplest case, one declares a symbol to be global, i.e. it can be referenced from anywhere in the program:

```
        PUBLIC  <name>
```

As a symbol cannot be moved in the symbol table once it has been sorted in, this statement has to appear **before** the symbol itself is defined. AS stores all `PUBLICs` in a list and removes an entry from this list when the corresponding symbol is defined. AS prints errors at the end of a section in case that not all `PUBLICs` have been resolved.

Regarding the hierarchical section concept, the method of defining a symbol as purely global looks extremely brute. There is fortunately a way to do this in a bit more differentiated way: by appending a section name:

```
        PUBLIC  <name>:<section>
```

The symbol will be assigned to the referenced section and therefore also becomes accessible for all its subsections (except they define a symbol of the same name that hides the "more global" symbol). AS will naturally protest if several subsections try to export a symbol of same name to the same level. The special `PARENTn` values mentioned in the previous section are also valid for <`section`> to export a symbol exactly `n` levels up in the section hierarchy. Otherwise only sections that are parent sections of the current section are valid for <`section`>. Sections that are in another part of the section tree are not allowed. If several sections in the parent section path should have the same name (this is possible), the lowest level will be taken.

This tool lets the abovementioned macro become useful:

```
proc    MACRO   name
        SECTION name
        PUBLIC  name:PARENT
name    LABEL   $
        ENDM
```

This setting is equal to the Pascal model that also only allows the "father" to see its children, but not the "grandpa".

AS will quarrel about double-defined symbols if more than one section attempts to export a symbol of a certain name to the same upper section. This is by itself a correct reaction, and one needs to "qualify" symbols somehow to make them distinguishable if these exports were deliberate. A `GLOBAL` statement does just this. The syntax of `GLOBAL` is identical to `PUBLIC`, but the symbol stays local instead of being assigned to a higher section. Instead, an additional symbol of the same value but with the subsection's name appended to the symbol's name is created, and only this symbol is made public according to the section specification. If for example two sections `A` and `B` both define a symbol named `SYM` and export it with a `GLOBAL` statement to their parent section, the symbols are sorted in under the names `A_SYM` resp. `B_SYM` .

In case that source and target section are separated by more than one level, the complete name path is prepended to the symbol name.

3.8.4 FORWARD

The model described so far may look beautiful, but there is an additional detail not present in Pascal that may spoil the happiness: Assembler allows forward references. Forward references may lead to situations where AS accesses a symbol from a higher section in the first pass. This is not a disaster by itself as long as the correct symbol is used in the second pass, but accidents of the following type may happen:

```
loop:   .
        <code>
        .
        .
        SECTION sub
        .               ; ***
        .
        bra.s   loop
        .
        .
```

```
loop:   .
        .
        ENDSECTION
        .
        .
        jmp     loop    ; main loop
```

AS will take the global label `loop` in the first pass and will quarrel about an out-of-branch situation if the program part at <`code`> is long enough. The second pass will not be started at all. One way to avoid the ambiguity would be to explicitly specify the symbol's section:

```
        bra.s   loop[sub]
```

If a local symbol is referenced several times, the brackets can be saved by using a `FORWARD` statement. The symbol is thereby explicitly announced to be local, and AS will only look in the local symbol table part when this symbol is referenced. For our example, the statement

```
        FORWARD loop
```

should be placed at the position marked with `***`.

`FORWARD` must not only be stated prior to a symbol's definition, but also prior to its first usage in a section to make sense. It does not make sense to define a symbol private and public; this will be regarded as an error by AS.

3.8.5 Performance Aspects

The multi-stage lookup in the symbol table and the decision to which section a symbol shall be assigned of course cost a bit of time to compute. An 8086 program of 1800 lines length for example took 34.5 instead of 33 seconds after a modification to use sections (80386 SX, 16MHz, 3 passes). The overhead is therefore limited. As it has already been stated at the beginning, is is up to the programmer if (s)he wants to accept it. One can still use AS without sections.

3.9 Miscellaneous

3.9.1 SHARED

valid for: all processors

This statement instructs AS to write the symbols given in the parameter list (regardless if they are integer, float or string symbols) together with their values into the share file. It depends upon the command line parameters described in section 2.4 whether such a file is generated at all and in which format it is written. If AS detects this instruction and no share file is generated, a warning is the result.

CAUTION! A comment possibly appended to the statement itself will be copied to the first line outputted to the share file (if `SHARED`'s argument list is empty, only the comment will be written). In case a share file is written in C or Pascal format, one has to assure that the comment itself does not contain character sequences that close the comment ("*/" resp. "*)"). AS does not check for this!

3.9.2 INCLUDE

valid for: all processors

This instruction inserts the file given as a parameter into the just as if it would have been inserted with an editor (the file name may optionally be enclosed with " characters). This instruction is useful to split source files that would otherwise not fit into the editor or to create "tool boxes".

In case that the file name does not have an extension, it will automatically be extended with `INC`.

Via the `-i <path list>` option, one can specify a list of directories that will automatically be searched for the file. If the file is not found, a **fatal** error occurs, i.e. assembly terminates immediately.

For compatibility reasons, it is valid to enclose the file name in " characters, i.e.

```
include stddef51
```

and

```
include "stddef51.inc"
```

are equivalent. **CAUTION!** This freedom of choice is the reason why only a string constant but no string expression is allowed!

The search list is ignored if the file name itself contains a path specification.

3.9.3 BINCLUDE

valid for: all processors

`BINCLUDE` can be used to embed binary data generated by other programs into the code generated by AS (this might theoretically even be code created by AS itself...). `BINCLUDE` has three forms:

```
BINCLUDE <file>
```

This way, the file is completely included.

```
BINCLUDE <file>,<offset>
```

This way, the file's contents are included starting at `<offset>` up to the file's end.

```
BINCLUDE <file>,<offset>,<length>
```

This way, <`length`> bytes are included starting at <`offset`>.

The same rules regarding search paths apply as for `INCLUDE`.

3.9.4 MESSAGE, WARNING, ERROR, and FATAL

valid for: all processors

Though the assembler checks source files as strict as possible and delivers differentiated error messages, it might be necessary from time to time to issue additional error messages that allow an automatic check for logical error. The assembler distinguishes among three different types of error messages that are accessible to the programmer via the following three instructions:

- `WARNING`: Errors that hint at possibly wrong or inefficient code. Assembly continues and a code file is generated.
- `ERROR`: True errors in a program. Assembly continues to allow detection of possible further errors in the same pass. A code file is not generated.
- `FATAL`: Serious errors that force an immediate termination of assembly. A code file may be generated but will be incomplete.

All three instructions have the same format for the message that shall be issued: an arbitrary (possibly computed?!) string expression which may therefore be either a constant or variable.

These instructions generally only make sense in conjunction wit conditional assembly. For example, if there is only a limited address space for a program, one can test for overflow in the following way:

```
ROMSize equ     8000h   ; 27256 EPROM

ProgStart:
        .
        .
        <the program itself>
        .
        .
ProgEnd:

        if      ProgEnd-ProgStart>ROMSize
         error  "\athe program is too long!"
        endif
```

Apart from the instructions generating errors, there is also an instruction MESSAGE that simply prints a message to the console resp. to the assembly listing. Its usage is equal to the other three instructions.

3.9.5 READ

valid for: all processors

One could say that READ is the counterpart to the previous instruction group: it allows to read values from the keyboard during assembly. You might ask what this is good for. I will break with the previous principles and put an example before the exact description to outline the usefulness of this instruction:

A program needs for data transfers a buffer of a size that should be set at assembly time. One could store this size in a symbol defined with EQU, but it can also be done interactively with READ:

```
IF      MomPass=1
 READ    "buffer size",BufferSize
ENDIF
```

Programs can this way configure themselves dynamically during assembly and one could hand over the source to someone who can assemble it without having to dive into the source code. The IF conditional shown in the example should always be used to avoid bothering the user multiple times with questions.

READ is quite similar to SET with the difference that the value is read from the keyboard instead of the instruction's arguments. This for example also implies that AS will automatically set the symbol's type (integer, float or string) or that it is valid to enter formula expressions instead of a simple constant.

READ may either have one or two parameters because the prompting message is optional. AS will print a message constructed from the symbol's name if it is omitted.

3.9.6 RELAXED

valid for: all processors

By default, AS assigns a distinct syntax for integer constants to a processor family (which is in general equal to the manufacturer's specifications, as long as the syntax is not too bizarre...). Everyone however has his own preferences for another syntax and may well live with the fact that his programs cannot be translated any more with the standard assembler. If one places the instruction

```
RELAXED ON
```

right at the program's beginning, one may furtherly use any syntax for integer constants, even mixed in a program. AS tries to guess automatically for every expression the syntax that was used. This automatism does not always deliver the result one might have in mind, and this is also the reason why this option has to be enable explicitly: if there are no prefixes or postfixes that unambiguously identify either Intel or Motorola syntax, the C mode will be used. Leading zeroes that are superfluous in other modes have a meaning in this mode:

```
move.b  #08,d0
```

This constant will be understood as an octal constant and will result in an error message as octal numbers may only contain digits from 0 to 7. One might call this a lucky case; a number like 077 would result in trouble without getting a message about this. Without the relaxed mode, both expressions unambiguously would have been identified as decimal constants.

The current setting may be read from a symbol with the same name.

3.9.7 END

valid for: all processors

`END` marks the end of an assembler program. Lines that eventually follow in the source file will be ignored. **IMPORTANT:** `END` may be called from within a macro, but the `IF`-stack for conditional assembly is not cleared automatically. The following construct therefore results in an error:

```
IF      DontWantAnymore
 END
ELSEIF
```

`END` may optionally have an integer expression as argument that marks the program's entry point. AS stores this in the code file with a special record and it may be post-processed e.g. with P2HEX.

`END` has always been a valid instruction for AS, but the only reason for this in earlier releases of AS was compatibility; `END` had no effect.

Chapter 4

Processor-specific Hints

When writing the individual code generators, I strived for a maximum amount of compatibility to the original assemblers. However, I only did this as long as it did not mean an unacceptable additional amount of work. I listed important differences, details and pitfalls in the following chapter.

4.1 6811

"Where can I buy such a beast, a HC11 in NMOS?", some of you might ask. Well, of course it does not exist, but an H cannot be represented in a hexadecimal number (older versions of AS would not have accepted such a name because of this), and so I decided to omit all the letters...

> *"Someone stating that something is impossible should be at least as cooperative as not to hinder the one who currently does it."*

From time to time, one is forced to revise one's opinions. Some versions earlier, I stated at his place that I couldn't use AS's parser in a way that it is also possible to to separate the arguments of `BSET/BCLR` resp. `BRSET/BRCLR` with spaces. However, it seems that it can do more than I wanted to believe...after the n+1th request, I sat down once again to work on it and things seem to work now. You may use either spaces or commas, but not

in all variants, to avoid ambiguities: for every variant of an instruction, it is possible to use only commas or a mixture of spaces and commas as Motorola seems to have defined it (their data books do not always have the quality of the corresponding hardware...):

```
Bxxx  abs8 #mask          is equal to Bxxx  abs8,#mask
Bxxx  disp8,X #mask       is equal to Bxxx  disp8,X,#mask
BRxxx abs8 #mask addr     is equal to BRxxx abs8,#mask,addr
BRxxx disp8,X #mask addr is equal to BRxxx disp8,X,#mask,addr
```

In this list, `xxx` is a synonym either for `SET` or `CLR`; `#mask` is the bit mask to be applied (the # sign is optional). Of course, the same statements are also valid for Y-indexed expression (not listed here).

With the K4 version of the HC11, Motorola has introduced a banking scheme, which one one hand easily allows to once again extend an architecture that has become 'too small', but on the other hand not really makes programmers' and tool developers' lifes simpler...how does one sensibly map something like this on a model for a programmer?

The K4 architecture *extends* the HC11 address space by 2x512 Kbytes, which means that we now have a total address space of 64+1024=1088 Kbytes. AS acts like this were one large unified addres space, with the following layout:

- $000000...$00ffff: the old HC11 address space
- $010000...$08ffff: Window 1
- $090000...$10ffff: Window 2

Via the `ASSUME` statement, one tells AS how the banking registers are set up, which in turn describes which extended areas are mapped to which physical addresses. For absolute addresses modes with addresses beyond $10000, AS automatically computes the address within the first 64K that is to be used. Of course this only works for direct addressing modes, it is the programmer's responsibility to keep the overview for indirect or indexed addressing modes!

In case one is not really sure if the current mapping is really the desired one, the pseudo instruction `PRWINS` may be used, which prints the assumes MMxxx register contents plus the current mapping(s), like this:

```
MMSIZ $e1 MMWBR $84 MM1CR $00 MM2CR $80
Window 1: 10000...12000 --> 4000...6000
Window 1: 90000...94000 --> 8000...c000
```

An instruction

```
        jmp     *+3
```

located at $10000 would effectively result in a jump to address $4003.

4.2 PowerPC

Of course, it is a bit crazy idea to add support in AS for a processor that was mostly designed for usage in work stations. Remember that AS mainly is targeted at programmers of single board computers. But things that today represent the absolute high end in computing will be average tomorrow and maybe obsolete the next day, and in the meantime, the Z80 as the 8088 have been retired as CPUs for personal computers and been moved to the embedded market; modified versions are marketed as microcontrollers. With the appearance of the MPC505 and PPC403, my suspicion has proven to be true that IBM and Motorola try to promote this architecture in as many fields as possible.

However, the current support is a bit incomplete: Temporarily, the Intel-style mnemonics are used to allow storage of data and the more uncommon RS/6000 machine instructions mentioned in [57] are missing (hopefully noone misses them!). I will finish this as soon as information about them is available!

4.3 DSP56xxx

Motorola, which devil rode you! Which person in your company had the "brilliant" idea to separate the parallel data transfers with spaces! In result, everyone who wants to make his code a bit more readable, e.g. like this:

```
        move    x:var9 ,r0
        move    y:var10,r3   ,
```

is p****ed because the space gets recognized as a separator for parallel data transfers!

Well...Motorola defined it that way, and I cannot change it. Using tabs instead of spaces to separate the parallel operations is also allowed, and the individual operations' parts are again separated with commas, as one would expect it.

[52] states that instead of using `MOVEC, MOVEM, ANDI` or `ORI`, it is also valid to use the more general Mnemonics `MODE, AND` or `OR`. AS (currently) does not support this.

4.4 H8/300

Regarding the assembler syntax of these processors, Hitachi generously copied from Motorola (that wasn't by far the worst choice...), unfortunately the company wanted to introduce its own format for hexadecimal numbers. To make it even worse, it is a format that uses unbalanced single quotes, just like Microchip does. This is something I could not (I even did not want to) reproduce with AS, as AS uses single quotes to surround ASCII character sequences. Instead, one has to write hexadecimal numbers in the well-known Motorola syntax: with a leading dollar sign.

4.5 SH7000/7600/7700

Unfortunately, Hitachi once again used their own format for hexadecimal numbers, and once again I was not able to reproduce this with AS...please use Motorola syntax!

When using literals and the `LTORG` instruction, a few things have to be kept in mind if you do not want to suddenly get confronted with strange error messages:

Literals exist due to the fact that the processor is unable to load constants out of a range of -128 to 127 with immediate addressing. AS (and the Hitachi assembler) hide this inability by the automatic placement of constants in

memory which are then referenced via PC-relative addressing. The question that now arises is where to locate these constants in memory. AS does not automatically place a constant in memory when it is needed; instead, they are collected until an LTORG instruction occurs. The collected constants are then dumped en bloc, and their addresses are stored in ordinary labels which are also visible in the symbol table. Such a label's name is of the form

```
LITERAL_s_xxxx_n  .
```

In this name, `s` represents the literal's type. Possible values are `W` for 16-bit constants, `L` for 32-bit constants and `F` for forward references where AS cannot decide in anticipation which size is needed. In case of `s=W` or `L`, `xxxx` denotes the constant's value in a hexadecimal notation, whereas `xxxx` is a simple running number for forward references (in a forward reference, one does not know the value of a constant when it is referenced, so one obviously cannot incorporate its value into the name). `n` is a counter that signifies how often a literal of this value previously occurred in the current section. Literals follow the standard rules for localization by sections. It is therefore absolutely necessary to place literals that were generated in a certain section before the section is terminated!

The numbering with `n` is necessary because a literal may occur multiple times in a section. One reason for this situation is that PC-relative addressing only allows positive offsets; Literals that have once been placed with an `LTORG` can therefore not be referenced in the code that follows. The other reason is that the displacement is generally limited in length (512 resp. 1024 bytes).

An automatic `LTORG` at the end of a program or previously to switching to a different target CPU does not occur; if AS detects unplaced literals in such a situation, an error message is printed.

As the PC-relative addressing mode uses the address of the current instruction plus 4, it is not possible to access a literal that is stored directly after the instruction, like in the following example:

```
        mov     #$1234,r6
        ltorg
```

This is a minor item since the CPU anyway would try to execute the following data as code. Such a situation should not occur in a real program...another

pitfall is far more real: if PC-relative addressing occurs just behind a delayed branch, the program counter is already set to the destination address, and the displacement is computed relative to the branch target plus 2. Following is an example where this detail leads to a literal that cannot be addressed:

```
        bra     Target
        mov     #$12345678,r4        ; is executed
        .
        .
        ltorg                        ; here is the literal
        .
        .
Target: mov     r4,r7                ; execution continues here
```

As `Target`+2 is on an address behind the literal, a negative displacement would result. Things become especially hairy when one of the branch instructions `JMP, JSR, BRAF, or BSRF` is used: as AS cannot calculate the target address (it is generated at runtime from a register's contents), a PC value is assumed that should never fit, effectively disabling any PC-relative addressing at this point.

It is not possible to deduce the memory usage from the count and size of literals. AS might need to insert a padding word to align a long word to an address that is evenly divisible by 4; on the other hand, AS might reuse parts of a 32-bit literal for other 16-bit literals. Of course multiple use of a literal with a certain value will create only one entry. However, such optimizations are completely suppressed for forward references as AS does not know anything about their value.

As literals use the PC-relative addressing which is only allowed for the `MOV` instruction, the usage of literals is also limited to `MOV` instructions. The way AS uses the operand size is a bit tricky: A specification of a byte or word move means to generate the shortest possible instruction that results in the desired value placed in the register's lowest 8 resp. 16 bits. The upper 24 resp. 16 bits are treated as "don't care". However, if one specifies a longword move or omits the size specification completely, this means that the complete 32-bit register should contain the desired value. For example, in the following sequence

```
mov.b   #$c0,r0
mov.w   #$c0,r0
mov.l   #$c0,r0   ,
```

the first instruction will result in true immediate addressing, the second and third instruction will use a word literal: As bit 7 in the number is set, the byte instruction will effectively create the value $FFFFFFC0 in the register. According to the convention, this wouldn't be the desired value in the second and third example. However, a word literal is also sufficient for the third case because the processor will copy a cleared bit 15 of the operand to bits 16..31.

As one can see, the whole literal stuff is rather complex; I'm sorry but there was no chance of making things simpler. It is unfortunately a part of its nature that one sometimes gets error messages about literals that were not found, which logically should not occur because AS does the literal processing completely on his own. However, if other errors occur in the second pass, all following labels will move because AS does not generate any code any more for statements that have been identified as erroneous. As literal names are partially built from other symbols' values, other errors might follow because literal names searched in the second pass differ from the names stored in the first pass and AS quarrels about undefined symbols...if such errors should occur, please correct all other errors first before you start cursing on me and literals...

People who come out of the Motorola scene and want to use PC-relative addressing explicitly (e.g. to address variables in a position-independent way) should know that if this addressing mode is written like in the programmer's manual:

```
mov.l   @(Var,PC),r8
```

no implicit conversion of the address to a displacement will occur, i.e. the operand is inserted as-is into the machine code (this will probably generate a value range error...). If you want to use PC-relative addressing on the SH7x00, simply use "absolute" addressing (which does not exist on machine level):

```
mov.l   Var,r8
```

In this example, the displacement will be calculated correctly (of course, the same limitations apply for the displacement as it was the case for literals).

4.6 MELPS-4500

The program memory of these microcontrollers is organized in pages of 128 words. Honestly said, this organization only exists because there are on the one hand branch instructions with a target that must lie within the same page, and on the other hand "long" branches that can reach the whole address space. The standard syntax defined by Mitsubishi demands that page number and offset have to be written as two distinct arguments for the latter instructions. As this is quite inconvenient (except for indirect jumps, a programmer has no other reason to deal with pages), AS also allows to write the target address in a "linear" style, for example

```
        bl      $1234
```

instead of

```
        bl      $24,$34 .
```

4.7 6502UNDOC

Since the 6502's undocumented instructions naturally aren't listed in any data book, they shall be listed shortly at this place. Of course, you are using them on your own risk. There is no guarantee that all mask revisions will support all variants! They anyhow do not work for the CMOS successors of the 6502, since they allocated the corresponding bit combinations with "official" instructions...

The following symbols are used:

&	binary AND
—	binary OR
^	binary XOR
<<	logical shift left
>>	logical shift right
<<<	rotate left
>>>	rotate right
←	assignment

(..)	contents of ..
..	bits ..
A	accumulator
X,Y	index registers X,Y
S	stack pointer
An	accumulator bit n
M	operand
C	carry
PCH	upper half of program counter

Instruction : `JAM` or `KIL` or `CRS`
Function : none, prozessor is halted
Addressing Modes : implicit

Instruction : `SLO`
Function : $M \leftarrow ((M) << 1)|(A)$
Addressing Modes : absolute long/short, X-indexed long/short, Y-indexed long, X/Y-indirect

Instruction : `ANC`
Function : $A \leftarrow (A)\&(M), C \leftarrow A7$
Addressing Modes : immediate

Instruction : `RLA`
Function : $M \leftarrow ((M) << 1)\&(A)$
Addressing Modes : absolute long/short, X-indexed long/short, Y-indexed long, X/Y-indirect

Instruction : `SRE`
Function : $M \leftarrow ((M) >> 1)\hat{}(A)$
Addressing Modes : absolute long/short, X-indexed long/short, Y-indexed long, X/Y-indirect

Instruction : `ASR`
Function : $A \leftarrow ((A)\&(M)) >> 1$
Addressing Modes : immediate

Instruction : `RRA`
Function : $M \leftarrow ((M) >>> 1) + (A) + (C)$
Addressing Modes : absolute long/short, X-indexed long/short, Y-indexed long, X/Y-indirect

Instruction : `ARR`
Function : $A \leftarrow ((A)\&(M)) >>> 1$
Addressing Modes : immediate

Instruction : `SAX`
Function : $M \leftarrow (A)\&(X)$
Addressing Modes : absolute long/short, Y-indexed short, Y-indirect

Instruction : `ANE`
Function : $M \leftarrow ((A)\&\$ee)|((X)\&(M))$
Addressing Modes : immediate

Instruction : `SHA`
Function : $M \leftarrow (A)\&(X)\&(PCH + 1)$
Addressing Modes : X/Y-indexed long

Instruction : `SHS`
Function : $X \leftarrow (A)\&(X), S \leftarrow (X), M \leftarrow (X)\&(PCH + 1)$
Addressing Modes : Y-indexed long

Instruction : `SHY`
Function : $M \leftarrow (Y)\&(PCH+1)$
Addressing Modes : Y-indexed long

Instruction : `SHX`
Function : $M \leftarrow (X)\&(PCH+1)$
Addressing Modes : X-indexed long

Instruction : `LAX`
Function : $A, X \leftarrow (M)$
Addressing Modes : absolute long/short, Y-indexed long/short, X/Y-indirect

Instruction : `LXA`
Function : $X04 \leftarrow (X)04\&(M)04$, $A04 \leftarrow (A)04\&(M)04$
Addressing Modes : immediate

Instruction : `LAE`
Function : $X, S, A \leftarrow ((S)\&(M))$
Addressing Modes : Y-indexed long

Instruction : `DCP`
Function : $M \leftarrow (M)-1, Flags \leftarrow ((A)-(M))$
Addressing Modes : absolute long/short, X-indexed long/short, Y-indexed long, X/Y-indirect

Instruction : `SBX`
Function : $X \leftarrow ((X)\&(A))-(M)$
Addressing Modes : immediate

Instruction : `ISB`
Function : $M \leftarrow (M)+1, A \leftarrow (A)-(M)-(C)$
Addressing Modes : absolute long/short, X-indexed long/short, Y-indexed long, X/Y-indirect

4.8 MELPS-740

Microcontrollers of this family have a quite nice, however well-hidden feature: If one sets bit 5 of the status register with the `SET` instruction, the accumulator will be replaced with the memory cell addressed by the X register for all load/store and arithmetic instructions. An attempt to integrate this feature cleanly into the assembly syntax has not been made so far, so the only way to use it is currently the "hard" way (`SET`...instructions with accumulator addressing...`CLT`).

Not all MELPS-740 processors implement all instructions. This is a place where the programmer has to watch out for himself that no instructions are used that are unavailable for the targeted processor; AS does not differentiate among the individual processors of this family. For a description of the details regarding special page addressing, see the discussion of the `ASSUME` instruction.

4.9 MELPS-7700/65816

As it seems, these two processor families took disjunct development paths, starting from the 6502 via their 8 bit predecessors. Shortly listed, the following differences are present:

- The 65816 does not have a B accumulator.
- The 65816 does not have instructions to multiply or divide.
- The 65816 misses the instructions `SEB, CLB, BBC, BBS, CLM, SEM, PSH, PUL` and `LDM`. Instead, the instructions `TSB, TRB, BIT, CLD, SED, XBA, XCE` and `STZ` take their places in the opcode table.

The following instructions have identical function, yet different names:

65816	MELPS-7700	65816	MELPS-7700
REP	CLP	PHK	PHG
TCS	TAS	TSC	TSA
TCD	TAD	TDC	TDA
PHB	PHT	PLB	PLT
WAI	WIT		

Especially tricky are the instructions `PHB, PLB` and `TSB`: these instructions have a totally different encoding and meaning on both processors!

Unfortunately, these processors address their memory in a way that is IMHO even one level higher on the open-ended chart of perversity than the Intel-like segmentation: They do banking! Well, this seems to be the price for the 6502 upward-compatibility; before one can use AS to write code for these processors, one has to inform AS about the contents of several registers (using the `ASSUME` instruction):

The M flag rules whether the accumulators A and B should be used with 8 bits (1) or 16 bits (0) width. Analogously, the X flag decides the width of the X and Y index registers. AS needs this information for the decision about the argument's width when immediate addressing (`#<constant>`) occurs.

The memory is organized in 256 banks of 64 KBytes. As all registers in the CPU core have a maximum width of 16 bits, the upper 8 bits have to be fetched from 2 special bank registers: DT delivers the upper 8 bits for data accesses, and PG extends the 16-bit program counter to 24 bits. A 16 bits wide register DPR allows to move the zero page known from the 6502 to an arbitrary location in the first bank. If AS encounters an address (it is irrelevant if this address is part of an absolute, indexed, or indirect expression), the following addressing modes will be tested:

1. Is the address in the range of DPR..DPR+$ff? If yes, use direct addressing with an 8-bit address.

2. Is the address contained in the page addressable via DT (resp. PG for branch instructions)? If yes, use absolute addressing with a 16-bit address.

3. If nothing else helps, use long addressing with a 24-bit address.

As one can see from this enumeration, the knowledge about the current values of DT, PG and DPR is essential for a correct operation of AS; if the specifications are incorrect, the program will probably do wrong addressing at runtime. This enumeration also implied that all three address lengths are available; if this is not the case, the decision chain will become shorter.

The automatic determination of the address length described above may be overridden by the usage of prefixes. If one prefixes the address by a <, >, or >> without a separating space, an address with 1, 2, or 3 bytes of length will be used, regardless if this is the optimal length. If one uses an address length that is either not allowed for the current instruction or too short for the address, an error message is the result.

To simplify porting of 6502 programs, AS uses the Motorola syntax for hexadecimal constants instead of the Intel/IEEE syntax that is the format preferred by Mitsubishi for their 740xxx series. I still think that this is the better format, and it looks as if the designers of the 65816 were of the same opinion (as the `RELAXED` instruction allows the alternative use of Intel notation, this decision should not hurt anything). Another important detail for the porting of programs is that it is valid to omit the accumulator A as target for operations. For example, it is possible to simply write `LDA #0` instead of `LDA A,#0`.

A real goodie in the instruction set are the instructions `MVN` resp. `MVP` to do block transfers. However, their address specification rules are a bit strange: bits 0–15 are stored in index registers, bits 16–23 are part of the instruction. When one uses AS, one simply specifies the full destination and source addresses. AS will then automatically grab the correct bits. This is a fine yet important difference Mitsubishi's assembler where you have to extract the upper 8 bits on your own. Things become really convenient when a macro like the following is used:

```
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
```

Caution, possible pitfall: if the accumulator contains the value n, the Mitsubishi chip will transfer n bytes, but the 65816 will transfer n+1 bytes!

The `PSH` and `PUL` instructions are also very handy because they allow to save a user-defined set to be saved to the stack resp. to be restored from the stack. According to the Mitsubishi data book [41], the bit mask has to be specified as an immediate operand, so the programmer either has to keep all bit↔register assignments in mind or he has to define some appropriate symbols. To make things simpler, I decided to extend the syntax at this point: It is valid to use a list as argument which may contain an arbitrary sequence of register names or immediate expressions. Therefore, the following instructions

```
        psh     #$0f
        psh     a,b,#$0c
        psh     a,b,x,y
```

are equivalent. As immediate expressions are still valid, AS stays upward compatible to the Mitsubishi assemblers.

One thing I did not fully understand while studying the Mitsubishi assembler is the treatment of the `PER` instruction: this instruction allows to push a 16-bit variable onto the stack whose address is specified relative to the program counter. Therefore, it is an absolute addressing mode from the programmer's point of view. Nevertheless, the Mitsubishi assembler requests immediate addressing, and the instructions argument is placed into the code just as-is. One has to calculate the address in his own, which is something symbolic assemblers were designed for to avoid...as I wanted to stay compatible, AS contains a compromise: If one chooses immediate addressing (with a leading # sign), AS will behave like the original from Mitsubishi. But if the # sign is omitted, as will calculate the difference between the argument's value and the current program counter and insert this difference instead.

A similar situation exists for the `PEI` instruction that pushes the contents of a 16-bit variable located in the zero page: Though the operand represents an address, once again immediate addressing is required. In this case, AS will simply allow both variants (i.e. with or without a # sign).

4.10 M16

The M16 family is a family of highly complex CISC processors with an equally complicated instruction set. One of the instruction set's properties is the detail that in an instruction with two operands, both operands may be of different sizes. The method of appending the operand size as an attribute of the instruction (known from Motorola and adopted from Mitsubishi) therefore had to be extended: it is valid to append attributes to the operands themselves. For example, the following instruction

```
mov     r0.b,r6.w
```

reads the lowest 8 bits of register 0, sign-extends them to 32 bits and stores the result into register 6. However, as one does not need this feature in 9 out of 10 cases, it is still valid to append the operand size to the instruction itself, e.g.

```
mov.w   r0,r6
```

Both variants may be mixed; in such a case, an operand size appended to an operand overrules the "default". An exception are instructions with two operands. For these instructions, the default for the source operand is the destination operand's size. For example, in the following example

```
mov.h   r0,r6.w
```

register 0 is accessed with 32 bits, the size specification appended to the instruction is not used at all. If an instruction does not contain any size specifications, word size (`w`) will be used. Remember: in contrast to the 68000 family, this means 32 bits instead of 16 bits!

The chained addressing modes are also rather complex; the ability of AS to automatically assign address components to parts of the chain keeps things at least halfway manageable. The only way of influencing AS allows (the original assembler from Mitsubishi/Green Hills allows a bit more in this respect) is the explicit setting of displacement lengths by appending `:4, :16` and `:32`.

4.11 4004/4040

Thanks to John Weinrich, I now have the official Intel data sheets describing these 'grandfathers' of all microprocessors, and the questions about the syntax of register pairs (for 8-bit operations) have been weeded out for the moment: It is `RnRm` with `n` resp. `m` being even integers in the range from 0 to E resp. 1 to F. The equation `m = n + 1` must be fulfilled.

4.12 MCS-48

The maximum address space of these processors is 4 Kbytes large. This address space is not organized in a linear way (how could this be on an Intel CPU...). Instead, it is split into 2 banks of 2 Kbytes. The only way to change the program counter from one bank to the other are the instructions `CALL` and `JMP`, by setting the most significant bit of the address with the instructions `SEL MB0` resp. `SEL MB1`.

To simplify jumps between these two banks, the instructions `JMP` and `CALL` contain an automatism that inserts one of these two instructions if the current program counter and the target address are in different banks. Explicit usage of these `SEL MBx` instructions should therefore not be necessary (though it is possible), and it can puzzle the automatism, like in the following example:

```
000:  SEL      MB1
      JMP      200h
```

AS assumes that the MB flag is 0 and therefore does not insert a `SEL MB0` instruction, with the result that the CPU jumps to address A00h.

Furthermore, one should keep in mind that a jump instruction might become longer (3 instead of 2 bytes).

4.13 MCS-51

The assembler is accompanied by the files `STDDEF51.INC` resp. `80C50X.INC` that define all bits and SFRs of the processors 8051, 8052, and 80515 resp.

80C501, 502, and 504. Depending on the target processor setting (made with the CPU statement), the correct subset will be included. Therefore, the correct order for the instructions at the beginning of a program is

```
CPU     <processor type>
INCLUDE stddef51.inc    .
```

Otherwise, the MCS-51 pseudo instructions will lead to error messages.

As the 8051 does not have instructions to to push the registers 0..7 onto the stack, one has to work with absolute addresses. However, these addresses depend on which register bank is currently active. To make this situation a little bit better, the include files define the macro USING that accepts the symbols Bank0...Bank3 as arguments. In response, the macro will assign the registers' correct absolute addresses to the symbols AR0..AR7. This macro should be used after every change of the register banks. The macro itself does **not** generate any code to switch to the bank!

The macro also makes bookkeeping about which banks have been used. The result is stored in the integer variable RegUsage: bit 0 corresponds to bank 0, bit 1 corresponds to bank 1. and so on. To output its contents after the source has been assembled, use something like the following piece of code:

```
irp       BANK,Bank0,Bank1,Bank2,Bank3
 if        (RegUsage&(2^BANK))<>0
  message   "bank \{BANK} has been used"
 endif
endm
```

The multipass feature introduced with version 1.38 allowed to introduce the additional instructions JMP and CALL. If branches are coded using these instructions, AS will automatically use the variant that is optimal for the given target address. The options are SJMP, AJMP, or LJMP for JMP resp. ACALL or LCALL for CALL. Of course it is still possible to use these variants directly, in case one wants to force a certain coding.

4.14 MCS-251

When designing the 80C251, Intel really tried to make the move to the new family as smooth as possible for programmers. This culminated in the fact

that old applications can run on the new processor without having to re-compile them. However, as soon as one wants to use the new features, some details have to be regarded which may turn into hidden pitfalls.

The most important thing is the absence of a distinct address space for bits on the 80C251. All SFRs can now be addressed bitwise, regardless of their address. Furthermore, the first 128 bytes of the internal RAM are also bit addressable. This has become possible because bits are not any more handled by a separate address space that overlaps other address spaces. Instead, similar to other processors, bits are addressed with a two-dimensional address that consists of the memory location containing the bit and the bit's location in the byte. One result is that in an expression like `PSW.7`, AS will do the separation of address and bit position itself. Unlike to the 8051, it is not any more necessary to explicitly generate 8 bit symbols. This has the other result that the `SFRB` instruction does not exist any more. If it is used in a program that shall be ported, it may be replaced with a simple `SFR` instruction.

Furthermore, Intel cleaned up the cornucopia of different address spaces on the 8051: the internal RAM (`DATA` resp. `IDATA`), the `XDATA` space and the former `CODE` space were unified to a single `CODE` space that is now 16 Mbytes large. The internal RAM starts at address 0, the internal ROM starts at address ff0000h, which is the address code has to be relocated to. In contrast, the SFRs were moved to a separate address space (which AS refers to as the `IO` segment). However, they have the same addresses in this new address space as they used to have on the 8051. The `SFR` instructions knows of this difference and automatically assigns symbols to either the `DATA` or `IO` segment, depending on the target processor. As there is no `BIT` segment any more, the `BIT` instruction operates completely different: Instead of a linear address ranging from 0..255, a bit symbol now contains the byte's address in bit 0..7, and the bit position in bits 24..26. Unfortunately, creating arrays of flags with a symbolic address is not that simple any more: On an 8051, one simply wrote:

```
        segment bitdata

bit1    db      ?
bit2    db      ?

or
```

```
defbit  macro   name
name    bit     cnt
cnt     set     cnt+1
        endm
```

On a 251, only the second way still works, like this:

```
adr     set     20h     ; start address of flags
bpos    set     0       ; in the internal RAM

defbit  macro   name
name    bit     adr.bpos
bpos    set     bpos+1
        if      bpos=8
bpos     set     0
adr      set     adr+1
        endif
        endm
```

Another small detail: Intel now prefers `CY` instead of `C` as a symbolic name for the carry, so you might have to rename an already existing variable of the same name in your program. However, AS will continue to understand also the old variant when using the instructions `CLR, CPL, SETB, MOV, ANL,` or `ORL`. The same is conceptually true for the additional registers `R8..R15, WR0..WR30, DR0..DR28, DR56, DR60, DPX,` and `SPX`.

Intel would like everyone to write absolute addresses in a syntax of `XX:YYYY`, where `XX` is a 64K bank in the address space resp. signifies addresses in the I/O space with an `S`. As one might guess, I am not amused about this, which is why it is legal to alternitavely use linear addresses in all places. Only the `S` for I/O addresses is incircumventable, like in this case:

```
Carry   bit     s:0d0h.7
```

Without the prefix, AS would assume an address in the `CODE` segment, and only the first 128 bits in this space are bit-addressable...

Like for the 8051, the generic branch instructions `CALL` and `JMP` exist that automatically choose the shortest machine code depending on the address

layout. However, while JMP also may use the variant with a 24-bit address, CALL will not do this for a good reason: In contrast to ACALL and LCALL, ECALL places an additional byte onto the stack. A CALL instruction would result where you would not know what it will do. This problem does not exist for the JMP instructions.

There is one thing I did not understand: The 80251 is also able to push immediate operands onto the stack, and it may push either single bytes or complete words. However, the same mnemonic (PUSH) is assigned to both variants - how on earth should an assembler know if an instruction like

```
push    #10
```

shall push a byte or a word containing the value 10? So the current rule is that PUSH always pushes a byte; if one wants to push a word, simply use PUSHW instead of PUSH.

Another well-meant advise: If you use the extended instruction set, be sure to operate the processor in source mode; otherwise, all instructions will become one byte longer! The old 8051 instructions that will in turn become one byte longer are not a big matter: AS will either replace them automatically with new, more general instructions or they deal with obsolete addressing modes (indirect addressing via 8 bit registers).

4.15 8085UNDOC

Similarly to the Z80 or 6502, Intel did not further specify the undocumented 8085 instructions. This however means that other assemblers might use different mnemonics for the same function. Therefore, I will list the nistructions in the following. Once again, usage of these instructions is at one's own risk - even the Z80 which is principally upward compatible to the 8085 uses the opcodes for entirely different functions...

Instruction : DSUB [reg]
Function : HL ← HL - reg
Flags : CY, S, X5, AC, Z, V, P
Arguments : reg = B for BC (optional)

Instruction : `ARHL`
Function : HL,CY ← HL >> 1 (arithmetisch)
Flags : CY
Arguments : none

Instruction : `RDEL`
Function : CY,DE ← DE << 1
Flags : CY, V
Arguments : none

Instruction : `LDHI d8`
Function : DE ← HL + `d8`
Flags : none
Arguments : `d8` = 8-bit constant

Instruction : `LDSI d8`
Function : DE ← SP + `d8`
Flags : none
Arguments : `d8` = 8-bit constant

Instruction : `RST flag`
Function : restart to 40h if `flag`=1
Flags : none
Arguments : `flag` = V for overflow bit

Instruction : `SHLX [reg]`
Function : [reg] ← HL

Flags : none
Arguments : `reg` = D for DE (optional)

Instruction : `LHLX [reg]`
Function : HL ← [reg]
Flags : none
Arguments : `reg` = D for DE (optional)

Instruction : `JNX5 addr`
Function : jump to `addr` if X5=0
Flags : none
Arguments : `addr` = absolute 16-bit address

Instruction : `JX5 addr`
Function : jump to `addr` if X5=1
Flags : none
Arguments : `addr` = absolute 16-bit address

Mit X5 ist dabei das ansonsten unbenutzte Bit 5 im PSW-Register gemeint.

4.16 8086..V35

Actually, I had sworn myself to keep the segment disease of Intel's 8086 out of the assembler. However, as there was a request and as students are more flexible than the developers of this processor obviously were, there is now a rudimentary support of these processors in AS. When saying, 'rudimentary', it does not mean that the instruction set is not fully covered. It means that the whole pseudo instruction stuff that is available when using MASM, TASM, or something equivalent does not exist. To put it in clear words, AS

was not primarily designed to write assembler programs for PC's (heaven forbid, this really would have meant reinventing the wheel!); instead, the development of programs for single-board computers was the main goal (which may also be equipped with an 8086 CPU).

For die-hards who still want to write DOS programs with AS, here is a small list of things to keep in mind:

- Only `COM` files may be created.
- Only use the `CODE` segment, and place also all variables in this segment.
- DOS initializes all segment registers to the code segment. An `ASSUME DS:DATA, SS:DATA` right at the program's beginning is therefore necessary.
- DOS loads the code to a start address of 100h. An `ORG` to this address is absolutely necessary.
- The conversion to a binary file is done with P2BIN (see later in this document), with an address filter of `$-$`.

For these processors, AS only supports a small programming model, i.e. there is **one** code segment with a maximum of 64 Kbytes and a data segment of equal size for data (which cannot be set to initial values for `COM` files). The `SEGMENT` instruction allows to switch between these two segments. From this facts results that branches are always intrasegment branches if they refer to targets in this single code segment. In case that far jumps should be necessary, they are possible via `CALLF` or `JMPF` with a memory address or a `Segment:Offset` value as argument.

Another big problem of these processors is their assembler syntax, which is sometimes ambiguous and whose exact meaning can then only be deduced by looking at the current context. In the following example, either absolute or immediate addressing may be meant, depending on the symbol's type:

```
        mov     ax,value
```

assembler	address	contents

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]`

Table 4.1: Differences AS↔MASM Concerning Addressing Syntax

When using AS, an expression without brackets always is interpreted as immediate addressing. For example, when either a variable's address or its contents shall be loaded, the differences listed in table 4.1 are present between MASM and AS:

When addressing via a symbol, the assembler checks whether they are assigned to the data segment and tries to automatically insert an appropriate segment prefix. This happens for example when symbols from the code segment are accessed without specifying a `CS` segment prefix. However, this mechanism can only work if the `ASSUME` instruction (see there) has previously been applied correctly.

The Intel syntax also requires to store whether bytes or words were stored at a symbol's address. AS will do this only when the `DB` resp. `DW` instruction is in the same source line as the label. For any other case, the operand size has to be specified explicitly with the `BYTE PTR, WORD PTR,...` operators. As long as a register is the other operator, this may be omitted, as the operand size is then clearly given by the register's name.

In an 8086-based system, the coprocessor is usually synchronized via via the processor's TEST input line which is connected to toe coprocessor's BUSY output line. AS supports this type of handshaking by automatically inserting a `WAIT` instruction prior to every 8087 instruction. If this is undesired for any reason, an `N` has to be inserted after the `F` in the mnemonic; for example,

```
FINIT
FSTSW   [vari]
```

becomes

```
FNINIT
FNSTSW  [vari]
```

This variant is valid for **all** coprocessor instructions.

4.17 8X30x

The processors of this family have been optimized for an easy manipulation of bit groups at peripheral addresses. The instructions `LIV` and `RIV` were introduced to deal with such objects in a symbolic fashion. They work similar to `EQU`, however they need three parameters:

1. the address of the peripheral memory cell that contains the bit group (0..255);
2. the number of the group's first bit (0..7);
3. the length of the group, expressed in bits (1..8).

CAUTION! The 8X30x does not support bit groups that span over more than one memory address. Therefore, the valid value range for the length can be stricter limited, depending on the start position. AS does **not** perform any checks at this point, you simply get strange results at runtime!

Regarding the machine code, length and position are expressed vis a 3 bit field in the instruction word and a proper register number (`LIVx` resp. `RIVx`). If one uses a symbolic object, AS will automatically assign correct values to this field, but it is also allowed to specify the length explicitly as a third operand if one does not work with symbolic objects. If AS finds such a length specification in spite of a symbolic operand, it will compare both lengths and issue an error if they do not match (the same will happen for the MOVE instruction if two symbolic operands with different lengths are used - the instruction simply only has a single length field...).

Apart from the real machine instructions, AS defines similarly to its "idol" MCCAP some pseudo instructions that are implemented as builtin macros:

- `NOP` is a shortform for `MOVE AUX,AUX`

- HALT is a shortform for JMP *
- XML ii is a shortform for XMIT ii,R12 (only 8X305)
- XMR ii is a shortform for XMIT ii,R13 (only 8X305)
- SEL <busobj> is a shortform for XMIT <adr>,IVL/IVR, i.e. it performs the necessary preselection to access <busobj>.

The CALL and RTN instructions MCCAP also implements are currently missing due to sufficient documentation. The same is true for a set of pseudo instructions to store constants to memory. Time may change this...

4.18 XA

Similar to its predecessor MCS/51, but in contrast to its 'competitor' MCS/251, the Philips XA has a separate address space for bits, i.e. all bits that are accessible via bit instructions have a certain, one-dimensional address which is stored as-is in the machine code. However, I could not take the obvious opportunity to offer this third address space (code and data are the other two) as a separate segment. The reason is that - in contrast to the MCS/51 - some bit addresses are ambiguous: bits with an address from 256 to 511 refer to the bits of memory cells 20h..3fh in the current data segment. This means that these addresses may correspond to different physical bits, depending on the current state. Defining bits with the help of DC instructions - something that would be possible with a separate segment - would not make too much sense. However, the BIT instruction still exists to define individual bits (regardless if they are located in a register, the RAM or SFR space) that can then be referenced symbolically. If the bit is located in RAM, the address of the 64K-bank is also stored. This way, AS can check whether the DS register has previously be assigned a correct value with an ASSUME instruction.

In contrast, nothing can stop AS's efforts to align potential branch targets to even addresses. Like other XA assemblers, AS does this by inserting NOPs right before the instruction in question.

4.19 AVR

In contrast to the AVR assembler, AS by default uses the Intel format to write hexadecimal contants instead of the C syntax. All right, I did not look into the (free) AVR assembler before, but when I started with the AVR part, there was hardly mor einformation about the AVR than a preliminary manual describing processor types that were never sold...this problem can be solved with a simple RELAXED ON.

Optionally, AS can generate so-called "object files" for the AVRs (it also works for other CPUs, but it does not make any sense for them...). These are files containing code and source line info what e.g. allows a step-by-step execution on source level with the WAVRSIM simulator delivered by Atmel. Unfortunately, the simulator seems to have trouble with source file names longer than approx. 20 characters: Names are truncated and/or extended by strange special characters when the maximum length is exceeded. AS therefore stores file name specifications in object files without a path specification. Therefore, problems may arise when files like includes are not in the current directory.

A small specialty are machine instructions that have already been defined by Atmel as part of the architecture, but up to now haven't been implemented in any of the family's members. The instructions in question are `MUL, JMP,` and `CALL`. Considering the latter ones, one may ask himself how to reach the 4 Kwords large address space of the AT90S8515 when the 'next best' instructions `RJMP` and `RCALL` can only branch up to 2 Kwords forward or backward. The trick is named 'discarding the upper address bits' and described in detail with the `WRAPMODE` statement.

4.20 Z80UNDOC

As one might guess, Zilog did not make any syntax definitions for the undocumented instructions; furthermore, not everyone might know the full set. It might therefore make sense to list all instructions at this place:

Similar to a Z380, it is possible to access the byte halves of IX and IY separately. In detail, these are the instructions that allow this:

```
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
```

`Rx` and `Ry` are synonyms for `IXL, IXU, IYL` or `IYU`. Keep however in mind that in the case of `LD Rx,Ry`, both registers must be part of the same index register.

The coding of shift instructions leaves an undefined bit combination which is now accessible as the `SLIA` instruction. `SLIA` works like `SLA` with the difference of entering a 1 into bit position 0. Like all other shift instructions, `SLIA` also allows another undocumented variant:

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

In this case, `R` is an arbitrary 8-bit register (excluding index register halves...), and `(XY+d)` is a normal indexed address. This operation has the additional effect of copying the result into the register. This also works for the `RES` and `SET` instructions:

```
SET/RES R,n,(XY+d)
```

Furthermore, two hidden I/O instructions exist:

```
IN      (C) resp. TSTI
OUT     (C),0
```

Their operation should be clear. **CAUTION!** Noone can guarantee that all mask revisions of the Z80 execute these instructions, and the Z80's successors will react with traps if they find one of these instructions. Use them on your own risk...

4.21 Z380

As this processor was designed as a grandchild of the still most popular 8-bit microprocessor, it was a sine-qua-non design target to execute existing Z80 programs without modification (of course, they execute a bit faster, roughly by a factor of 10...). Therefore, all extended features can be enabled after a reset by setting two bits which are named XM (eXtended Mode, i.e. a

32-bit instead of a 16-bit address space) respectively LW (long word mode, i.e. 32-bit instead of 16-bit operands). One has to inform AS about their current setting with the instructions `EXTMODE` resp. `LWORDMODE`, to enable AS to check addresses and constants against the correct upper limits. The toggle between 32- and 16-bit instruction of course only influences instructions that are available in a 32-bit variant. Unfortunately, the Z380 currently offers such variants only for load and store instructions; arithmetic can only be done in 16 bits. Zilog really should do something about this, otherwise the most positive description for the Z380 would be "16-bit processor with 32-bit extensions"...

The whole thing becomes complicated by the ability to override the operand size set by LW with the instruction prefixes `DDIR W` resp. `DDIR LW`. AS will note the occurrence of such instructions and will toggle setting for the instruction following directly. By the way, one should never explicitly use other `DDIR` variants than `W` resp. `LW`, as AS will introduce them automatically when an operand is discovered that is too long. Explicit usage might puzzle AS. The automatism is so powerful that in a case like this:

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

the necessary `IW` prefix will automatically be merged into the previous instruction, resulting in

```
DDIR    LW,IW
LD      BC,12345668h    .
```

The machine code that was first created for `DDIR LW` is retracted and replaced, which is signified with an `R` in the listing.

4.22 Z8 and eZ8

The CPU core contained in the Z8 microcontrollers does not contain any specific registers. Instead, a block of 16 consecutive cells of the internal address space (contains RAM and I/O registers) may be used as 'work registers' and be addressed with 4-bit addresses. The RP register defines which memory

block is used as work registers: bits 4 to 7 of RP define the 'offset' that is added to every 4-bit work register address to get a complete 8-bit address.

Usually, one refers to work registers as R0..R15 in assembly statements. It is however also posssible to regard work registers as an efficient way to address a block of memory addresses in internal RAM.

The `ASSUME` statement is used to inform AS about the current value of RP. AS is then capable to automatically decide whether an address in internal RAM may be reached with a 4-bit or 8-bit address. This may be used to assign symbolic names to work registers:

```
op1     equ     040h
op2     equ     041h

        srp     #040h
        assume  rp:040h

        ld      op1,op2         ; equal to ld r0,r1
```

Opposed to the original Zilog assembler, it is not necessary to explicitly specify 'work register addressing' with a prefixed exclamation mark. AS however also understands this syntax - a prefixed exclamation mark enforces 4-bit addressing, even when the address does not lie within the 16-address block defined by RP (AS will issue a warning in that case). Vice versa, a prefixed > character enforces 8-bit addressing even when the address is within the current 16-address block.

The eZ8 takes this 'game' to the next level: the internal address space now has 12 instead of 8 bits. To assure compatibility with the old Z8 core, Zilog placed the additional 4 bits in the *lower* four bits of RP. For instance, an RP value of 12h defines an address window from 210h to 21fh.

At the same time, the lower four bits of RP define a window of 256 addresses that can be addressed with 8-bit addresses. The mechanism to automatically select between 8- and 12-bit addresses is analogous. 'Long' 12-bit addresses may be enforced by prefixing two > characters.

4.23 TLCS-900(L)

These processors may run in two operating modes: on the one hand, in minimum mode, which offers almost complete source code compatibility to the Z80 and TLCS-90, and on the other hand in maximum mode, which is necessary to make full use of the processor's capabilities. The main differences between these two modes are:

- width of the registers WA, BC, DE, and HL: 16 or 32 bits;
- number of register banks: 8 or 4;
- code address space: 64 Kbytes or 16 Mbytes;
- length of return addresses: 16 or 32 bits.

To allow AS to check against the correct limits, one has to inform him about the current execution mode via the `MAXMODE` instruction (see there). The default is the minimum mode.

From this follows that, depending on the operating mode, the 16-bit resp. 32-bit versions of the bank registers have to be used for addressing, i.e. WA, BC, DE and HL for the minimum mode resp. XWA, XBC, XDE and XHL for the maximum mode. The registers XIX..XIZ and XSP are **always** 32 bits wide and therefore always have to to be used in this form for addressing; in this detail, existing Z80 code definitely has to be adapted (not including that there is no I/O space and all I/O registers are memory-mapped...).

The syntax chosen by Toshiba is a bit unfortunate in the respect of choosing an single quote (') to reference the previous register bank. The processor independent parts of AS already use this character to mark character constants. In an instruction like

```
ld      wa',wa   ,
```

AS will not recognize the comma for parameter separation. This problem can be circumvented by usage of an inverse single quote (‘), for example

```
ld      wa‘,wa
```

Toshiba delivers an own assembler for the TLCS-900 series (TAS900), which is different from AS in the following points:

Symbol Conventions

- TAS900 differentiates symbol names only on the first 32 characters. In contrast, AS always stores symbol names with the full length (up to 255 characters) and uses them all for differentiation.
- TAS900 allows to write integer constants either in Intel or C notation (with a 0 prefix for octal or a 0x prefix for hexadecimal constants). By default, AS only supports the Intel notation. With the help of the `RELAXED` instruction, one also gets the C notation (among other).
- AS does not distinguish between upper and lower case. In contrast, TAS900 differentiates between upper- and lowercase letters in symbol names. One needs to engage the `-u` command line option to force AS to do this.

Syntax

For many instructions, the syntax checking of AS is less strict than the checking of TAS900. In some (rare) cases, the syntax is slightly different. These extensions and changes are on the one hand for the sake of a better portability of existing Z80 codes, on the other hand they provide a simplification and better orthogonality of the assembly syntax:

- In the case of `LDA, JP`, and `CALL`, TAS requires that address expressions like `XIX+5` must not be placed in parentheses, as it is usually the case. For the sake of better orthogonality, AS requires parentheses for `LDA`. They are optional if `JP` resp. `CALL` are used with a simple, absolute address.
- In the case of `JP, CALL, JR`, and `SCC`, AS leaves the choice to the programmer whether to explicitly write out the default condition `T` (= true) as first parameter or not. TAS900 in contrast only allows to use the default condition implicitly (e.g. `jp (xix+5)` instead of `jp t,(xix+5))`.
- For the `EX` instruction, AS allows operand combinations which are not listed in [106] but can be reduced to a standard combination by swapping the operands. Combinations like `EX f',f` or `EX wa,(xhl)` become possible. In contrast, TAS900 limits to the 'pure' combinations.

- AS allows to omit an increment resp. decrement of 1 when using the instructions `INC` and `DEC`. TAS900 instead forces the programmer to explicit usage of '1'.

- The similar is true for the shift instructions: If the operand is a register, TAS900 requires that even a shift count of 1 has to be written explicitly; however, when the operand is in memory, the hardware limits the shift count to 1 which must not be written in this case. With AS, a shift count of 1 is always optional and valid for all types of operands.

Macro Processor

The macro processor of TAS900 is an external program that operates like a preprocessor. It consists of two components: The first one is a C-like preprocessor, and the second one is a special macro language (MPL) that reminds of high level languages. The macro processor of AS instead is oriented towards "classic" macro assemblers like MASM or M80 (both programs from Microsoft). It is a fixed component of AS.

Output Format

TAS900 generates relocatable code that allows to link separately compiled programs to a single application. AS instead generates absolute machine code that is not linkable. There are currently no plans to extend AS in this respect.

Pseudo Instructions

Due to the missing linker, AS lacks a couple of pseudo instructions needed for relocatable code TAS900 implements. The following instructions are available with equal meaning:

```
EQU, DB, DW, ORG, ALIGN, END, TITLE, SAVE, RESTORE
```

TAS900	AS	meaning/function
DL <Data>	DD <Data>	define longword constants
DSB <number>	DB <number> DUP (?)	reserve bytes of memory
DSW <number>	DW <number> DUP (?)	reserve words of memory
DSD <number>	DD <number> DUP (?)	reserve longwords of memory
$MIN[IMUM]	MAXMODE OFF	following code runs in minimum mode
$MAX[IMUM]	MAXMODE ON	following code runs in maximum mode
$SYS[TEM]	SUPMODE ON	following code runs in system mode
$NOR[MAL]	SUPMODE OFF	following code runs in user mode
$NOLIST	LISTING OFF	turn off assembly listing
$LIST	LISTING ON	turn on assembly listing
$EJECT	NEWPAGE	start new page in listing

Table 4.2: equivalent instructions TAS900↔AS

The latter two have an extended functionality for AS. Some TAS900 pseudo instructions can be replaced with equivalent AS instructions (see table 4.2).

Toshiba manufactures two versions of the processor core, with the L version being an "economy version". AS will make the following differences between TLCS-900 and TLCS-900L:

- The instructions MAX and NORMAL are not allowed for the L version; the MIN instruction is disabled for the full version.
- The L version does not know the normal stack pointer XNSP/NSP, but instead has the interrupt nesting register INTNEST.

The instructions SUPMODE and MAXMODE are not influenced, just as their initial setting OFF. The programmer has to take care of the fact that the L version starts in maximum mode and does not have a normal mode. However, AS shows a bit of mercy against the L variant by suppressing warnings for privileged instructions.

4.24 TLCS-90

Maybe some people might ask themselves if I mixed up the order a little bit, as Toshiba first released the TLCS-90 as an extended Z80 and afterwards the 16-bit version TLCS-900. Well, I discovered the '90 via the '900 (thank you Oliver!). The two families are quite similar, not only regarding their syntax but also in their architecture. The hints for the '90 are therefore a subset of of the chapter for the '900: As the '90 only allows shifts, increments, and decrements by one, the count need not and must not be written as the first argument. Once again, Toshiba wants to omit parentheses for memory operands of `LDA, JP, and CALL`, and once again AS requires them for the sake of orthogonality (the exact reason is of course that this way, I saved an extra in the address parser, but one does not say such a thing aloud).

Principally, the TLCS-90 series already has an address space of 1 Mbyte which is however only accessible as data space via the index registers. AS therefore does not regard the bank registers and limits the address space to 64 Kbytes. This should not limit too much as this area above is anyway only reachable via indirect addressing.

4.25 TLCS-870

Once again Toshiba...a company quite productive at the moment! Especially this branch of the family (all Toshiba microcontrollers are quite similar in their binary coding and programming model) seems to be targeted towards the 8051 market: the method of separating the bit position from the address expression with a dot had its root in the 8051. However, it creates now exactly the sort of problems I anticipated when working on the 8051 part: On the one hand, the dot is a legal part of symbol names, but on the other hand, it is part of the address syntax. This means that AS has to separate address and bit position and must process them independently. Currently, I solved this conflict by seeking the dot starting at the **end** of the expression. This way, the last dot is regarded as the separator, and further dots stay parts of the address. I continue to urge everyone to omit dots in symbol names, they will lead to ambiguities:

```
LD      CF,A.7  ; accumulator bit 7 to carry
LD      C,A.7   ; constant 'A.7' to accumulator
```

4.26 TLCS-47

This family of 4-bit microcontrollers should mark the low end of what is supportable by AS. Apart from the `ASSUME` instruction for the data bank register (see there), there is only one thing that is worth mentioning: In the data and I/O segment, nibbles are reserved instead of byte (it's a 4-bitter...). The situation is similar to the bit data segment of the 8051, where a `DB` reserves a single bit, with the difference that we are dealing with nibbles.

Toshiba defined an "extended instruction set" for this processor family to facilitate the work with their limited instruction set. In the case of AS, it is defined in the include file `STDDEF47.INC`. However, some instructions that could not be realized as macros are "builtins" and are therefore also available without the include file:

- the `B` instruction that automatically chooses the optimal version of the jump instruction (`BSS; BS`, or `BSL`);
- `LD` in the variant of `HL` with an immediate operand;
- `ROLC` and `RORC` with a shift amplitude higher than one.

4.27 TLCS-9000

This is the first time that I implemented a processor for AS which was not available at that point of time. Unfortunately, Toshiba decided to put this processor "on ice", so we won't see any silicon in the near future. This has of course the result that this part

1. is a "paper design", i.e. there was so far no chance to test it in the reality and
2. the documentation for the '9000 I could get hold of [109] were preliminary, so they could not deliver clarity on every detail.

Therefore, errors in this code generator are quite possible (and will of course be fixed if it should ever become possible!). At least the few examples listed in [109] are assembled correctly.

4.28 29xxx

As it was already described in the discussion of the `ASSUME` instruction, AS can use the information about the current setting of the RBP register to detect accesses to privileged registers in user mode. This ability is of course limited to direct accesses (i.e. without using the registers IPA...IPC), and there is one more pitfall: as local registers (registers with a number >127) are addressed relative to the stack pointer, but the bits in RBP always refer to absolute numbers, the check is NOT done for local registers. An extension would require AS to know always the absolute value of SP, which would at least fail for recursive subroutines...

4.29 80C16x

As it was already explained in the discussion of the `ASSUME` instruction, AS tries to hide the fact that the processor has more physical than logical RAM as far as possible. Please keep in mind that the DPP registers are valid only for data accesses and only have an influence on absolute addressing, neither on indirect nor on indexed addresses. AS cannot know which value the computed address may take at runtime... The paging unit unfortunately does not operate for code accesses so one has to work with explicit long or short `CALL`s, `JMP`s, or `RET`s. At least for the "universal" instructions `CALL` and `JMP`, AS will automatically use the shortest variant, but at least for the RET one should know where the call came from. `JMPS` and `CALLS` principally require to write segment and address separately, but AS is written in a way that it can split an address on its own, e.g. one can write

```
jmps    12345h
```

instead of

```
jmps    1,2345h
```

Unfortunately, not all details of the chip's internal instruction pipeline are hidden: if CP (register bank address), SP (stack), or one of the paging registers are modified, their value is not available for the instruction immediately

following. AS tries to detect such situations and will issue a warning in such cases. Once again, this mechanism only works for direct accesses.

Bits defined with the `BIT` instruction are internally stored as a 12-bit word, containing the address in bits 4..11 and the bit position in the four LSBs. This order allows to refer the next resp. previous bit by incrementing or decrementing the address. This will however not work for explicit bit specifications when a word boundary is crossed. For example, the following expression will result in a range check error:

```
        bclr    r5.15+1
```

We need a `BIT` in this situation:

```
msb     bit     r5.15
        .
        .
        bclr    msb+1
```

The SFR area was doubled for the 80C167/165/163: bit 12 flags that a bit lies in the second part. Siemens unfortunately did not foresee that 256 SFRs (128 of them bit addressable) would not suffice for successors of the 80C166. As a result, it would be impossible to reach the second SFR area from F000H..F1DFH with short addresses or bit instructions if the developers had not included a toggle instruction:

```
        EXTR    #n
```

This instruction has the effect that for the next `n` instructions ($0 < n < 5$), it is possible to address the alternate SFR space instead of the normal one. AS does not only generate the appropriate machine code when it encounters this instruction. It also sets an internal flag that will only allow accesses to the alternate SFR space for the next `n` instructions. Of course, they may not contain jumps... Of course, it is always possible to define bits from either area at any place, and it is always possible to reach all registers with absolute addresses. In contrast, short and bit addressing only works for one area at a time, attempts contradicting to this will result in an error message.

The situation is similar for prefix instructions and absolute resp. indirect addressing: as the prefix argument and the address expression cannot always be evaluated at assembly time, chances for checking are limited and AS will limit itself to warnings...in detail, the situation is as follows:

- fixed specification of a 64K bank with `EXTS` or `EXTSR`: the address expression directly contains the lower 16 bits of the target address. If the prefix and the following instruction have a constant operand, AS will check if the the prefix argument and bits 16..23 of the target address are equal.

- fixed specification of a 16K page with `EXTP` or `EXTPR`: the address expression directly contains the lower 14 bits of the target address. Bits 14 and 15 are fixed to 0, as the processor ignores them in this mode. If the prefix and the following instruction have a constant operand, AS will check if the the prefix argument and bits 14..23 of the target address are equal.

An example to clarify things a bit (the DPP registers have their reset values):

```
extp    #7,#1       ; range from 112K..128K
mov     r0,1cdefh   ; results in address 0defh in code
mov     r0,1cdefh   ; -->warning
exts    #1,#1       ; range from 64K..128K
mov     r0,1cdefh   ; results in address 0cdefh in code
mov     r0,1cdefh   ; -->warning
```

4.30 PIC16C5x/16C8x

Similar to the MCS-48 family, the PICs split their program memory into several banks because the opcode does not offer enough space for a complete address. AS uses the same automatism for the instructions `CALL` and `GOTO`, i.e. the PA bits in the status word are set according to the start and target address. However, this procedure is far more problematic compared to the 48's:

1. The instructions are not any more one word long (up to three words). Therefore, it is not guaranteed that they can be skipped with a conditional branch.

2. It is possible that the program counter crosses a page boundary while the program sequence is executed. The setting of PA bits AS assumes may be different from reality.

The instructions that operate on register W and another register normally require a second parameter that specifies whether the result shall be stored in W or the register. Under AS, it is valid to omit the second parameter. The assumed target then depends upon the operation's type: For unary operations, the result is by default stored back into the register. These instructions are:

```
COMF, DECF, DECFSZ, INCF, INCFSZ, RLF, RRF, and SWAPF
```

The other operations by default regard W as an accumulator:

```
ADDWF, ANDWF, IORWF, MOVF, SUBWF, and XORWF
```

The syntax defined by Microchip to write literals is quite obscure and reminds of the syntax used on IBM 360/370 systems (greetings from the stone-age...). To avoid introducing another branch into the parser, with AS one has to write constants in the Motorola syntax (optionally Intel or C in `RELAXED` mode).

4.31 PIC 17C4x

With two exceptions, the same hints are valid as for its two smaller brothers: the corresponding include file only contains register definitions, and the problems concerning jump instructions are much smaller. The only exception is the `LCALL` instruction, which allows a jump with a 16-bit address. It is translated with the following "macro":

```
MOVLW   <addr15..8>
MOWF    3
LCALL   <addr0..7>
```

4.32 ST6

These processors have the ability to map their code ROM pagewise into the data area. I am not keen on repeating the whole discussion of the `ASSUME` instruction at this place, so I refer to the corresponding section (3.2.15) for an explanation how to read constants out of the code ROM without too much headache.

Some builtin "macros" show up when one analyzes the instruction set a bit more in detail. The instructions I found are listed in table 4.3 (there are probably even more...):

instruction	in reality
`CLR A`	`SUB A,A`
`SLA A`	`ADD A,A`
`CLR addr`	`LDI addr,0`
`NOP`	`JRZ PC+1`

Table 4.3: Hidden Macros in the ST62's Instruction Set

Especially the last case is a bit astonishing...unfortunately, some instructions are really missing. For example, there is an `AND` instruction but no `OR`...not to speak of an `XOR`. For this reason, the include file `STDDEF62.INC` contains also some helping macros (additionally to register definitions).

The original assembler AST6 delivered by SGS-Thomson partially uses different pseudo instructions than AS. Apart from the fact that AS does not mark pseudo instructions with a leading dot, the following instructions are identical:

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

Table 4.4 shows the instructions which have AS counterparts with similar function.

AST6	AS	meaning/function
.DISPLAY	MESSAGE	output message
.EJECT	NEWPAGE	new page in assembly listing
.ELSE	ELSEIF	conditional assembly
.ENDC	ENDIF	conditional assembly
.IFC	IF...	conditional assembly
.INPUT	INCLUDE	insert include file
.LIST	LISTING, MACEXP	settings for listing
.PL	PAGE	page length of listing
.ROMSIZE	CPU	set target processor
.VERS	VERSION (symbol)	query version
.SET	EVAL	redefine variables

Table 4.4: Equivalent Instructions AST6↔AS

4.33 ST7

In [81], the .w postfix to signify 16-bit addresses is only defined for memory indirect operands. It is used to mark that a 16-bit address is stored at a zero page address. AS additionally allows this postfix for absolute addresses or displacements of indirect address expressions to force 16-bit displacements in spite of an 8-bit value (0..255).

4.34 ST9

The ST9's bit addressing capabilities are quite limited: except for the BTSET instruction, only bits within the current set of working registers are accessible. A bit address is therefore of the following style:

```
rn.[!]b   ,
```

whereby ! means an optional complement of a source operand. If a bit is defined symbolically, the bit's register number is stored in bits 7..4, the bit's position is stored in bits 3..1 and the optional complement is kept in bit 0. AS distinguishes explicit and symbolic bit addresses by the missing dot. A

bit's symbolic name therefore must not contain a dot, thought it would be legal in respect to the general symbol name conventions. It is also valid to invert a symbolically referred bit:

```
bit2    bit     r5.3
        .
        .
        bld     r0.0,!bit2
```

This opportunity also allows to undo an inversion that was done at definition of the symbol.

The include file `REGST9.INC` defines the symbolic names of all on-chip registers and their associated bits. Keep however in mind that the bit definitions only work after previously setting the working register bank to the address of these peripheral registers!

In contrast to the definition file delivered with the AST9 assembler from SGS-Thomson, the names of peripheral register names are only defined as general registers (`R...`), not also as working registers (`r...`). The reason for this is that AS does not support register aliases; a tribute to assembly speed.

4.35 6804

To be honest: I only implemented this processor in AS to quarrel about SGS-Thomson's peculiar behaviour. When I first read the 6804's data book, the "incomplete" instruction set and the built-in macros immediately reminded me of the ST62 series manufactured by the same company. A more thorough comparison of the opcodes gave surprising insights: A 6804 opcode can be generated by taking the equivalent ST62 opcode and mirroring all the bits! So Thomson obviously did a bit of processor core recycling...which would be all right if they would not try to hide this: different peripherals, motorola instead of Zilog-style syntax, and the awful detail of **not** mirroring operand fields in the opcode (e.g. bit fields containing displacements). The last item is also the reason that finally convinced me to support the 6804 in AS. I personally can only guess which department at Thomson did the copy...

In contrast to its ST62 counterpart, the include file for the 6804 does not contain instruction macros that help a bit to deal with the limited machine instruction set. This is left as an exercise to the reader!

4.36 TMS3201x

It seems that every semiconductor's ambition is to invent an own notation for hexadecimal numbers. Texas Instrument took an especially eccentric approach for these processors: a > sign as prefix! The support of such a format in AS would have lead to extreme conflicts with AS's compare and shift operators. I therefore decided to use the Intel notation, which is what TI also uses for the 340x0 series and the 3201x's successors...

The instruction word of these processors unfortunately does not have enough bits to store all 8 bits for direct addressing. This is why the data address space is split into two banks of 128 words. AS principally regards the data address space as a linear segment of 256 words and automatically clears bit 7 on direct accesses (an exception is the `SST` instruction that can only write to the upper bank). The programmer has to take care that the bank flag always has the correct value!

Another hint that is well hidden in the data book: The `SUBC` instruction internally needs more than one clock for completion, but the control unit already continues to execute the next instruction. An instruction following `SUBC` therefore may not access the accumulator. AS does not check for such conditions!

4.37 TMS320C2x

As I did not write this code generator myself (that does not lower its quality by any standard), I can only roughly line out why there are some instructions that force a prefixed label to be untyped, i.e. not assigned to any specific address space: The 2x series of TMS signal processors has a code and a data segment which are both 64 Kbytes large. Depending on external circuitry, code and data space may overlap, e.g. to allow storage of constants in the code area and access them as data. Data storage in the code segment may be necessary because older versions of AS assume that the data segment only consists of RAM that cannot have a defined power-on state in a single board system. They therefore reject storage of contents in other segments than `CODE`. Without the feature of making symbols untyped, AS would punish every access to a constant in code space with a warning ("symbol out of

wrong segment"). To say it in detail, the following instructions make labels untyped:

```
BSS, STRING, RSTRING, BYTE, WORD , LONG
FLOAT, DOUBLE, EFLOAT, BFLOAT and TFLOAT
```

If one needs a typed label in front of one of these instructions, one can work around this by placing the label in a separate line just before the pseudo instruction itself. On the other hand, it is possible to place an untyped label in front of another pseudo instruction by defining the label with `EQU`, e.g.

```
<name>  EQU     $       .
```

4.38 TMS320C3x

The syntax detail that created the biggest amount of headache for me while implementing this processor family is the splitting of parallel instructions into two separate source code lines. Fortunately, both instructions of such a construct are also valid single instructions. AS therefore first generates the code for the first instruction and replaces it by the parallel machine code when a parallel construct is encountered in the second line. This operation can be noticed in the assembly listing by the machine code address that does not advance and the double dot replaced with a `R`.

Compared to the TI assembler, AS is not as flexible regarding the position of the double lines that signify a parallel operation (`||`): One either has to place them like a label (starting in the first column) or to prepend them to the second mnemonic. The line parser of AS will run into trouble if you do something else...

4.39 TMS9900

Similar to most older TI microprocessor families, TI used an own format for hexadecimal and binary constants. AS instead favours the Intel syntax which is also common for newer processor designs from TI.

The TI syntax for registers allows to use a simple integer number between 0 and 15 instead of a real name (`Rx` or `WRx`). This has two consequences:

- R0...R15 resp. WR0..WR15 are simple predefined integer symbols with values from 0 to 15, and the definition of register aliases is a simple matter of EQU.

- In contrast to several other processors, I cannot offer the additional AS feature that allows to omit the character sigifying absolute addressing (a sign in this case). As a missing character would mean register numbers (from 0 to 15) in this case, it was not possible to offer the optional omission.

Furthermore, TI sometimes uses Rx to name registers and WRx at other places...currently both variants are recognized by AS.

4.40 TMS70Cxx

This processor family belongs to the older families developed by TI and therefore TI's assemblers use their proprietary syntax for hexadecimal resp. binary constants (a prefixed < resp. ? character). As this format could not be realized for AS, the Intel syntax is used by default. This is the format TI to which also switched over when introducing the successors, of this family, the 370 series of microcontrollers. Upon a closer inspection of both's machine instruction set, one discovers that about 80% of all instruction are binary upward compatible, and that also the assembly syntax is almost identical - but unfortunately only almost. TI also took the chance to make the syntax more orthogonal and simple. I tried to introduce the majority of these changes also into the 7000's instruction set:

- It is valid to use the more common # sign for immediate addressing instead of the percent sign.

- If a port address (P...) is used as source or destination in a AND, BTJO, BTJZ, MOV, OR, or XOR instruction, it is not necessary to use the mnemonic variant with an appended P - the general form is sufficient.

- The prefixed @ sign for absolute or B-relative addressing may be omitted.

- Instead of `CMPA, CMP` with `A` as target may be written.
- Instead of `LDA` resp. `STA`, one can simply use the `MOV` instruction with `A` as source resp. destination.
- One can write `MOVW` instead of `MOVD`.
- It is valid to abbreviate `RETS` resp. `RETI` as `RTS` resp. `RTI`.
- `TSTA` resp. `TSTB` may be written as `TST A` resp. `TST B`.
- `XCHB B` is an alias for `TSTB`.

An important note: these variants are only allowed for the TMS70Cxx - the corresponding 7000 variants are not allowed for the 370 series!

4.41 TMS370xxx

Though these processors do not have specialized instructions for bit manipulation, the assembler creates (with the help of the `DBIT` instruction - see there) the illusion as if single bits were addressable. To achieve this, the `DBIT` instructions stores an address along with a bit position into an integer symbol which may then be used as an argument to the pseudo instructions `SBIT0, SBIT1, CMPBIT, JBIT0`, and `JBIT1`. These are translated into the instructions `OR, AND, XOR, BTJZ`, and `BTJO` with an appropriate bit mask.

There is nothing magic about these bit symbols, they are simple integer values that contain the address in their lower and the bit position in their upper half. One could construct bit symbols without the `DBIT` instruction, like this:

```
defbit  macro   name,bit,addr
name    equ     addr+(bit<<16)
        endm
```

but this technique would not lead to the `EQU`-style syntax defined by TI (the symbol to be defined replaces the label field in a line). **CAUTION!** Though `DBIT` allows an arbitrary address, the pseudo instructions can only operate with addresses either in the range from 0..255 or 1000h..10ffh. The processor does not have an absolute addressing mode for other memory ranges...

4.42 MSP430

The MSP was designed to be a RISC processor with a minimal power consumption. The set of machine instructions was therefore reduced to the absolute minimum (RISC processors do not have a microcode ROM so every additional instruction has to be implemented with additional silicon that increases power consumption). A number of instructions that are hardwired for other processors are therefore emulated with other instructions. For AS, these instructions are defined in the include file `REGMSP.INC`. You will get error messages for more than half of the instructions defined by TI if you forget to include this file!

4.43 COP8 & SC/MP

National unfortunately also decided to use the syntax well known from IBM mainframes (and much hated by me..) to write non-decimal integer constants. Just like with other processors, this does not work with AS's parser. ASMCOP however fortunately also seems to allow the C syntax, which is why this became the default for the COP series and the SC/MP...

4.44 SC144xxx

Originally, National offered a relatively simple assembler for this series of DECT controllers. An much more powerful assembler has been announced by IAR, but it is not available up to now. However, since the development tools made by IAR are as much target-independent as possible, one can roughly estimate the pseudo instructions it will support by looking at other available target platforms. With this in mind, the (few) SC144xx-specific instructions `DC, DC8, DW16, DS, DS8, DS16, DW` were designed. Of course, I didn't want to reinvent the wheel for pseudo instructions whose functionality is already part of the AS core. Therefore, here is a little table with equivalences. The statements `ALIGN, END, ENDM, EXITM, MACRO, ORG, RADIX, SET,` and `REPT` both exist for the IAR assembler and AS and have same functionality. Changes are needed for the following instructions:

IAR	AS	Funktion
`#include`	`include`	include file
`#define`	`SET, EQU`	define symbol
`#elif, ELIF, ELSEIF`	`ELSEIF`	start another IF branch
`#else, ELSE`	`ELSE`	last branch of an IF construct
`#endif, ENDIF`	`ENDIF`	ends an IF construct
`#error`	`ERROR, FATAL`	create error message
`#if, IF`	`IF`	start an IF construct
`#ifdef`	`IFDEF`	symbol defined ?
`#ifndef`	`IFNDEF`	symbol not defined ?
`#message`	`MESSAGE`	output message
`=, DEFINE, EQU`	`=, EQU`	fixed value assignment
`EVEN`	`ALIGN 2`	force PC to be equal
`COL, PAGSIZ`	`PAGE`	set page size for listing
`ENDR`	`ENDM`	end REPT construct
`LSTCND, LSTOUT`	`LISTING`	control amount of listing
`LSTEXP, LSTREP`	`MACEXP`	list expanded macros?
`LSTXRF`	`<command line>`	generate cross reference
`PAGE`	`NEWPAGE`	new page in listing
`REPTC`	`IRPC`	repetition with character replacement

There is no direct equivalent for `CASEON`, `CASEOFF, LOCAL`, `LSTPAG`, `#undef,` and `REPTI`.

A 100% equivalent is of course impossible as long as there is no C-like preprocessor in AS. C-like comments unfortunately are also impossible at the moment. Caution: When modifying IAR codes for AS, do not forget to move converted preprocessor statements out of column 1 as AS reserves this column exclusively for labels!

4.45 75K0

Similar to other processors, the assembly language of the 75 series also knows pseudo bit operands, i.e. it is possible to assign a combination of address and

bit number to a symbol that can then be used as an argument for bit oriented instructions just like explicit expressions. The following three instructions for example generate the same code:

```
ADM     sfr     0fd8h
SOC     bit     ADM.3

        skt     0fd8h.3
        skt     ADM.3
        skt     SOC
```

AS distinguishes direct and symbolic bit accesses by the missing dot in symbolic names; it is therefore forbidden to use dots in symbol names to avoid misunderstandings in the parser.

The storage format of bit symbols mostly accepts the binary coding in the machine instructions themselves: 16 bits are used, and there is a "long" and a "short" format. The short format can store the following variants:

- direct accesses to the address range from 0FBxH to 0FFxH
- indirect accesses in the style of `Addr.@L` (0FC0H $\leq$ `Addr` $\leq$0FFFH)
- indirect accesses in the style of `@H+d4.bit`

The upper byte is set to 0, the lower byte contains the bit expression coded according to [73]. The long format in contrast only knows direct addressing, but it can cover the whole address space (given a correct setting of MBS and MBE). A long expression stores bits 0..7 of the address in the lower byte, the bit position in bits 8 and 9, and a constant value of 01 in bits 10 and 11. The highest bits allow to distinguish easily between long and short addresses via a check if the upper byte is 0. Bits 12..15 contain bits 8..11 of the address; they are not needed to generate the code, but they have to be stored somewhere as the check for correct banking can only take place when the symbol is actually used.

4.46 78K0

NEC uses different ways to mark absolute addressing in its data books:

- absolute short: no prefix
- absolute long: prefix of `!`
- PC relative: prefix of `$`

Under AS, these prefixes are only necessary if one wants to force a certain addressing mode and the instruction allows different variants. Without a prefix, AS will automatically select the shortest variant. It should therefore rarely be necessary to use a prefix in practice.

4.47 78K2

Analogous to the 78K0, NEC here also uses dollar signs and exclamation marks to specify different lengths of address expressions. The selection between long and short addresses is done automatically (both in RAM and SFR areas), only relative addressing has to be selected explicitly, if an instruction supports both variants (like `BR`).

An additional remark (which is also true for the 78K0): Those who want to use Motorola syntax via `RELAXED`, might have to put hexadecimal constants in parentheses, since the leading dollar sign might be misunderstood as relative addressing...

4.48 uPD772x

Both the 7720 and 7725 are provided by the same code generator and are extremely similar in their instruction set. One should however not beleive that they are binary compatible: To get space for the longer address fields and additional instructions, the bit positions of some fields in the instruction word have changed, and the instruction length has changed from 23 to 24 bits. The code format therefore uses different header ids for both CPUs.

They both have in common that in addition to the code and data segment, there is also a ROM for storage of constants. In the case of AS, it is mapped onto the `ROMDATA` segment!

4.49 F2MC16L

Along with the discussion of the `ASSUME` statement, it has already been mentioned that it is important to inform AS about the correct current values of all bank registers - if your program uses more than 64K RAM or 64K ROM. With these assumptions in mind, AS checks every direct memory access for attempts to access a memory location that is currently not in reach. Of course, standard situations only require knowledge of DTB and DPR for this purpose, since ADB resp. SSB/USB are only used for indirect accesses via RW2/RW6 resp. RW3/RW7 and this mechanism anyway doesn't work for indirect accesses. However, similar to the 8086, it is possible to place a prefix in front of an instruction to replace DTB by a different register. AS therefore keeps track of used segment prefixes and toggles appropriately for the next *machine instruction*. A pseudo instruction placed between the prefix and the machine instruction does *not* reset the toggle. This is also true for pseudo instructions that store data or modify the program counter. Which doesn't make much sense anyway...

Chapter 5

File Formats

In this chapter, the formats of files AS generates shall be explained whose formats are not self-explanatory.

5.1 Code Files

The format for code files generated by the assembler must be able to separate code parts that were generated for different target processors; therefore, it is a bit different from most other formats. Though the assembler package contains tools to deal with code files, I think is a question of good style to describe the format in short:

If a code file contains multibyte values, they are stored in little endian order. This rule is already valid for the 16-bit magic word $1489, i.e. every code file starts with the byte sequence $89/$14.

This magic word is followed by an arbitrary number of "records". A record may either contain a continuous piece of the code or certain additional information. Even without switching to different processor types, a file may contain several code-containing records, in case that code or constant data areas are interrupted by reserved memory areas that should not be initialized. This way, the assembler tries to keep the file as short as possible.

Common to all records is a header byte which defines the record's type and its contents. Written in a PASCALish way, the record structure can be described in the following way:

```
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
```

This description does not express fully that the length of data fields is variable and depends on the value of the `Length` entries.

A record with a header byte of $81 is a record that may contain code or data from arbitrary segments. The first byte (`Header`) describes the processor family the following code resp. data was generated for (see tables 5.1 and 5.2).

The `Segment` field signifies the address space the following code belongs to. The assignment defined in table 5.3 applies. The `Gran` field describes the code's "granularity", i.e. the size of the smallest addressable unit in the following set of data. This value is a function of processor type and segment and is an important parameter for the interpretation of the following two fields that describe the block's start address and its length: While the start address refers to the granularity, the `Length` value is always expressed in bytes! For example, if the start address is $300 and the length is 12, the resulting end address would be $30b for a granularity of 1, however $303 for a granularity of 4! Granularities that differ from 1 are rare and mostly appear in DSP CPU's that are not designed for byte processing. For example, a DSP56K's address space is organized in 64 Kwords of 16 bits. The resulting storage capacity is 128 Kbytes, however it is organized as 2^{16} words that are addressed with addresses 0,1,2,...65535!

The start address is always 32 bits in size, independent of the processor family. In contrast, the length specification has only 16 bits, i.e. a record may have a maximum length of 4+4+2+(64K-1) = 65545 bytes.

Header	Family	Header	Family
$01	680x0, 6833x	$03	M*Core
$04	XGATE	$05	PowerPC
$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
$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	$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
$73	TMS-7000	$74	TMS3201x
$75	TMS320C2x	$76	TMS320C3x

Table 5.1: Header Bytes for the Different Processor Families

Header	Family	Header	Family
$77	TMS320C20x/TMS320C5x	$78	ST6
$79	Z8	$7a	μPD78(C)10
$7b	75K0	$7c	78K0
$7d	μPD7720	$7e	μPD7725
$7f	μPD77230		

Table 5.2: Header Bytes for the Different Processor Families

number	segment	number	segment
$00	<undefined>	$01	`CODE`
$02	`DATA`	$03	`IDATA`
$04	`XDATA`	$05	`YDATA`
$06	`BDATA`	$07	`IO`
$08	`REG`	$09	`ROMDATA`

Table 5.3: Codings of the `Segment` Field

Data records with a Header ranging from $01 to $7f present a shortcut and preserve backward compatibility to earlier definitions of the file format: in their case, the Header directly defines the processor type, the target segment is fixed to `CODE` and the granularity is implicitly given by the processor type, rounded up to the next power of two. AS prefers to use these records whenever data or code should go into the `CODE` segment.

A record with a Header of $80 defines an entry point, i.e. the address where execution of the program should start. Such a record is the result of an `END` statement with a corresponding address as argument.

The last record in a file bears the Header $00 and has only a string as data field. This string does not have an explicit length specification; its end is equal to the file's end. The string contains only the name of the program that created the file and has no further meaning.

5.2 Debug Files

Debug files may optionally be generated by AS. They deliver important information for tools used after assembly, like disassemblers or debuggers. AS

can generate debug files in one of three formats: On the one hand, the object format used by the AVR tools from Atmel respectively a NoICE-compatible command file, and on the other hand an own format. The first two are described in detail in [4] resp. the NoICE documentations, which is why the following description limits itself to the AS-specific MAP format:

The information in a MAP file is split into three groups:

- symbol table
- memory usage per section
- machine addresses of source lines

The second item is listed first in the file. A single entry in this list consists of two numbers that are separated by a `:` character:

```
<line number>:<address>
```

Such an entry states that the machine code generated for the source statement in a certain line is stored at the mentioned address (written in hexadecimal notation). With such an information, a debugger can display the corresponding source lines while stepping through a program. As a program may consist of several include files, and due to the fact that a lot of processors have more than one address space (though admittedly only one of them is used to store executable code), the entries described above have to be sorted. AS does this sorting in two levels: The primary sorting criteria is the target segment, and the entries in one of these sections are sorted according to files. The sections resp. subsections are separated by special lines in the style of

```
Segment <segment name>
```

resp.

```
File <file name>   .
```

The source line info is followed by the symbol table. Similar to the source line info, the symbol table is primarily sorted by the segments individual symbols are assigned to. In contrast to the source line info, an additional section `NOTHING` exists which contains the symbols that are not assigned to any specific segment (e.g. symbols that have been defined with a simple `EQU` statement). A section in the symbol table is started with a line of the following type:

```
Symbols in Segment <segment name>
```

The symbols in a section are sorted according to the alphabetical order of their names, and one symbol entry consists of exactly one line. Such a line consists of 5 fields witch are separated by at least a single space:

The first field is the symbol's name, possibly extended by a section number enclosed in brackets. Such a section number limits the range of validity for a symbol. The second field designates the symbol's type: `Int` stands for integer values, `Float` for floating point numbers, and `String` for character arrays. The third field finally contains the symbol's value. If the symbol contains a string, it is necessary to use a special encoding for control characters and spaces. Without such a coding, spaces in a string could be misinterpreted as delimiters to the next field. AS uses the same syntax that is also valid for assembly source files: Instead of the character, its ASCII value with a leading backslash (\) is inserted. For example, the string

```
This is a test
```

becomes

```
This\032is\032\a\032test    .
```

The numerical value always has three digits and has to be interpreted as a decimal value. Naturally, the backslash itself also has to be coded this way.

The fourth field specifies - if available - the size of the data structure placed at the address given by the symbol. A debugger may use this information to automatically display variables in their correct length when they are referred symbolically. In case AS does not have any information about the symbol size, this field simply contains the value -1.

Finally,the fifth field states via the values 0 or 1 if the symbol has been used during assembly. A program that reads the symbol table can use this field to skip unused symbols as they are probably unused during the following debugging/disassembly session.

The third section in a debug file describes the program's sections in detail. The need for such a detailed description arises from the sections' ability to limit the validity range of symbols. A symbolic debugger for example cannot use certain symbols for a reverse translation, depending on the current PC value. It may also have to regard priorities for symbol usage when a value is represented by more than one symbol. The definition of a section starts with a line of the following form:

`Info for Section nn ssss pp`

`nn` specifies the section's number (the number that is also used in the symbol table as a postfix for symbol names), `ssss` gives its name and `pp` the number of its parent section. The last information is needed by a retranslator to step upward through a tree of sections until a fitting symbol is found. This first line is followed by a number of further lines that describe the code areas used by this section. Every single entry (exactly one entry per line) either describes a single address or an address range given by a lower and an upper bound (separation of lower and upper bound by a minus sign). These bounds are "inclusive", i.e. the bounds themselves also belong to the area. Is is important to note that an area belonging to a section is not additionally listed for the section's parent sections (an exception is of course a deliberate multiple allocation of address areas, but you would not do this, would you?). On the one hand, this allows an optimized storage of memory areas during assembly. On the other hand, this should not be an obstacle for symbol backtranslation as the single entry already gives an unambiguous entry point for the symbol search path. The description of a section is ended by an empty line or the end of the debug file.

Program parts that lie out of any section are not listed separately. This implicit "root section" carries the number -1 and is also used as parent section for sections that do not have a real parent section.

It is possible that the file contains empty lines or comments (semi colon at line start). A program reading the file has to ignore such lines.

Chapter 6

Utility Programs

To simplify the work with the assembler's code format a bit, I added some tools to aid processing of code files. These programs are released under the same license terms as stated in section 1.1!

Common to all programs are the possible return codes they may deliver upon completion (see table 6.1).

return code	error condition
0	no errors
1	error in command line parameters
2	I/O error
3	file format error

Table 6.1: Return Codes of the Utility Programs

Just like AS, all programs take their input from STDIN and write messages to STDOUT (resp. error messages to STDERR). Therefore, input and output redirections should not be a problem.

In case that numeric or address specifications have to be given in the command line, they may also be written in hexadecimal notation when they are prefixed with a dollar character or a `0x` like in C. (e.g. `$10` or `0x10` instead of 16).

Unix shells however assign a special meaning to the dollar sign, which makes *UNIX*

it necessary to escape a dollar sign with a backslash. The `0x` variant is definitely more comfortable in this case.

Otherwise, calling conventions and variations are equivalent to those of AS (except for PLIST and AS2MSG); i.e. it is possible to store frequently used parameters in an environment variable (whose name is constructed by appending CMD to the program's name, i.e. `BINDCMD` for BIND), to negate options, and to use all upper- resp. lower-case writing (for details on this, see section 2.4).

Address specifications always relate to the granularity of the processor currently in question; for example, on a PIC, an address difference of 1 means a word and not a byte.

6.1 PLIST

PLIST is the simplest one of the five programs supplied: its purpose is simply to list all records that are stored in a code file. As the program does not do very much, calling is quite simple:

```
PLIST <file name>
```

The file name will automatically be extended with the extension `P` if it doesn't already have one.

CAUTION! At this place, no wildcards are allowed! If there is a necessity to list several files with one command, use the following "mini batch":

```
for %n in (*.p) do plist %n
```

PLIST prints the code file's contents in a table style, whereby exactly one line will be printed per record. The individual rows have the following meanings:

- code type: the processor family the code has been generated for.
- start address: absolute memory address that expresses the load destination for the code.
- length: length of this code chunk in bytes.

- end address: last address of this code chunk. This address is calculated as start address+length-1.

All outputs are in hexadecimal notation.

Finally, PLIST will print a copyright remark (if there is one in the file), together with a summaric code length.

Simply said, PLIST is a sort of DIR for code files. One can use it to examine a file's contents before one continues to process it.

6.2 BIND

BIND is a program that allows to concatenate the records of several code files into a single file. A filter function is available that can be used to copy only records of certain types. Used in this way, BIND can also be used to split a code file into several files.

The general syntax of BIND is

```
BIND <source file(s)> <target file> [options]
```

Just like AS, BIND regards all command line arguments that do not start with a `+`, `-` or `/` as file specifications, of which the last one must designate the destination file. All other file specifications name sources, which may again contain wildcards.

Currently, BIND defines only one command line option:

- `f` <`Header[,Header]`>: sets a list of record headers that should be copied. Records with other header IDs will not be copied. Without such an option, all records will be copied. The headers given in the list correspond to the `HeaderID` field of the record structure described in section 5.1. Individual headers in this list are separated with commas.

For example, to filter all MCS-51 code out of a code file, use BIND in the following way:

```
BIND <source name> <target name> -f $31
```

If a file name misses an extension, the extension `P` will be added automatically.

6.3 P2HEX

P2HEX is an extension of BIND. It has all command line options of BIND and uses the same conventions for file names. In contrary to BIND, the target file is written as a Hex file, i.e. as a sequence of lines which represent the code as ASCII hex numbers.

P2HEX knows 8 different target formats, which can be selected via the command line parameter `F`:

- Motorola S-Records `(-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, see [4])

If no target format is explicitly specified, P2HEX will automatically choose one depending in the processor type: S-Records for Motorola CPUs, Hitachi, and TLCS-900, MOS for 65xx/MELPS, DSK for the 16 bit signal processors from Texas, Atmel Generic for the AVRs, and Intel Hex for the rest. Depending on the start addresses width, the S-Record format will use Records of type 1, 2, or 3, however, records in one group will always be of the same type. This automatism can be partially suppressed via the command line option

```
-M <1|2|3>
```

A value of 2 resp. 3 assures that that S records with a minimum type of 2 resp. 3 will be used, while a value of 1 corresponds to the full automatism.

Normally, the AVR format always uses an address length of 3 bytes. Some programs however do not like that...which is why there is a switch

```
-avrlen <2|3>
```

that allows to reduce the address length to two bytes in case of emergency.

The Intel, MOS and Tektronix formats are limited to 16 bit addresses, the 16-bit Intel format reaches 4 bits further. Addresses that are to long for a given format will be reported by P2HEX with a warning; afterwards, they will be truncated (!).

For the PIC microcontrollers, the switch

```
-m <0..3>
```

allows to generate the three different variants of the Intel Hex format. Format 0 is INHX8M which contains all bytes in a Lo-Hi-Order. Addresses become double as large because the PICs have a word-oriented address space that increments addresses only by one per word. This format is also the default. With Format 1 (INHX16M), bytes are stored in their natural order. This is the format Microchip uses for its own programming devices. Format 2 (INHX8L) resp. 3 (INHX8H) split words into their lower resp. upper bytes. With these formats, P2HEX has to be called twice to get the complete information, like in the following example:

```
p2hex test -m 2
rename test.hex test.obl
p2hex test -m 3
rename test.hex test.obh
```

For the Motorola format, P2HEX additionally uses the S5 record type mentioned in [8]. This record contains the number of data records (S1/S2/S3) to follow. As some programs might not know how to deal with this record, one can suppress it with the option

```
+5  .
```

In case a source file contains code record for different processors, the different hex formats will also show up in the target file - it is therefore strongly advisable to use the filter function.

Apart form this filter function, P2HEX also supports an address filter, which is useful to split the code into several parts (e.g. for a set of EPROMs):

```
-r <start address>-<end address>
```

The start address is the first address in the window, and the end address is the last address in the window, **not** the first address that is out of the window. For example, to split an 8051 program into 4 2764 EPROMs, use the following commands:

```
p2hex <source file> eprom1 -f $31 -r $0000-$1fff
p2hex <source file> eprom2 -f $31 -r $2000-$3fff
p2hex <source file> eprom3 -f $31 -r $4000-$5fff
p2hex <source file> eprom4 -f $31 -r $6000-$7fff
```

By default, the address window is 32 Kbytes large and starts at address 0.

CAUTION! This type of splitting does not change the absolute addresses that will be written into the files! If the addresses in the individual hex files should rather start at 0, one can force this with the additional switch

```
-a      .
```

On the other hand, to move the addresses to a different location, one may use the switch

```
-R <value> .
```

The value given is an *offset*, i.e. it is added to the addresses given in the code file.

A special value for start and stop address arguments is a single dollar sign (`$`). This stands for the very first resp. last address that has been used in the code file. So, if you want to be sure that always the whole program is stored in the hex file, set the address filter

```
-r $-$
```

and you do not have to worry about address filters any more. Dollar signs and fixed addresses may of course me mixed. For example, the setting

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

limits the upper end to 32 Kbytes.

By using an offset, it is possible to move a file's contents to an arbitrary position. This offset is simply appended to a file's name, surrounded with parentheses. For example, if the code in a file starts at address 0 and you want to move it to address 1000 hex in the hex file, append `($1000)` to the file's name (without spaces!).

As the TI DSK format has the ability to distinguish between data and code, there is a switch

```
-d <start>-<end>
```

to designate the address range that should be written as data instead of code. For this option, single dollar signs are **not** allowed! This option should not be used in new projects any more, since P2HEX now can directly handle data in the DATA segment.

While this switch is only relevant for the DSK format, the option

```
-e <address>
```

is also valid for the Intel and Motorola formats. Its purpose is to set the entry address that will be inserted into the hex file. If such a command line parameter is missing, P2HEX will search a corresponding entry in the code file. If even this fails, no entry address will be written to the hex file (DSK/Intel) or the field reserved for the entry address will be set to 0 (Motorola).

Unfortunately, one finds different statements about the last line of an Intel-Hex file in literature. Therefore, P2HEX knows three different variants that may be selected via the command-line parameter `i` and an additional number:

```
0  :00000001FF
1  :00000001
2  :0000000000
```

By default, variant 0 is used which seems to be the most common one.

If the target file name does not have an extension, an extension of `HEX` is supposed.

By default, P2HEX will print a maximum of 16 data bytes per line, just as most other tools that output Hex files. If you want to change this, you may use the switch

`-l <count>` .

The allowed range of values goes from 2 to 254 data bytes; odd values will implicitly be rounded down to an even count.

In most cases, the temporary code files generated by AS are not of any further need after P2HEX has been run. The command line option

`-k`

allows to instruct P2HEX to erase them automatically after conversion.

In contrast to BIND, P2HEX will not produce an empty target file if only one file name (i.e. the target name) has been given. Instead, P2HEX will use the corresponding code file. Therefore, a minimal call in the style of

```
P2HEX <name>
```

is possible, to generate <name>.hex out of <name>.p.

6.4 P2BIN

P2BIN works similar to P2HEX and offers the same options (except for the a and i options that do not make sense for binary files), however, the result is stored as a simple binary file instead of a hex file. Such a file is for example suitable for programming an EPROM.

P2BIN knows three additional options to influence the resulting binary file:

- `l <8 bit number>`: sets the value that should be used to fill unused memory areas. By default, the value $ff is used. This value assures that every half-way intelligent EPROM burner will skip these areas. This option allows to set different values, for example if you want to generate an image for the EPROM versions of MCS-48 microcontrollers (empty cells of their EPROM array contain zeroes, so $00 would be the correct value in this case).

- `s`: commands the program to calculate a checksum of the binary file. This sum is printed as a 32-bit value, and the two's complement of the least significant bit will be stored in the file's last byte. This way, the modulus- 256-sum of the file will become zero.

- `m`: is designed for the case that a CPU with a 16- or 32-bit data bus is used and the file has to be split for several EPROMs. The argument may have the following values:

  - `ALL`: copy everything
  - `ODD`: copy all bytes with an odd address
  - `EVEN`: copy all bytes with an even address
  - `BYTE0..BYTE3`: copy only bytes with an address of 4n+0 .. 4n+3
  - `WORD0, WORD1`: copy only the lower resp. upper 16- bit word of a 32-bit word

To avoid confusions: If you use this option, the resulting binary file will become smaller because only a part of the source will be copied. Therefore, the resulting file will be smaller by a factor of 2 or 4 compared to `ALL`. This is just natural...

In case the code file does not contain an entry address, one may set it via the `-e` command line option just like with P2HEX. Upon request, P2BIN prepends the resulting image with this address. The command line option

```
-S
```

activates this function. It expects a numeric specification ranging from 1 to 4 as parameter which specifies the length of the address field in bytes. This number may optionally be prepended wit a `L` or `B` letter to set the endian order of the address. For example, the specification `B4` generates a 4 byte address in big endian order, while a specification of `L2` or simply `2` creates a 2 byte address in little endian order.

6.5 AS2MSG

AS2MSG is not a tool in the real sense, it is a filter that was designed to simplify the work with the assembler for (fortunate) users of Borland Pascal 7.0. The DOS IDEs feature a 'tools' menu that can be extended with own programs like AS. The filter allows to directly display the error messages paired with a line specification delivered by AS in the editor window. A new entry has to be added to the tools menu to achieve this (Options/Tools/New). Enter the following values:

```
- Title: ~m~acro assembler
- Program path: AS
- Command line:
     -E !1 $EDNAME $CAP MSG(AS2MSG) $NOSWAP $SAVE ALL
- assign a hotkey if wanted (e.g. Shift-F7)
```

The -E option assures that Turbo Pascal will not become puzzled by STDIN and STDERR.

I assume that AS and AS2MSG are located in a directory listed in the `PATH` variable. After pressing the appropriate hotkey (or selecting AS from the tools menu), as will be called with the name of the file loaded in the active editor window as parameter. The error messages generated during assembly are redirected to a special window that allows to browse through the errors. `Ctrl-Enter` jumps to an erroneous line. The window additionally contains the statistics AS prints at the end of an assembly. These lines obtain the dummy line number 1.

`TURBO.EXE` (Real Mode) and `BP.EXE` (Protected Mode) may be used for this way of working with AS. I recommend however BP, as this version does not have to 'swap' half of the DOS memory before before AS is called.

Appendix A

Error Messages of AS

Here is a list of all error messages emitted by AS. Each error message is described by:

- the internal error number (it is displayed only if AS is started with the `-n` option)
- the text of the error message
- error type:
  - Warning: informs the user that a possible error was found, or that some inefficient binary code could be generated. The assembly process is not stopped.
  - Error: an error was detected. The assembly process continues, but no binary code is emitted.
  - Fatal: unrecoverable error. The assembly process is terminated.
- reason of the error: the situation originating the error.
- argument: a further explanation of the error message.

0 useless displacement

Type:
warning

Reason:
680x0, 6809 and COP8 CPUs: an address displacement of 0 was given. An address expression without displacement is generated, and a convenient number of NOPs are emitted to avoid phasing errors.

Argument:
none

10 short addressing possible

Type:
warning

Reason:
680x0-, 6502 and 68xx CPUs: a given memory location can be reached using short addressing. A short addressing instruction is emitted, together with the required number of NOPs to avoid phasing errors.

Argument:
none

20 short jump possible

Type:
warning

Reason:
680x0- and 8086 CPUs can execute jumps using a short or long displacement. If a shorter jump was not explicitly requested, in the first pass room for the long jump is reserved. Then the code for the shorter jump is emitted, and the remaining space is filled with NOPs to avoid phasing errors.

Argument:
none

30 no sharefile created, SHARED ignored

Type:
warning

Reason:

A `SHARED` directive was found, but on the command line no options were specified, to generate a shared file.

Argument:

none

40 FPU possibly cannot read this value (>=1E1000)

Type:

warning

Reason:

The BCD-floating point format used by the 680x0-FPU allows such a large exponent, but according to the latest databooks, this cannot be fully interpreted. The corresponding word is assembled, but the associated function is not expected to produce the correct result.

Argument:

none

50 privileged instruction

Type:

warning

Reason:

A Supervisor-mode directive was used, that was not preceded by an explicit `SUPMODE ON` directive

Argument:

none

60 distance of 0 not allowed for short jump (NOP created instead)

Type:

warning

Reason:

A short jump with a jump distance equal to 0 is not allowed by 680x0 resp. COP8 processors, since the associated code word is used to identify long jump instruction. Instead of a jump instruction, AS emits a NOP

Argument:
none

70 symbol out of wrong segment

Type:
warning

Reason:
The symbol used as an operand comes from an address space that cannot be addressed together with the given instruction

Argument:
none

75 segment not accessible

Type:
warning

Reason:
The symbol used as an operand belongs to an address space that cannot be accessed with any of the segment registers of the 8086

Argument:
The name of the inaccessible segment

80 change of symbol values forces additional pass

Type:
warning

Reason:
A symbol changed value, with respect to previous pass. This warning is emitted only if the `-r` option is used.

Argument:
name of the symbol that changed value.

90 overlapping memory usage

Type:
warning

Reason:
The analysis of the usage list shows that part of the program memory was used more than once. The reason can be an excessive usage of `ORG` directives.

Argument:
none

100 none of the CASE conditions was true

Type:
warning

Reason:
A `SWITCH...CASE` directive without `ELSECASE` clause was executed, and none of the `CASE` conditions was found to be true.

Argument:
none

110 page might not be addressable

Type:
warning

Reason:
The symbol used as an operand was not found in the memory page defined by an `ASSUME` directive (ST6, 78(C)10).

Argument:
none

120 register number must be even

Type:
warning

Reason:
The CPU allows to concatenate only register pairs, whose start address is even (RR0, RR2, ..., only for Z8).

Argument:
none

130 obsolete instruction, usage discouraged

Type:
warning

Reason:
The instruction used, although supported, was superseded by a new instruction. Future versions of the CPU could no more implement the old instruction.

Argument:
none

140 unpredictable execution of this instruction

Type:
warning

Reason:
The addressing mode used for this instruction is allowed, however a register is used in such a way that its contents cannot be predicted after the execution of the instruction.

Argument:
none

150 localization operator senseless out of a section

Type:
warning

Reason:
An aheaded must be used, so that it is explicitly referred to the local symbols used in the section. When the operator is used out of a section, there are no local symbols, because this operator is useless in this context.

Argument:
none

160 senseless instruction

Type:
warning

Reason:

The instruction used has no meaning, or it can be substituted by an other instruction, shorter and more rapidly executed.

Argument:

none

170 unknown symbol value forces additional pass

Type:

warning

Reason:

AS expects a forward definition of a symbol, i.e. a symbol was used before it was defined. A further pass must be executed. This warning is emitted only if the `-r` option was used.

Argument:

none

180 address is not properly aligned

Type:

warning

Reason:

An address was used that is not an exact multiple of the operand size. Although the CPU databook forbids this, the address could be stored in the instruction word, so AS simply emits a warning.

Argument:

none.

190 I/O-address must not be used here

Type:

warning

Reason:

The addressing mode or the address used are correct, but the address refers to the peripheral registers, and it cannot be used in this circumstance.

Argument:

none.

200 possible pipelining effects

Type:
warning

Reason:
A register is used in a series of instructions, so that a sequence of instructions probably does not generate the desired result. This usually happens when a register is used before its new content was effectively loaded in it.

Argument:
the register probably causing the problem.

210 multiple use of address register in one instruction

Type:
warning

Reason:
A register used for the addressing is used once more in the same instruction, in a way that results in a modification of the register value. The resulting address does not have a well defined value.

Argument:
the register used more than once.

220 memory location is not bit addressable

Type:
warning

Reason:
Via a `SFRB` statement, it was tried to declare a memory cell as bit addressable which is not bit addressable due to the 8051's architectural limits.

Argument:
none

230 stack is not empty

Type:
warning

Reason:
At the end of a pass, a stack defined by the program is not empty.

Argument:
the name of the stack and its remaining depth

240 NUL character in string, result is undefined

Type:
warning

Reason:
A string constant contains a NUL character. Though this works with the Pascal version, it is a problem for the C version of AS since C itself terminates strings with a NUL character. i.e. the string would have its end for C just at this point...

Argument:
none

250 instruction crosses page boundary

Type:
warning

Reason:
The parts of a machine statement partiallly lie on different pages. As the CPU's instruction counter does not get incremented across page boundaries, the processor would fetch at runtime the first byte of the old page instead of the instruction's following byte; the program would execute incorrectly.

Argument:
none

260 range overflow

Type:
warning

Reason:
A numeric value was out of the allowed range. AS brought the value back into the allowed range by truncating upper bits, but it is not guaranteed that meaningful and correct code is generated by this.

Argument:
none

270 negative argument for DUP

Type:
warning

Reason:
The repetition argument of a DUP directive was smaller than 0. Analogous to a count of exactly 0, no data is stored.

Argument:
none

280 single X operand interpreted as indexed and not implicit addressing

Type:
warning

Reason:
A single X operand may be interpreted either as register X or x-indexed addressing with zero displacement, since Motorola does not specify this variant. AS chooses the latter, which may not be the desired one.

Argument:
none

300 bit number will be truncated

Type:
warning

Reason:
This instruction only operates on byte resp. longword operands. bit numbers beyond 7 resp. 31 will be treated modulo-8 resp. modulo-32 by the CPU.

Argument:
none

310 invalid register pointer value

Type:
warning

Reason:
Valid values for the RP register range from 0x00 to 0x70 resp. 0xf0, because all other areas are unused on the Z8.

Argument:
none

1000 symbol double defined

Type:
error

Reason:
A new value is assigned to a symbol, using a label or a `EQU, PORT, SFR, LABEL, SFRB` or `BIT` instruction: however this can be done only using `SET/EVAL`.

Argument:
the name of the offending symbol, and the line number where it was defined for the first time, according to the symbol table.

1010 symbol undefined

Type:
error

Reason:
A symbol is still not defined in the symbol table, also after a second pass.

Argument:
the name of the undefined symbol.

1020 invalid symbol name

Type:
error

Reason:
A symbol does not fulfill the requirements that symbols must have to be considered valid by AS. Please pay attention that more stringent syntax rules exist for macros and function parameters.

Argument:
the wrong symbol

1090 invalid format

Type:
error

Reason:
The instruction format used does not exist for this instruction.

Argument:
the known formats for this command

1100 useless attribute

Type:
error

Reason:
The instruction (processor or pseudo) cannot be used with a point-suffixed attribute.

Argument:
none

1105 attribute may only be one character long

Type:
error

Reason:
The attribute following a point after an instruction must not be longer or shorter than one character.

Argument:
none

1107 undefined attribute

Type:
error

Reason:
This instruction uses an invalid attribute.

Argument:
none

1110 wrong number of operands

Type:
error

Reason:
The number of arguments issued for the instruction (processor or pseudo) does not conform with the accepted number of operands.

Argument:
none

1115 wrong number of operations

Type:
error

Reason:
The number of options given with this command is not correct.

Argument:
none

1120 addressing mode must be immediate

Type:
error

Reason:
The instruction can be used only with immediate operands (preceded by `#`).

Argument:
none

1130 invalid operand size

Type:
error

Reason:
Although the operand is of the right type, it does not have the correct length (in bits).

Argument:
none

1131 conflicting operand sizes

Type:
error

Reason:
The operands used have different length (in bits)

Argument:
none

1132 undefined operand size

Type:
error

Reason:
It is not possible to estimate, from the opcode and from the operands, the size of the operand (a trouble with 8086 assembly). You must define it with a `BYTE or WORD PTR` prefix.

Argument:
none

1135 invalid operand type

Type:
error

Reason:
an expression does not have a correct operand type (integer/-decimal/string)

Argument:
the operand type

1140 too many arguments

Type:
error

Reason:
No more than 20 arguments can be given to any instruction

Argument:
none

1200 unknown opcode

Type:
error

Reason:
An was used that is neither an AS instruction, nor a known mnemonic for the current processor type.

Argument:
none

1300 number of opening/closing brackets does not match

Type:
error

Reason:
The expression parser found an expression enclosed by parentheses, where the number of opening and closing parentheses does not match.

Argument:
the wrong expression

1310 division by 0

Type:
error

Reason:
An expression on the right side of a division or modulus operation was found to be equal to 0.

Argument:
none

1315 range underflow

Type:
error

Reason:
An integer word underflowed the allowed range.

Argument:
the value of the word and the allowed minimum (in most cases, maybe I will complete this one day...)

1320 range overflow

Type:
error

Reason:
An integer word overflowed the allowed range.

Argument:
the value of the world, and the allowed maximum (in most cases, maybe I will complete this one day...)

1325 address is not properly aligned

Type:
error

Reason:
The given address does not correspond with the size needed by the data transfer, i.e. it is not an integral multiple of the operand size. Not all processor types can use unaligned data.

Argument:
none

1330 distance too big

Type:
error

Reason:
The displacement used for an address is too large.

Argument:
none

1340 short addressing not allowed

Type:
error

Reason:
The address of the operand is outside of the address space that can be accessed using short-addressing mode.

Argument:
none

1350 addressing mode not allowed here

Type:
error

Reason:
the addressing mode used, although usually possible, cannot be used here.

Argument:
none

1351 number must be even

Type:
error

Reason:
At this point, only even addresses are allowed, since the low order bit is used for other purposes or it is reserved.

Argument:
none

1355 addressing mode not allowed in parallel operation

Type:
error

Reason:
The addressing mode(s) used are allowed in sequential, but not in parallel instructions

Argument:
none

1360 undefined condition

Type:
error

Reason:
The branch condition used for a conditional jump does not exist.

Argument:
none

1365 incompatible conditions

Type:
error

Reason:
The used combination of conditions is not poosible in a single instruction.

Argument:
the condition where the incompatibility was detected.

1370 jump distance too big

Type:
error

Reason:
the jump instruction and destination are too apart to execute the jump with a single step

Argument:
none

1375 jump distance is odd

Type:
error

Reason:
Since instruction must only be located at even addresses, the jump distance between two instructions must always be even, and the LSB of the jump distance is used otherwise. This issue was not verified here. The reason is usually the presence of an odd number of data in bytes or a wrong `ORG`.

Argument:
none

1380 invalid argument for shifting

Type:
error

Reason:
only a constant or a data register can be used for defining the shift size. (only for 680x0)

Argument:
none

1390 operand must be in range 1..8

Type:
error

Reason:
constants for shift size or `ADDQ` argument can be only within the 1..8 range (only for 680x0)

Argument:
none

1400 shift amplitude too big

Type:
error

Reason:
(no more used)

Argument:
none

1410 invalid register list

Type:
error

Reason:
The register list argument of `MOVEM` or `FMOVEM` has a wrong format (only for 680x0)

Argument:
none

1420 invalid addressing mode for CMP

Type:
error

Reason:
The operand combination used with the `CMP` instruction is not allowed (only for 680x0)

Argument:
none

1430 invalid CPU type

Type:
error

Reason:
The processor type used as argument for `CPU` command is unknown to AS.

Argument:
the unknown processor type

1440 invalid control register

Type:
error

Reason:
The control register used by a `MOVEC` is not (yet) available for the processor defined by the `CPU` command.

Argument:
none

1445 invalid register

Type:
error

Reason:
The register used, although valid, cannot be used in this context.

Argument:
none

1450 RESTORE without SAVE

Type:
error

Reason:
A `RESTORE` command was found, that cannot be coupled with a corresponding `SAVE`.

Argument:
none

1460 missing RESTORE

Type:
error

Reason:
After the assembling pass, a `SAVE` command was missing.

Argument:
none.

1465 unknown macro control instruction

Type:
error

Reason:
A macro option parameter is unknown to AS.

Argument:
the dubious option.

1470 missing ENDIF/ENDCASE

Type:
error

Reason:
after the assembling, some of the `IF-` or `CASE-` constructs were found without the closing command

Argument:
none

1480 invalid IF-structure

Type:
error

Reason:
The command structure in a `IF-` or `SWITCH-` sequence is wrong.

Argument:
none

1483 section name double defined

Type:
error

Reason:
In this program module a section with the same name still exists.

Argument:
the multiple-defined name

1484 unknown section

Type:
error

Reason:
In the current scope, there are no sections with this name

Argument:
the unknown name

1485 missing ENDSECTION

Type:
error

Reason:
Not all the sections were properly closed.

Argument:
none

1486 wrong ENDSECTION

Type:
error

Reason:
The given `ENDSECTION` does not refer to the most deeply nested one.

Argument:
none

1487 ENDSECTION without SECTION

Type:
error

Reason:
An `ENDSECTION` command was found, but the associated section was not defined before.

Argument:
none

1488 unresolved forward declaration

Type:
error

Reason:
A symbol declared with a `FORWARD` or `PUBLIC` statement could not be resolved.

Argument:
the name of the unresolved symbol.

1489 conflicting FORWARD $< - >$ PUBLIC-declaration

Type:
error

Reason:
A symbol was defined both as public and private.

Argument:
the name of the symbol.

1490 wrong numbers of function arguments

Type:
error

Reason:
The number of arguments used for referencing a function does not match the number of arguments defined in the function definition.

Argument:
none

1495 unresolved literals (missing LTORG)

Type:
error

Reason:
At the end of the program, or just before switching to another processor type, unresolved literals still remain.

Argument:
none

1500 instruction not allowed on

Type:
error

Reason:
Although the instruction is correct, it cannot be used with the selected member of the CPU family.

Argument:
none

1505 addressing mode not allowed on

Type:
error

Reason:
Although the addressing mode used is correct, it cannot be used with the selected member of the CPU family.

Argument:
none

1510 invalid bit position

Type:
error

Reason:
Either the number of bits specified is not allowed, or the command is not completely specified.

Argument:
none

1520 only ON/OFF allowed

Type:
error

Reason:
This pseudo command accepts as argument either `ON` or `OFF`

Argument:
none

1530 stack is empty or undefined

Type:
error

Reason:
It was tried to access a stack via a `POPV` instruction that was either never defined or already emptied.

Argument:
the name of the stack in question

1540 not exactly one bit set

Type:
error

Reason:
Not exactly one bit was set in a mask passed to the `BITPOS` function.

Argument:
none

1550 ENDSTRUCT without STRUCT

Type:
error

Reason:
An `ENDSTRUCT` instruction was found though there is currently no structure definition in progress.

Argument:
none

1551 open structure definition

Type:
error

Reason:
After end of assembly, not all `STRUCT` instructions have been closed with appropriate `ENDSTRUCT`s.

Argument:
the innermost, unfinished structure definition

1552 wrong ENDSTRUCT

Type:
error

Reason:
the name parameter of an `ENDSTRUCT` instruction does not correspond to the innermost open structure definition.

Argument:
none

1553 phase definition not allowed in structure definition

Type:
error

Reason:
What should I say about that? `PHASE` inside a record simply does not make sense and only leads to confusion...

Argument:
none

1554 invalid `STRUCT` directive

Type:
error

Reason:
Only `EXTNAMES` resp. `NOEXTNAMES` are allowed as directives of a `STRUCT` statement.

Argument:
the unknown directive

1560 instruction is not repeatable

Type:
error

Reason:
This machine instruction cannot be repeated via a `RPT` construct.

Argument:
none

1600 unexpected end of file

Type:
error

Reason:
It was tried to read past the end of a file with a `BINCLUDE` statement.

Argument:
none

1700 ROM-offset must be in range 0..63

Type:
error

Reason:
The ROM table of the 680x0 coprocessor has only 64 entries.

Argument:
none

1710 invalid function code

Type:
error

Reason:
The only function code arguments allowed are SFC, DFC, a data register, or a constant in the interval of 0..15 (only for 680x0 MMU).

Argument:
none

1720 invalid function code mask

Type:
error

Reason:
Only a number in the interval 0..15 can be used as function code mask (only for 680x0 MMU)

Argument:
none

1730 invalid MMU register

Type:
error

Reason:
The MMU does not have a register with this name (only for 680x0 MMU).

Argument:
none

1740 level must be in range 0..7

Type:
error

Reason:
The level for `PTESTW` and `PTESTR` must be a constant in the range of 0...7 (only for 680x0 MMU).

Argument:
none

1750 invalid bit mask

Type:
error

Reason:
The bit mask used for a bit field command has a wrong format (only for 680x0).

Argument:
none

1760 invalid register pair

Type:
error

Reason:
The register here defined cannot be used in this context, or there is a syntactic error (only for 680x0).

Argument:
none

1800 open macro definition

Type:
error

Reason:
An incomplete macro definition was found. Probably an `ENDM` was forgotten.

Argument:
none

1805 EXITM not called from within macro

Type:
error

Reason:
`EXITM` is designed to terminate a macro expansion. This instruction only makes sense within macros and an attempt was made to call it in the absence of macros.

Argument:
none

1810 more than 10 macro parameters

Type:
error

Reason:
A macro cannot have more than 10 parameters

Argument:
none

1815 macro double defined

Type:
error

Reason:
A macro was defined more than once in a program section.

Argument:
the multiply defined macro name.

1820 expression must be evaluatable in first pass

Type:
error

Reason:
The command used has an influence on the length of the emitted code, so that forward references cannot be resolved here.

Argument:
none

1830 too many nested IFs

Type:
error

Reason:
(no more implemented)

Argument:
none

1840 ELSEIF/ENDIF without IF

Type:
error

Reason:
A `ELSEIF-` or `ENDIF-` command was found, that is not preceded by an `IF-` command.

Argument:
none

1850 nested / recursive macro call

Type:
error

Reason:
(no more implemented)

Argument:
none

1860 unknown function

Type:
error

Reason:
The function invoked was not defined before.

Argument:
The name of the unknown function

1870 function argument out of definition range

Type:
error

Reason:
The argument does not belong to the allowed argument range associated to the referenced function.

Argument:
none

1880 floating point overflow

Type:
error

Reason:
Although the argument is within the range allowed to the function arguments, the result is not valid

Argument:
none

1890 invalid value pair

Type:
error

Reason:
The base-exponent pair used in the expression cannot be computed

Argument:
none

1900 instruction must not start on this address

Type:
error

Reason:
No jumps can be performed by the selected CPU from this address.

Argument:
none

1905 invalid jump target

Type:
error

Reason:
No jumps can be performed by the selected CPU to this address.

Argument:
none

1910 jump target not on same page

Type:
error

Reason:
Jump command and destination must be in the same memory page.

Argument:
none

1920 code overflow

Type:
error

Reason:
An attempt was made to generate more than 1024 code or data bytes in a single memory page.

Argument:
none

1925 address overflow

Type:
error

Reason:
The address space for the processor type actually used was filled beyond the maximum allowed limit.

Argument:
none

1930 constants and placeholders cannot be mixed

Type:
error

Reason:
Instructions that reserve memory, and instructions that define constants cannot be mixed in a single pseudo instruction.

Argument:
none

1940 code must not be generated in structure definition

Type:
error

Reason:
a `STRUCT` construct is only designed to describe a data structure and not to create one; therefore, no instructions are allowed that generate code.

Argument:
none

1950 parallel construct not possible here

Type:
error

Reason:
Either these instructions cannot be executed in parallel, or they are not close enough each other, to do parallel execution.

Argument:
none

1960 invalid segment

Type:
error

Reason:
The referenced segment cannot be used here.

Argument:
The name of the segment used.

1961 unknown segment

Type:
error

Reason:
The segment referenced with a `SEGMENT` command does not exist for the CPU used.

Argument:
The name of the segment used

1962 unknown segment register

Type:
error

Reason:
The segment referenced here does not exist (8086 only)

Argument:
none

1970 invalid string

Type:
error

Reason:
The string has an invalid format.

Argument:
none

1980 invalid register name

Type:
error

Reason:
The referenced register does not exist, or it cannot be used here.

Argument:
none

1985 invalid argument

Type:
error

Reason:
The command used cannot be performed with the `REP`-prefix.

Argument:
none

1990 indirect mode not allowed

Type:
error

Reason:
Indirect addressing cannot be used in this way

Argument:
none

1995 not allowed in current segment

Type:
error

Reason:
(no more implemented)

Argument:
none

1996 not allowed in maximum mode

Type:
error

Reason:
This register can be used only in minimum mode

Argument:
none

1997 not allowed in minimum mode

Type:
error

Reason:
This register can be used only in maximum mode

Argument:
none

2000 execution packet crosses address boundary

Type:
error

Reason:
An execution packet must not cross a 32-byte address boundary

Argument:
none

2001 multiple use of same execution unit

Type:
error

Reason:
One of the CPU's execution units was used more than once in an execution packet

Argument:
the name of the execution unit

2002 multiple long read operations

Type:
error

Reason:
An execution packet contains more than one long read operation, which is not allowed

Argument:
one of the functional units executing a long read

2003 multiple long write operations

Type:
error

Reason:
An execution packet contains more than one long write operation, which is not allowed

Argument:
one of the functional units executing a long write

2004 long read with write operation

Type:
error

Reason:
An execution packet contains both a long read and a write operation, which is not allowed.

Argument:
one of the execution units executing the conflicting operations

2005 too many reads of one register

Type:
error

Reason:
The same register was referenced more than four times in the same execution packet.

Argument:
the name of the register referenced too often

2006 overlapping destinations

Type:
error

Reason:
The same register was written more than one time in the same instruction packet, which is not allowed.

Argument:
the name of the register in question

2008 too many absolute branches in one execution packet

Type:
error

Reason:
An execution packet contains more than one direct branch, which is not allowed.

Argument:
none

2009 instruction cannot be executed on this unit

Type:
error

Reason:
This instruction cannot be executed on this functional unit.

Argument:
none

2010 invalid escape sequence

Type:
error

Reason:
The special character defined using a backslash sequence is not defined

Argument:
none

2020 invalid combination of prefixes

Type:
error

Reason:
The prefix combination here defined is not allowed, or it cannot be translated into binary code

Argument:
none

2030 constants cannot be redefined as variables

Type:
error

Reason:
A symbol that has once been declared as constant with `EQU` must not be modified afterwards with `SET`.

Argument:
the name of the symbol in question

2035 variables cannot be redefined as constants

Type:
error

Reason:
A symbol that has once been declared as variable with `SET` must not be redeclared afterwards as constant (e.g. with `EQU`.

Argument:
the name of the symbol in question

2040 structure name missing

Type:
error

Reason:
A structure's definition lacks the identifier name for the new structure

Argument:
none

2050 empty argument

Type:
error

Reason:
Empty strings must not be used in the argument list for this statement

Argument:
none

10001 error in opening file

Type:
fatal

Reason:
An error was detected while trying to open a file for input.

Argument:
description of the I/O error

10002 error in writing listing

Type:
fatal

Reason:
An error happened while AS was writing the listing file.

Argument:
description of the I/O error

10003 file read error

Type:
fatal

Reason:
An error was detected while reading a source file.

Argument:
description of the I/O error

10004 file write error

Type:
fatal

Reason:
While AS was writing a code or share file, an error happened.

Argument:
description of the I/O error

10006 heap overflow

Type:
fatal

Reason:
The memory available is not enough to store all the data needed by AS. Try using the DPMI or OS/2 version of AS.

Argument:
none

10007 stack overflow

Type:
fatal

Reason:
The program stack crashed, because too complex formulas, or a bad disposition of symbols and/or macros were used. Try again, using AS with the option `-A`.

Argument:
none

Appendix B

I/O Error Messages

The following error messages are generated not only by AS, but also by the auxiliary programs, like PLIST, BIND, P2HEX, and P2BIN. Only the most probable error messages are here explained. Should you meet an undocumented error message, then you probably met a program bug! Please inform us immediately about this!!

2 file not found
: The file requested does not exist, or it is stored on another drive.

3 path not found
: The path of a file does not exist, or it is on another drive.

4 too much open files
: There are no more file handles available to DOS. Increase their number changing the value associated to `FILES=` in the file `CONFIG.SYS`.

5 file access not allowed
: Either the network access rights do not allow the file access, or an attempt was done to rewrite or rename a protected file.

6 invalid file handler

12 invalid access mode

15 invalid drive letter
: The required drive does not exist.

16 The file cannot be deleted

17 RENAME cannot be done on this drive

100 Unexpected end of file
A file access tried to go beyond the end of file, although according to its structure this should not happen. The file is probably corrupted.

101 disk full
This is self explaining! Please, clean up !

102 ASSIGN failed

103 file not open

104 file not open for reading

105 file not open for writing

106 invalid numerical format

150 the disk is write-protected
When you don't use a hard disk as work medium storage, you should sometimes remove the protecting tab from your diskette!

151 unknown device
you tried to access a peripheral unit that is unknown to DOS. This should not usually happen, since the name should be automatically interpreted as a filename.

152 drive not ready
close the disk drive door.

153 unknown DOS function

154 invalid disk checksum
A bad read error on the disk. Try again; if nothing changes, reformat the floppy disk resp. begin to take care of your hard disk!

155 invalid FCB

156 position error
the diskette/hard disk controller has not found a disk track. See nr. 154 !

157 format unknown
DOS cannot read the diskette format

158 sector not found
As nr. 156, but the controller this time could not find a disk sector in the track.

159 end of paper
You probably redirected the output of AS to a printer. Assembler printout can be veeery long...

160 device read error
The operating system detected an unclassificable read error

161 device write error
The operating system detected an unclassificable write error

162 general failure error
The operating system has absolutely no idea of what happened to the device.

Appendix C

Frequently Asked Questions

In this chapter, I tried to collect some questions that arise very often together with their answers. Answers to the problems presented in this chapter might also be found at other places in this manual, but one maybe does not find them immediately...

Q: I am fed up with DOS. Are there versions of AS for other operating systems ?

A: Apart from the protected mode version that offers more memory when working under DOS, ports exist for OS/2 and Unix systems like Linux (currently in test phase). Versions that help operating system manufacturers located in Redmont to become even richer are currently not planned. I will gladly make the sources of AS available for someone else who wants to become active in this direction. The C variant is probably the best way to start a port into this direction. He should however not expect support from me that goes beyond the sources themselves...

Q: Is a support of the XYZ processor planned for AS?

A: New processors are appearing all the time and I am trying to keep pace by extending AS. The stack on my desk labeled "undone" however never goes below the 4 inch watermark... Wishes coming from users

of course play an important role in the decision which candidates will be done first. The internet and the rising amount of documentation published in electronic form make the acquisition of data books easier than it used to be, but it always becomes difficult when more exotic or older architectures are wanted. If the processor family in question is not in the list of families that are planned (see chapter 1), adding a data book to a request will have a highly positive influence. Borrowing books is also fine.

Q: Having a free assembler is really fine, but I now also had use for a disassembler...and a debugger...a simulator would also really be cool!

A: AS is a project I work on in leisure time, the time I have when I do not have to care of how to make my living. AS already takes a significant portion of that time, and sometimes I make a time-out to use my soldering iron, enjoy a Tangerine Dream CD, watch TV, or simply to fulfill some basic human needs... I once started to write the concept of a disassembler that was designed to create source code that can be assembled and that automatically separates code and data areas. I quickly stopped this project again when I realized that the remaining time simply did not suffice. I prefer to work on one good program than to struggle for half a dozen of mediocre apps. Regarded that way, the answer to the question is unfortunately "no"...

Q: The screen output of AS is messed up with strange characters, e.g. arrows and brackets. Why?

A: AS will by default use some ANSI control sequences for screen control. These sequences will appear unfiltered on your screen if you did not install an ANSI driver. Either install an ANSI driver or use the DOS command `SET USEANSI=N` to turn the sequences off.

Q: AS suddenly terminates with a stack overflow error while assembling my program. Did my program become to large?

A: Yes and No. Your program's symbol table has grown a bit unsymmetrically what lead to high recursion depths while accessing the table. Errors of this type especially happen in the 16-bit-OS/2 version of AS

which has a very limited stack area. Restart AS with the `-A` command line switch. If this does not help, too complex formula expression are also a possible cause of stack overflows. In such a case, try to split the formula into intermediate steps.

Q: It seems that AS does not assemble my program up to the end. It worked however with an older version of AS (1.39).

A: Newer versions of AS no longer ignore the `END` statement; they actually terminate assembly when an `END` is encountered. Especially older include files made by some users tended to contain an `END` statement at their end. Simply remove the superfluous `END` statements.

Q: I made an assembly listing of my program because I had some more complicated assembly errors in my program. Upon closer investigation of the listing, I found that some branches do not point to the desired target but instead to themselves!

A: This effect happens in case of forward jumps in the first pass. The formula parser does not yet have the target address in its symbol table, and as it is a completely independent module, it has to think of a value that even does not hurt relative branches with short displacement lengths. This is the current program counter itself...in the second pass, the correct values would have appeared, but the second pass did not happen due to errors in the first one. Correct the other errors first so that AS gets into the second pass, and the listing should look more meaningful again.

Q: Assembly of my program works perfectly, however I get an empty file when I try to convert it with P2HEX or P2BIN.

A: You probably did not set the address filter correctly. This filter by default cuts out an area ranging from 0 to 32 Kbytes. If you program contains memory chunks outside this range, they will be ignored. If your code is completely beyond the 32K barrier (this is commonplace for processors of the 65xx and 68xx series), you will get the result you just described. Simply set the address filter to a range that suits your needs (see the chapter dealing with P2BIN/P2HEX).

Q: I cannot enter the dollar character when using P2BIN or P2HEX under Unix. The automatic address range setting does not work, instead I get strange error messages.

A: Unix shells use the dollar character for expansion of shell variables. If you want to pass a dollar character to an application, prefix it with a backslash (\). In the special case of the address range specification for P2HEX and P2BIN, you may also use `0x` instead of the dollar character, which removes this prblen completely.

Q: I use AS on a Linux system, the loader program for my target system however runs on a Windows machine. To simplify things, both systems access the same network drive. Unfortunately, the Windows side refuses to read the hex files created by the Linux side :-(

A: Windows and Linux systems use slightly different formats for text files (hex files are a sort of text files). Windows terminates every line with the characters CR (carriage return) and LF (linefeed), however Linux only uses the linefeed. It depends on the Windows program's 'goodwill' whether it will accept text files in the Linux format or not. If not, it is possible to transfer the files via FTP in ASCII mode instead of a network drive. Alternatively, the hex files can be converted to the Windows format. For example, the program *unix2dos* can be used to do this, or a small script under Linux:

```
awk '{print $0"\r"}' test.hex >test_cr.hex
```

Appendix D

Pseudo-Instructions Collected

This appendix is designed as a quick reference to look up all pseudo instructions provided by AS. The list is ordered in two parts: The first part lists the instructions that are always available, and this list is followed by lists that enumerate the instructions additionally available for a certain processor family.

Instructions that are always available

=	:=	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	SAVE	SECTION	SEGMENT	SHARED
STRUC	STRUCT	SWITCH	TITLE	UNION
WARNING	WHILE			

There is an additional `SET` resp. `EVAL` instruction (in case `SET` is already a machine instruction).

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 6309

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

Motorola/Freescale 6805/68HC(S)08

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

Motorola 6809/Hitachi 6309

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

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

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	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 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
```

Appendix E

Predefined Symbols

To be exact, boolean symbols are just ordinary integer symbols with the difference that AS will assign only two different values to them (0 or 1, corresponding to False or True). AS does not store special symbols in the symbol table. For performance reasons, they are realized with hardcoded comparisons directly in the parser. They therefore do not show up in the assembly listing's symbol table. Predefined symbols are only set once at the beginning of a pass. The values of dynamic symbols may in contrast change during assembly as they reflect settings made with related pseudo instructions. The values added in parentheses give the value present at the beginning of a pass.

The names given in this table also reflect the valid way to reference these symbols in case-sensitive mode.

The names listed here should be avoided for own symbols; either one can define but not access them (special symbols), or one will receive an error message due to a double-defined symbol. The ugliest case is when the redefinition of a symbol made by AS at the beginning of a pass leads to a phase error and an infinite loop...

name	data type	definition	meaning
ARCHITECTURE	string	predef.	target platform AS was compiled for, in the style processor-manufacturer-operating system
BIGENDIAN	boolean	dyn.(0)	storage of constants MSB first ?
CASESENSITIVE	boolean	normal	case sensitivity in symbol names ?
CONSTPI	float	normal	constant Pi (3.1415.....)
DATE	string	predef.	date of begin of assembly
FALSE	boolean	predef.	0 = logically "false"
HASFPU	boolean	dyn.(0)	coprocessor instructions enabled ?
HASPMMU	boolean	dyn.(0)	MMU instructions enabled ?
INEXTMODE	boolean	dyn.(0)	XM flag set for 4 Gbyte address space ?
INLWORDMODE	boolean	dyn.(0)	LW flag set for 32 bit instructions ?
INMAXMODE	boolean	dyn.(0)	processor in maximum mode ?
INSUPMODE	boolean	dyn.(0)	processor in supervisor mode ?

Table E.1: Predefined Symbols - Part 1

name	data type	definition	meaning
INSRCMODE	boolean	dyn.(0)	processor in source mode ?
FULLPMMU	boolean	dyn.(0/1)	full PMMU instruction set allowed ?
LISTON	boolean	dyn.(1)	listing enabled ?
MACEXP	boolean	dyn.(1)	expansion of macro constructs in listing enabled ?
MOMCPU	integer	dyn. (68008)	number of target CPU currently set
MOMCPUNAME	string	dyn. (68008)	name of target CPU currently set
MOMFILE	string	special	current source file (including include files)
MOMLINE	integer	special	current line number in source file
MOMPASS	integer	special	number of current pass
MOMSECTION	string	special	name of current section or empty string if out of any section
MOMSEGMENT	string	special	name of address space currently selected with `SEGMENT`

Table E.2: Predefined Symbols - Part 2

name	data type	definition	meaning
NESTMAX	Integer	dyn.(256)	maximum nesting level of macro expansions
PADDING	boolean	dyn.(1)	pad byte field to even count ?
RELAXED	boolean	dyn.(0)	any syntax allowed integer constants ?
PC	integer	special	curr. program counter (Thomson)
TIME	string	predef.	time of begin of assembly (1. pass)
TRUE	integer	predef.	1 = logically "true"
VERSION	integer	predef.	version of AS in BCD coding, e.g. 1331 hex for version 1.33p1
WRAPMODE	Integer	predef.	shortened program counter assumed?
	integer	special	curr. program counter (Motorola, Rockwell, Microchip, Hitachi)
$	integer	special	curr. program counter (Intel, Zilog, Texas, Toshiba, NEC, Siemens, AMD)

Table E.3: Predefined Symbols - Part 3

Appendix F

Shipped Include Files

The distribution of AS contains a couple of include files. Apart from include files that only refer to a specific processor family (and whose function should be immediately clear to someone who works with this family), there are a few processor-independent files which include useful functions. The functions defined in these files shall be explained briefly in the following sections:

F.1 BITFUNCS.INC

This file defines a couple of bit-oriented functions that might be hardwired for other assemblers. In the case of AS however, thaey are implemented with the help of user-defined functions:

- *mask(start,bits)* returns an integer with *bits* bits set starting at position *start*;
- *invmask(start,bits)* returns one's complement to *mask()*;
- *cutout(x,start,bits)* returns *bits* bits masked out from *x* starting at position *start* without shifting them to position 0;
- *hi(x)* returns the second lowest byte (bits 8..15) of *x*;
- *lo(x)* returns the lowest byte (bits 8..15) of *x*;

- *hiword(x)* returns the second lowest word (bits 16..31) of *x*;
- *loword(x)* returns the lowest word (bits 0..15) of *x*;
- *odd(x)* returns TRUE if *x* is odd;
- *even(x)* returns TRUE if *x* is even;
- *getbit(x,n)* extracts bit *n* out of *x* and returns it as 0 or 1;
- *shln(x,size,n)* shifts a word *x* of length *size* to the left by *n* places;
- *shrn(x,size,n)* shifts a word *x* of length *size* to the right by *n* places;
- *rotln(x,size,n)* rotates the lowest *size* bits of an integer *x* to the left by *n* places;
- *rotrn(x,size,n)* rotates the lowest *size* bits of an integer *x* to the right by *n* places;

F.2 CTYPE.INC

This include file is similar to the C include file `ctype.h` which offers functions to classify characters. All functions deliver either TRUE or FALSE:

- *isdigit(ch)* becomes TRUE if *ch* is a valid decimal digit (0..9);
- *isxdigit(ch)* becomes TRUE if *ch* is a valid hexadecimal digit (0..9, A..F, a..f);
- *isupper(ch)* becomes TRUE if *ch* is an uppercase letter, excluding special national characters);
- *islower(ch)* becomes TRUE if *ch* is a lowercase letter, excluding special national characters);
- *isalpha(ch)* becomes TRUE if *ch* is a letter, excluding special national characters);

- *isalnum(ch)* becomes TRUE if *ch* is either a letter or a valid decimal digit;
- *isspace(ch)* becomes TRUE if *ch* is an 'empty' character (space, form feed, line feed, carriage return, tabulator);
- *isprint(ch)* becomes TRUE if *ch* is a printable character, i.e. no control character up to code 31;
- *iscntrl(ch)* is the opposite to *isprint()*;
- *isgraph(ch)* becomes TRUE if *ch* is a printable and visible character;
- *ispunct(ch)* becomes TRUE if *ch* is a printable special character (i.e. neither space nor letter nor number);

Appendix G

Acknowledgments

"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

If one decides to rewrite a chapter that has been out of date for two years, it is almost unavoidable that one forgets to mention some of the good ghosts who contributed to the success this project had up to now. The first "thank you" therefore goes to the people whose names I unwillingly forgot in the following enumeration!

The concept of AS as a universal cross assembler came from Bernhard (C.) Zschocke who needed a "student friendly", i.e. free cross assembler for his microprocessor course and talked me into extending an already existing 68000 assembler. The rest is history... The microprocessor course held at RWTH Aachen also always provided the most engaged users (and bug-searchers) of new AS features and therefore contributed a lot to today's quality of AS.

The internet and FTP have proved to be a big help for spreading AS and reporting of bugs. My thanks therefore go to the FTP admins (Bernd Casimir in Stuttgart, Norbert Breidor in Aachen, and Jürgen Meißburger in Jülich).

Especially the last one personally engaged a lot to establish a practicable way in Jülich.

As we are just talking about the ZAM: Though Wolfgang E. Nagel is not personally involved into AS, he is at least my boss and always puts at least four eyes on what I am doing. Regarding AS, there seems to be at least one that smiles...

A program like AS cannot be done without appropriate data books and documentation. I received information from an enormous amount of people, ranging from tips up to complete data books. An enumeration follows (as stated before, without guarantee for completelessness!):

Ernst Ahlers, Charles Altmann, Marco Awater, Len Bayles, Andreas Bolsch, Rolf Buchholz, Bernd Casimir, 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.

...and an ironic "thank you" to Rolf-Dieter-Klein and Tobias Thiel who demonstrated with their ASM68K how one should **not** do it and thereby indirectly gave me the impulse to write something better!

I did not entirely write AS on my own. AS contains the OverXMS routines from Wilbert van Leijen which can move the overlay modules into the extended memory. A really nice library, easy to use without problems!

The TMS320C2x/5x code generators and the file `STDDEF2x.INC` come from Thomas Sailer, ETH Zurich. It's surprising, he only needed one weekend to understand my coding and to implement the new code generator. Either that was a long nightshift or I am slowly getting old...

Appendix H

Changes since Version 1.3

- version 1.31:
  - additional MCS-51 processor type 80515. The number is again only stored by the assembler. The file `STDDEF51.INC` was extended by the necessary SFRs. **CAUTION!** Some of the 80515 SFRs have moved to other addresses!
  - additional support for the Z80 processor;
  - faster 680x0 code generator.
- version 1.32:
  - syntax for zero page addresses for the 65xx family was changed from `addr.z` to `<addr` (similar to 68xx);
  - additional support for the 6800, 6805, 6301, and 6811 processors;
  - the 8051 part now also understands `DJNZ, PUSH`, and `POP` (sorry);
  - the assembly listing now not also list the symbols but also the macros that have been defined;
  - additional instructions `IFDEF/IFNDEF` for conditional assembly based on the existence of a symbol;
  - additional instructions `PHASE/DEPHASE` to support code that shall be moved at runtime to a different address;

  - additional instructions `WARNING, ERROR`, and `FATAL` to print user-defined error messages;
  - the file `STDDEF51.INC` additionally contains the macro `USING` to simplify working with the MCS-51's register banks;
  - command line option `u` to print segment usage;

- version 1.33:
  - additionally supports the 6809 processor;
  - added string variables;
  - The instructions `TITLE, PRTINIT, PRTEXIT, ERROR`, `WARNING`, and `FATAL` now expect a string expression. Constants therefore now have to be enclosed in " instead of ' characters. This is also true for `DB`, `DC.B`, and `BYT`;
  - additional instruction `ALIGN` to align the program counter for Intel processors;
  - additional instruction `LISTING` to turn the generation of an assembly listing on or off;
  - additional instruction `CHARSET` for user-defined character sets.

- version 1.34:
  - the second pass is now omitted if there were errors in the first pass;
  - additional predefined symbol `VERSION` that contains the version number of AS;
  - additional instruction `MESSAGE` to generate additional messages under program control;
  - formula parser is now accessible via string constants;
  - if an error in a macro occurs, additionally the line number in the macro itself is shown;
  - additional function `UPSTRING` to convert a string to all upper-case.

- version 1.35:

  - additional function `TOUPPER` to convert a single character to upper case;
  - additional instruction `FUNCTION` for user-defined functions;
  - additional command line option `D` to define symbols from outside;
  - the environment variable `ASCMD` for commonly used command line options was introduced;
  - the program will additionally be checked for double usage of memory areas if the u option is enabled;
  - additional command line option `C` to generate a cross reference list.

- version 1.36:
  - additionally supports the PIC16C5x and PIC17C4x processor families;
  - the assembly listing additionally shows the nesting depth of include files;
  - the cross reference list additionally shows the definition point of a symbol;
  - additional command line option `A` to force a more compact layout of the symbol table.

- version 1.37:
  - additionally supports the processors 8086, 80186, V30, V35, 8087, and Z180;
  - additional instructions `SAVE` and `RESTORE` for an easier switching of some flags;
  - additional operators for logical shifts and bit mirroring;
  - command line options may now be negated with a plus sign;
  - additional filter AS2MSG for a more comfortable work with AS under Turbo-Pascal 7.0;
  - `ELSEIF` now may have an argument for construction of `IF-THEN--ELSE` ladders;

  - additional `CASE` construct for a more comfortable conditional assembly;
  - user-defined functions now may have more than one argument;
  - P2HEX can now additionally generate hex files in a format suitable for 65xx processors;
  - BIND, P2HEX, and P2BIN now have the same scheme for command line processing like AS;
  - additional switch `i` for P2HEX to select one out three possibilities for the termination record;
  - additional functions `ABS` and `SGN`;
  - additional predefined symbols `MOMFILE` and `MOMLINE`;
  - additional option to print extended error messages;
  - additional instruction `IFUSED` and `IFNUSED` to check whether a symbol has been used so far;
  - The environment variables `ASCMD, BINDCMD` etc. now optionally may contain the name of a file that provides more space for options;
  - P2HEX can now generate the hex formats specified by Microchip (p4);
  - a page length specification of 0 now allows to suppress automatic formfeeds in the assembly listing completely (p4);
  - symbols defined in the command line now may be assigned an arbitrary value (p5).

- version 1.38:

  - changed operation to multipass mode. This enables AS to generate optimal code even in case of forward references;
  - the 8051 part now also knows the generic `JMP` and `CALL` instructions;
  - additionally supports the Toshiba TLCS-900 series (p1);
  - additional instruction `ASSUME` to inform the assembler about the 8086's segment register contents (p2);

  - additionally supports the ST6 series from SGS-Thomson (p2);
  - ..and the 3201x signal processors from Texas Instruments (p2);
  - additional option `F` for P2HEX to override the automatic format selection (p2);
  - P2BIN now can automatically set the start resp. stop address of the address window by specifying dollar signs (p2);
  - the 8048 code generator now also knows the 8041/42 instruction extensions (p2);
  - additionally supports the Z8 microcontrollers (p3).

- version 1.39:

  - additional opportunity to define sections and local symbols;
  - additional command line switch `h` to force hexadecimal numbers to use lowercase;
  - additional predefined symbol `MOMPASS` to read the number of the currently running pass;
  - additional command line switch `t` to disable individual parts of the assembly listing;
  - additionally knows the L variant of the TLCS-900 series and the MELPS-7700 series from Mitsubishi (p1);
  - P2HEX now also accepts dollar signs as start resp. stop address (p2);
  - additionally supports the TLCS-90 family from Toshiba (p2);
  - P2HEX now also can output data in Tektronix and 16 bit Intel Hex format (p2);
  - P2HEX now prints warnings for address overflows (p2);
  - additional include file `STDDEF96.INC` with address definitions for the TLCS-900 series (p3);
  - additional instruction `READ` to allow interactive input of values during assembly (p3);
  - error messages are written to the STDERR channel instead of standard output (p3);

  - the `STOP` instruction missing for the 6811 is now available (scusi, p3);
  - additionally supports the μPD78(C)1x family from NEC (p3);
  - additionally supports the PIC16C84 from NEC (p3);
  - additional command line switch `E` to redirect error messages to a file (p3);
  - The MELPS-7700's 'idol' 65816 is now also available (p4);
  - the ST6 pseudo instruction `ROMWIN` has been removed was integrated into the `ASSUME` instruction (p4);
  - additionally supports the 6804 from SGS-Thomson (p4);
  - via the `NOEXPORT` option in a macro definition, it is now possible to define individually for every macro whether it shall appear in the `MAC` file or not (p4);
  - the meaning of `MACEXP` regarding the expansion of macros has changed slightly due to the additional `NOEXPAND` option in the macro definition (p4);
  - The additional `GLOBAL` option in the macro definition now additionally allows to define macros that are uniquely identified by their section name (p4).

- version 1.40:
  - additionally supports the DSP56000 from Motorola;
  - P2BIN can now also extract the lower resp. upper half of a 32-bit word;
  - additionally supports the TLCS-870 and TLCS-47 families from Toshiba (p1);
  - a prefixed `!` now allows to reach machine instructions hidden by a macro (p1);
  - the `GLOBAL` instruction now allows to export symbols in a qualified style (p1);
  - the additional `r` command line switch now allows to print a list of constructs that forced additional passes (p1);

- it is now possible to omit an argument to the `E` command line option; AS will then choose a fitting default (p1);
- the `t` command line option now allows to suppress line numbering in the assembly listing (p1);
- escape sequences may now also be used in ASCII style integer constants (p1);
- the additional pseudo instruction `PADDING` now allows to enable or disable the insertion of padding bytes in 680x0 mode (p2);
- `ALIGN` is now a valid instruction for all targets (p2);
- additionally knows the PIC16C64's SFRs (p2);
- additionally supports the 8096 from Intel (p2);
- `DC` additionally allows to specify a repetition factor (r3);
- additionally supports the TMS320C2x family from Texas Instruments (implementation done by Thomas Sailer, ETH Zurich, r3); P2HEX has been extended appropriately;
- an equation sign may be used instead of `EQU` (r3);
- additional `ENUM` instruction to define enumerations (r3);
- `END` now has a real effect (r3);
- additional command line switch `n` to get the internal error numbers in addition to the error messages (r3);
- additionally supports the TLCS-9000 series from Toshiba (r4);
- additionally supports the TMS370xxx series from Texas Instruments, including a new `DBIT` pseudo instruction (r5);
- additionally knows the DS80C320's SFR's (r5);
- the macro processor is now also able to include files from within macros. This required to modify the format of error messages slightly. If you use AS2MSG, replace it with the new version! (r5)
- additionally supports the 80C166 from Siemens (r5);
- additional `VAL` function to evaluate string expressions (r5);
- it is now possible to construct symbol names with the help of string expressions enclosed in braces (r5);

  - additionally knows the 80C167's peculiarities (r6);
  - the MELPS740's special page addressing mode is now supported (r6);
  - it is now possible to explicitly reference a symbol from a certain section by appending its name enclosed in brackets. The construction with an `@` sign has been removed! (r6)
  - additionally supports the MELPS-4500 series from Mitsubishi (r7);
  - additionally supports H8/300 and H8/300H series from Hitachi (r7);
  - settings made with `LISTING` resp. `MACEXP` may now be read back from predefined symbols with the same names (r7);
  - additionally supports the TMS320C3x series from Texas Instruments (r8);
  - additionally supports the SH7000 from Hitachi (r8);
  - the Z80 part has been extended to also support the Z380 (r9);
  - the 68K part has been extended to know the differences of the 683xx micro controllers (r9);
  - a label not any more has to be placed in the first row if it is marked with a double dot (r9);
  - additionally supports the 75K0 series from NEC (r9);
  - the additional command line option o allows to set a user-defined name for the code file (r9);
  - the `~~` operator has been moved to a bit more senseful ranking (r9);
  - `ASSUME` now also knows the 6809's DPR register and its implications (pardon, r9);
  - the 6809 part now also knows the 6309's secret extensions (r9);
  - binary constants now also may be written in a C-like notation (r9);

- version 1.41:

- the new predefined symbol `MOMSEGMENT` allows to inquire the currently active segment;
- `:=` is now allowed as a short form for `SET/EVAL`;
- the new command line switch `q` allows to force a "silent" assembly;
- the key word `PARENT` to reference the parent section has been extended by `PARENT0..PARENT9`;
- the PowerPC part has been extended by the microcontroller versions MPC505 and PPC403;
- symbols defined with `SET` or `EQU` may now be assigned to a certain segment (r1);
- the SH7000 part now also knows the SH7600's extensions (and should compute correct displacements...) (r1);
- the 65XX part now differentiates between the 65C02 and 65SC02 (r1);
- additionally to the symbol `MOMCPU`, there is now also a string symbol `MOMCPUNAME` that contains the processor's full name (r1);
- P2HEX now also knows the 32-bit variant of the Intel hex format (r1);
- additionally knows the 87C750's limitations (r2);
- the internal numbers for fatal errors have been moved to the area starting at 10000, making more space for normal error messages (r2);
- unused symbols are now marked with a star in the symbol table (r2);
- additionally supports the 29K family from AMD (r2);
- additionally supports the M16 family from Mitsubishi (r2);
- additionally supports the H8/500 family from Hitachi (r3);
- the number of data bytes printed per line by P2HEX can now be modified (r3);
- the number of the pass that starts to output warnings created by the `r` command line switch is now variable (r3);
- the macro processor now knows a `WHILE` statement that allows to repeat a piece of code a variable number of times (r3);

- the `PAGE` instruction now also allows to set the line with of the assembly listing (r3);
- CPU aliases may now be defined to define new pseudo processor devices (r3);
- additionally supports the MCS/251 family from Intel (r3);
- if the cross reference list has been enabled, the place of the first definition is given for double definitions of symbols (r3);
- additionally supports the TMS320C5x family from Texas Instruments (implementation done by Thomas Sailer, ETH Zurich, r3);
- the OS/2 version should now also correctly work with long file names. If one doesn't check every s**t personally... (r3);
- the new pseudo instruction `BIGENDIAN` now allows to select in MCS-51/251 mode whether constants should be stored in big endian or little endian format (r3);
- the 680x0 part now differentiates between the full and reduced MMU instruction set; a manual toggle can be done via the `FULLPMMU` instruction (r3);
- the new command line option `I` allows to print a list of all include files paired with their nesting level (r3);
- additionally supports the 68HC16 family from Motorola (r3);
- the `END` statement now optionally accepts an argument as entry point for the program (r3);
- P2BIN and P2HEX now allow to move the contents of a code file to a different address (r4);
- comments appended to a `SHARED` instruction are now copied to the share file (r4);
- additionally supports the 68HC12 family from Motorola (r4);
- additionally supports the XA family from Philips (r4);
- additionally supports the 68HC08 family from Motorola (r4);
- additionally supports the AVR family from Atmel (r4);
- to achieve better compatibility to the AS11 from Motorola, the pseudo instructions `FCB, FDB, FCC`, and `RMB` were added (r5);

- additionally supports the M16C from Mitsubishi (r5);
- additionally supports the COP8 from National Semiconductor (r5);
- additional instructions `IFB` and `IFNB` for conditional assembly (r5);
- the new `EXITM` instruction now allows to terminate a macro expansion (r5);
- additionally supports the MSP430 from Texas Instruments (r5);
- `LISTING` now knows the additional variants `NOSKIPPED` and `PURECODE` to remove code that was not assembled from the listing (r5);
- additionally supports the 78K0 family from NEC (r5);
- `BIGENDIAN` is now also available in PowerPC mode (r5);
- additional `BINCLUDE` instruction to include binary files (r5);
- additional `TOLOWER` and `LOWSTRING` functions to convert characters to lower case (r5);
- it is now possible to store data in other segments than `CODE`. The file format has been extended appropriately (r5);
- the `DS` instruction to reserve memory areas is now also available in Intel mode (r5);
- the `U` command line switch now allows to switch AS into a case sensitive mode that differentiates between upper and lower case in the names of symbols, user-defined functions, macros, macro parameters, and sections (r5);
- `SFRB` now also knows the mapping rules for bit addresses in the RAM areas; warnings are generated for addresses that are not bit addressable (r5);
- additional instructions `PUSHV` and `POPV` to save symbol values temporarily (r5);
- additional functions `BITCNT, FIRSTBIT, LASTBIT`, and `BITPOS` for bit processing (r5);
- the 68360 is now also known as a member of the CPU32 processors (r5);

- additionally supports the ST9 family from SGS-Thomson (r6);
- additionally supports the SC/MP from National Semiconductor (r6);
- additionally supports the TMS70Cxx family from Texas Instruments (r6);
- additionally supports the TMS9900 family from Texas Instruments (r6);
- additionally knows the 80296's instruction set extensions (r6);
- the supported number of Z8 derivatives has been extended (r6);
- additionally knows the 80C504's mask defects (r6);
- additional register definition file for Siemens' C50x processors (r6);
- additionally supports the ST7 family from SGS-Thomson (r6);
- the Tntel pseudo instructions for data disposal are now also valid for the 65816/MELPS-7700 (r6);
- for the 65816/MELPS-7700, the address length may now be set explicitly via prefixes (r6);
- additionally supports the 8X30x family from Signetics (r6);
- from now on, `PADDING` is enabled by default only for the 680x0 family (r7);
- the new predefined symbol `ARCHITECTURE` can now be used to query the platform AS was compiled for (r7);
- additional statements `STRUCT` and `ENDSTRUCT` to define data structures (r7);
- hex and object files for the AVR tools may now be generated directly (r7);
- `MOVEC` now also knows the 68040's control registers (r7);
- additional `STRLEN` function to calculate the length of a string (r7);
- additional ability to define register symbols (r7 currently only Atmel AVR);
- additionally knows the 6502's undocumented instructions (r7);
- P2HEX and P2BIN now optionally can erase the input files automatically (r7);

- P2BIN can additionally prepend the entry address to the resulting image (r7);
- additionally supports the ColdFire family from Motorola as a variation of the 680x0 core (r7);
- `BYT/FCB, ADR/FDB`, and `FCC` now also allow the repetition factor known from DC (r7);
- additionally supports Motorola's M*Core (r7);
- the SH7000 part now also knows the SH7700's extensions (r7);
- the 680x0 part now also knows the 68040's additional instructions (r7);
- the 56K part now also knows the instruction set extensions up to the 56300 (r7).
- the new `CODEPAGE` statement now allows to keep several character sets in parallel (r8);
- The argument variations for `CHARSET` have been extended (r8);
- New string functions `SUBSTR` and `STRSTR` (r8);
- additional `IRPC` statement in the macro processor (r8);
- additional `RADIX` statement to set the default numbering system for integer constants (r8);
- instead of `ELSEIF`, it is now valid to simply write `ELSE` (r8);
- == may be used as equality operator instead of = (r8);
- `BRANCHEXT` for the Philips XA now allows to automatically extend the reach of short branches (r8);
- debug output is now also possible in NoICE format (r8);
- additionally supports the i960 family from Intel (r8);
- additionally supports the μPD7720/7725 signal processors from NEC (r8);
- additionally supports the μPD77230 signal processor from NEC (r8);
- additionally supports the SYM53C8xx SCSI processors from Symbios Logic (r8);

  - additionally supports the 4004 from Intel (r8);
  - additionally supports the SC14xxx series of National (r8);
  - additionally supports the instruction extensions of the PPC 403GC (r8);
  - additional command line option `cpu` to set the default target processor (r8);
  - key files now also may be referenced from the command line (r8);
  - additional command line option `shareout` to set the output file for SHARED definitions (r8);
  - new statement `WRAPMODE` to support AVR processors with a shortened program counter (r8);
  - additionally supports the C20x instruction subset in the C5x part (r8);
  - hexadecimal address specifications for the tools now may also be made in C notation (r8);
  - the numbering system for integer results in `\{...}` expressions is now configurable via `OUTRADIX` (r8);
  - the register syntax for 4004 register pairs has been corrected (r8);
  - additionally supports the F^{2}MC8L family from Fujitsu (r8);
  - P2HEX now allows to set the minimum address length for S record addresses (r8);
  - additionally supports the ACE family from Fairchild (r8);
  - `REG` is now also allowed for PowerPCs (r8);
  - additional switch in P2HEX to relocate all addresses (r8);
  - The switch `x` now additionally allows a second level of detailness to print the source line in question (r8).

- version 1.42:
  - the default integer syntax for Atmel AVR is now the C Syntax;
  - additional command line option `olist` to set the list file's name and location;
  - additionally supports the F^{2}MC16L family from Fujitsu;

- additional instruction `PACKING` for the AVR family;
- additional implicit macro parameters `ALLARGS` and `ARGCOUNT`;
- additional instruction `SHIFT` to process variable macro argument lists;
- support for temporary symbols;
- additional instruction `MAXNEST` to set the maximum nesting depth of macro expansions;
- additional command line argument `noicemask` to control the amount of segments listed in a NoICE debug info file;
- additionally supports the 180x family from Intersil;
- additionally supports the 68HC11K4 address windowing;
- P2HEX now allows to vary the address field length of AVR HEX files;
- the new command line option `-gnuerrors` allows to output error messages in a GNU C-style format;
- additionally supports the TMS320C54x family from Texas Instruments;
- new macro option `INTLABEL`;
- added Atmel MegaAVR 8/16 instructions and register definitions;
- `ENDIF/ENDCASE` show the line number of the corresponding opening statement in the listing;
- the 8051 part now also supports the extended address space of the Dallas DS80C390;
- added nameless temporary smbols;
- additionally supports the undocumented 8085 instructions;
- improved structure handling;
- added EXPRTYPE() function;
- allow line continuation;
- integrated support for KCPSM/PicoBlaze provided by Andreas Wassatsch;

- additionally supports the 807x family from National Semiconductor;
- additionally supports the Intel 4040;
- additionally supports the Zilog eZ8;
- additionally supports the 78K2 family from NEC;
- additionally supports the KCPSM3 variant from Xilinx;
- additionally supports the LatticeMico8;
- additionally supports the 12X instruction extensions and the XGATE core of the 68HC12 family;
- additionally supports the Signetics 2650;
- additionally supports the COP4 family from National Semiconductor;
- additionally supports the HCS08 extensions by Freesacle;
- additionally supports the RS08 family by Freescale;
- additionally supports the Intel 8008;
- add another optional syntax for integer constants;
- added function `CHARFROMSTR`;
- additionally allow Q for octal constants in Intel mode;
- add another variant for temporary symbols;

Appendix I

Hints for the AS Source Code

As I already mentioned in the introduction, I release the source code of AS on request. The following shall give a few hints to their usage.

I.1 Language Preliminaries

In the beginning, AS was a program written in Turbo-Pascal. This was roughly at the end of the eighties, and there were a couple of reasons for this choice: First, I was much more used to it than to any C compiler, and compared to Turbo Pascal's IDE, all DOS-based C compilers were just crawling along. In the beginning of 1997 however, it became clear that things had changed: One factor was that Borland had decided to let its confident DOS developers down (once again, explicitly no 'thank you', you boneheads from Borland!) and replaced version 7.0 of Borland Pascal with something called 'Delphi', which is probably a wonderful tool to develop Windows programs which consist of 90% user interface and accidentaly a little bit of content, however completely useless for command-line driven programs like AS. Furthermore, my focus of operating systems had made a clear move towards Unix, and I probably could have waited arbitrarily long for a Borland Pascal for Linux (to all those remarking now that Borland would be working on something like that: this is *Vapourware*, dont́ believe them anything until you can go into a shop and actually buy it!). It was therefore clear that C was the way to go.

After this eperience what results the usage of 'island systems' may have, I put a big emphasize on portability while doing the translation to C; however, since AS for example deals with binary data in an exactly format and uses operating systen-specific functions at some places which may need adaptions when one compliles AS the first time for a new platform.

AS is tailored for a C compiler that conforms to the ANSI C standard; C++ is explicitly not required. If you are still using a compiler conforming to the outdated Kernighan&Ritchie standard, you should consider getting a newer compiler: The ANSI C standard has been fixed in 1989 and there should be an ANSI C compiler for every contemporary platform, maybe by using the old compiler to build GNU-C. Though there are some switches in the source code to bring it nearer to K&R, this is not an officially supported feature which I only use internally to support a quite antique Unix. Everything left to say about K&R is located in the file `README.KR`.

The inclusion of some additional features not present in the Pascal version (e.g. dynamically loadable message files, test suite, automatic generation of the documentation from *one* source format) has made the source tree substantially more complicated. I will attempt to unwire everything step by step:

I.2 Capsuling System dependencies

As I already mentioned, As has been tailored to provide maximum platform independence and portability (at least I believe so...). This means packing all platform dependencies into as few files as possible. I will describe these files now, and this section is the first one because it is probably one of the most important:

The Build of all components of AS takes place via a central `Makefile`. To make it work, it has to be accompanied by a fitting `Makefile.def` that gives the platform dependent settings like compiler flags. The subdirectory `Makefile.def-samples` contains a couple of includes that work for widespread platforms (but which need not be optimal...). In case your platform is not among them, you may take the file `Makefile.def.tmpl` as a starting point (and send me the result!).

A further component to capure system dependencies is the file `sysdefs.h`. Practically all compilers predefine a couple of preprocessor symbols that describe the target processor and the used operating system. For example, on a Sun Sparc under Solaris equipped with the GNU compiler, the symbols `__sparc` and `__SVR4`. `sysdefs.h` exploits these symbols to provide a homogeneous environment for the remaining, system-independent files. Especially, this covers integer datatypes of a specific length, but it may also include the (re)definition of C functions which are not present or non-standard-like on a specific platform. It's best to read this files yourself if you like to know which things may occur... Generally, the `#ifdef` statement are ordered in two levels: First, a specific processor platform is selected, the the operating systems are sorted out in such a section.

If you port AS to a new platform, you have to find two symbols typical for this platform and extend `sysdefs.h` accordingly. Once again, I'm interested in the result...

I.3 System-Independent Files

...represent the largest part of all modules. Describing all functions in detail is beyond the scope of this description (those who want to know more probably start studying the sources, my programming style isn't that horrible either...), which is why I can only give a short list at this place with all modules their function:

I.3.1 Modules Used by AS

as.c

This file is AS's root: it contains the *main()* function of AS, the processing of all command line options, the overall control of all passes and parts of the macro processor.

asmallg.c

This module processes all statements defined for all processor targets, e.g. `EQU` and `ORG`. The `CPU` pseudo-op used to switch among different processor targets is also located here.

asmcode.c

This module contains the bookkeping needed for the code output file. It exports an interface that allows to open and close a code file and offers functions to write code to (or take it back from) the file. An important job of this module is to buffer the write process, which speeds up execution by writing the code in larger blocks.

asmdebug.c

AS can optionally generate debug information for other tools like simulators or debuggers, allowing a backward reference to the source code. They get collected in this module and can be output after assembly in one of several formats.

asmdef.c

This modules only contains declarations of constants used in different places and global variables.

asmfnums.c

AS assigns internally assigns incrementing numbers for each used source file. These numbers are used for quick referencing. Assignment of numbers and the conversion between names and numbers takes place here.

asmif.c

Here ara ll routines located controlling conditional assembly. The most important exported variable is a flag called `IfAsm` which controls whether code generation is currently turned on or off.

asminclist.c

This module holds the definition of the list stucture that allows AS to print the nesting of include files to the assembly list file.

asmitree.c

When searching for the mnemonic used in a line of code, a simple linear comparison with all available machine instructions (as it is still done in most code generators, for reasons of simplicity and laziness) is not necessary the most effective method. This module defines two improved structures (binary tree and hash table) which provide a more efficient search and are destined to replace the simple linear search on a step-by-step basis...priorities as needed...

asmmac.c

Routines to store and execute macro constructs are located in this module. The real macro processor is (as already mentioned) in `as.c`.

asmpars.c

Here we really go into the innards: This module stores the symbol tables (global and local) in two binary trees. Further more, there is a quite large procedure `EvalExpression` which analyzes and evaluates a (formula) expression. The procedure returns the result (integer, floating point, or string) in a varaint record. However, to evaluate expressions during code generation, one should better use the functions `EvalIntExpression, EvalFloatExpression`, and `EvalStringExpression`. Modifications for tha esake of adding new target processors are unnecessary in this modules and should be done with extreme care, since you are touching something like 'AS's roots'.

asmsub.c

This module collects a couple of commonly used subroutines which primarily deal with error handling and 'advanced' string processing.

bpemu.c

As already mentioned at the beginning, AS originally was a program written in Borland Pascal. For some intrinsic functions of the compiler, it was simpler to emulate those than to touch all places in the source code where they are used. Well...

chunks.c

This module defines a data type to deal with a list of address ranges. This functionality is needed by AS for allocation lists; furthermore, P2BIN and P2HEX use such lists to warn about overlaps.

cmdarg.c

This module implements the overall mechanism of command line arguments. It needs a specification of allowed arguments, splits the command line and triggers the appropriate callbacks. In detail, the mechanism includes the following:

- Processing of arguments located in an environment variable or a corresponding file;
- Return of a set describing which command line arguments have not been processed;
- A backdoor for situations when an overlaying IDE converts the passed command line completely into upper or lower case.

codepseudo.c

You will find at this place pseudo instructions that are used by a subset of code generators. On the one hand, this is the Intel group of `DB..DT`, and on the other hand their counterparts for 8/16 bit CPUs from Motorola or Rockwell. Someone who wants to extend AS by a processor fitting into one of these groups can get the biggest part of the necessary pseudo instructions with one call to this module.

codevars.c

For reasons of memory efficiency, some variables commonly used by diverse code generators.

endian.c

Yet another bit of machine dependence, however one you do not have to spend attention on: This module automatically checks at startup whether a host machine is little or big endian. Furthermore, checks are made if the type definitions made for integer variables in `sysdefs.h` really result in the correct lengths.

headids.c

At this place, all processor families supported by AS are collected with their header IDs (see chapter 5.1) and the output format to be used by default by P2HEX. The target of this table is to centralize the addition of a new processor as most as possible, i.e. in contrast to earlier versions of AS, no further modifications of tool sources are necessary.

ioerrs.c

The conversion from error numbers to clear text messages is located here. I hope I'll never hit a system that does not define the numbers as macros, because I would have to rewrite this module completely...

nlmessages.c

The C version of AS reads all messages from files at runtime after the language to be used is clear. The format of message files is not a simple one, but instead a special compact and preindexed format that is generated at runtime by a program called 'rescomp' (we will talk about it later). This module is the counterpart to rescomp that reads the correct language part into a character field and offers functions to access the messages.

nls.c

This module checks which country-dependent settings (date and time format, country code) are present at runtime. Unfortunately, this is a highly operating system-dependend task, and currently, there are only three methods defines: The MS-DOS method, the OS/2 method and the typical Unix method via *locale* functions. For all other systems, there is unfortunately currently only `NO_NLS` available...

stdhandl.c

On the one hand, here is a special open function located knowing the special strings `!0...!2` as file names and creating duplicates of the standard file handles *stdin, stdout,* and *stderr*. On the other hand, investiagations are done whether the standard output has been redirected to a device or a file. On no-Unix systems, this unfortunately also incorporates some special operations.

stringlists.c

This is just a little 'hack' that defines routines to deal with linear lists of strings, which are needed e.g. in the macro processor of AS.

strutil.c

Some commonly needed string operations have found their home here.

version.c

The currently valid version is centrally stored here for AS and all other tools.

code????.c

These modules form the main part of AS: each module contains the code generator for a specific processor family.

I.3.2 Additional Modules for the Tools

hex.c

A small module to convert integer numbers to hexadecimal strings. It's not absolutely needed in C any more (except for the conversion of *long long* variables, which unfortunately not all `printf()`'s support), but it somehow survived the porting from Pascal to C.

p2bin.c

The sources of P2BIN.

p2hex.c

The sources of P2HEX.

pbind.c

The sources of BIND.

plist.c

The sources of PLIST.

toolutils.c

All subroutines needed by several tools are collected here, e.g. for reading of code files.

I.4 Modules Needed During the Build of AS

a2k.c

This is a minimal filter converting ANSI C source files to Kernighan-Ritchie style. To be exact: only function heads are converted, even this only when they are roughly formatted like my programming style. Noone should therefore think this were a universal C parser!

addcr.c

A small filter needed during installation on DOS- or OS/2-systems. Since DOS and OS/2 use a CR/LF for a newline, inc ontrast to the single LF of Unix systems, all assembly include files provided with AS are sent through this filter during assembly.

bincmp.c

For DOS and OS/2, this module takes the task of the *cmp* command, i.e. the binary comparison of files during the test run. While this would principally be possible with the *comp* command provided with the OS, *bincmp* does not have any nasty interactive questions (which seem to be an adventure to get rid of...)

findhyphen.c

This is the submodule in *tex2doc* providing hyphenation of words. The algorithm used for this is shamelessly stolen from TeX.

grhyph.c

The definition of hyphenation rules for the german language.

rescomp.c

This is AS's 'resource compiler', i.e. the tool that converts a readable file with string resources into a fast, indexed format.

tex2doc.c

A tool that converts the LaTeX documentation of AS into an ASCII format.

tex2html.c

A tool that converts the LaTeX documentation of AS into an HTML document.

umlaut.c and unumlaut.c

These tiny programs convert national special characters between their coding in ISO8859-1 (all AS files use this format upon delivery) and their system-specific coding. Apart from a plain ASCII7 variant, there are currently the IBM character sets 437 and 850.

ushyph.c

The definition of hyphenation rules for the english language.

I.5 Generation of Message Files

As already mentioned, the C source tree of AS uses a dynamic load principle for all (error) messages. In contrast to the Pasacl sources where all messages were bundled in an include file and compiled into the programs, this method eliminates the need to provide AS in multiple language variants; there is only one version which checks for the langugage to be used upon runtime and loads the corresponding component from the message files. Just to remind: Under DOS and OS/2, the `COUNTRY` setting is queried, while under Unix, the environment variables `LC_MESSAGES, LC_ALL,` and `LANG` are checked.

I.5.1 Format of the Source Files

A source file for the message compiler *rescomp* usually has the suffix `.res`. The message compiler generates one or two files from a source:

- a binary file which is read at runtime by AS resp. its tools

- optionally one further C header file assigning an index number to all messages. These index numbers in combination with an index table in the binary file allow a fast access to to individual messages at runtime.

The source file for the message compiler is a pure ASCII file and can therefore be modified with any editor. It consists of a sequence of control commands with parameters. Empty lines and lines beginning with a semicolon are ignored. Inclusion of other files is possible via the `Include` statement:

```
Include <Datei>
```

The first two statements in every source file must be two statements describing the languages defined in the following. The more important one is `Langs`, e.g.:

```
Langs DE(049) EN(001,061)
```

describes that two languages will be defined in the rest of the file. The first one shall be used under Unix when the language has been set to `DE` via environment variable. Similarly, It shall be used under DOS and OS/2 when the country code was set to 049. Similarly, the second set shall be used for the settings `DE` resp. 061 or 001. While multiple 'telephone numbers' may point to a single language, the assignment to a Unix language code is a one-to-one correspondence. This is no problem in practice since the `LANG` variables Unix uses describe subversions via appendices, e.g.:

```
de.de
de.ch
en.us
```

AS only compares the beginning of the strings and therefore still comes to the right decision. The `Default` statement defines the language that shall be used if either no language has been set at all or a language is used that is not mentioned in the asrgument list of `Langs`. This is typically the english language:

```
Default EN
```

These definitions are followed by an arbitrary number of `Message` statements, i.e. definitions of messages:

```
Message ErrName
 ": Fehler "
 ": error "
```

In case n languages were announced via the `Langs` statement, the message compiler takes **exactly** the following n as the strings to be stored. It is therefore impossible to leave out certain languages for individual messages, and an empty line following the strings should in no way be misunderstood as an end marker for the list; inserted lines between statements only serve purposes of better readability. It is however allowed to split individual messages across multiple lines in the source file; all lines except for the last one have to be ended with a backslash as continuation character:

```
Message TestMessage2
 "Dies ist eine" \
 "zweizeilige Nachricht"
 "This is a" \
 "two-line message"
```

As already mentioned, source files are pure ASCII files; national special characters may be placed in message texts (and the compiler will correctly pass them to the resulting file), a big disadvantage however is that such a file is not fully portable any more: in case it is ported to another system using a different coding for national special characters, the user will probably be confronted with funny characters at runtime...special characters should therefore always be written via special sequences borrowed from HTML resp. SGML (see table I.1). Linefeeds can be inserted into a line via `\n`, similar to C.

I.6 Creation of Documentation

A source distribution of AS contains this documentation in LaTeX format only. Other formats are created from this one automatically via tools provided with AS. One reason is to reduce the size of the source distribution, another reason is that changes in the documentation only have to be made once, avoiding inconsistencies.

LaTex was chosen as the master format because...because...because it's been there all the time before. Additionally, TeX is almost arbitrarily portable

Sequence...	results in...
`ä ö ü`	”a ”o ”u (Umlauts)
`Ä Ö Ü`	”A ”O ”U
`ß`	”s (sharp s)
`à è ì ò`	á é í ó
`ù`	ú
`À È Ì Ò`	Á É Í Ó
`Ù`	Ú (Accent grave)
`á é í ó`	à è ì ò
`ú`	ù
`Á É Í Ó`	À È Ì Ò
`Ú`	Ù (Accent agiu)
`â ê î ô`	â ê î ô
`û`	û
`Â Ê Î Ô`	Â Ê Î Ô
`Û`	Û (Accent circonflex)
`ç Ç`	ç Ç(Cedilla)
`ñ Ñ`	ñ Ñ
`å Å`	åÅ
`æ &Aelig;`	æÆ
`¿ ¡`	inverted ! or ?

Table I.1: Syntax for special character in *rescomp*

and fits quite well to the demands of AS. A standard distribution therefore allows a nice printout on about any printer; for a conversion to an ASCII file that used to be part of earlier distributions, the converter *tex2doc* is included, additionally the converter *tex2html* allowing to put the manual into the Web.

Generation of the documentation is started via a simple

```
make docs
```

The two converters mentioned are be built first, then applied to the TeX documentation and finally, LaTeX itself is called. All this of course for all languages...

I.7 Test Suite

Since AS deals with binary data of a precisely defined structure, it is naturally sensitive for system and compiler dependencies. To reach at least a minimum amount of secureness that everything went right during compilation, a set of test sources is provided in the subdirectory `tests` that allows to test the freshly built assembler. These programs are primarily trimmed to find faults in the translation of the machine instruction set, which are commonplace when integer lenghts vary. Target-independent features like the macro processors or conditional assembly are only casually tested, since I assume that they work everywhere when they work for me...

The test run is started via a simple *make test*. Each test program is assembled, converted to a binary file, and compared to a reference image. A test is considered to be passed if and only if the reference image and the newly generated one are identical on a bit-by-bit basis. At the end of the test, the assembly time for every test is printed (those who want may extend the file BENCHES with these results), accompanied with a success or failure message. Track down every error that occurs, even if it occurs in a processor target you are never going to use! It is always possible that this points to an error that may also come up for other targets, but by coincidence not in the test cases.

I.8 Adding a New Target Processor

The probably most common reason to modify the source code of AS is to add a new target processor. Apart from adding the new module to the Makefile, there are few places in other modules that need a modification. The new module will do the rest by registering itself in the list of code generators. I will describe the needed steps in a cookbook style in the following sections:

Choosing the Processor's Name

The name chosen for the new processor has to fulfill two criterias:

1. The name must not be already in use by another processor. If one starts AS without any parameters, a list of the names already in use will be printed.

2. If the name shall appear completely in the symbol `MOMCPU`, it may not contain other letters than A..F (except right at the beginning). The variable `MOMCPUNAME` however will always report the full name during assembly. Special characters are generally disallowed, lowercase letters will be converted by the `CPU` command to uppercase letters and are therefore senseless in the processor name.

The first step for registration is making an entry for the new processor (family) in the file `headids.c`. As already mentioned, this file is also used by the tools and specifies the code ID assigned to a processor family, along with the default hex file format to be used. I would like to have some coordination before choosing the ID...

Definition of the Code Generator Module

The unit's name that shall be responsible for the new processor should bear at least some similarity to the processor's name (just for the sake of uniformity) and should be named in the style of `code.....` The head with include statements is best taken from another existing code generator.

Except for an initialization function that has to be called at the begginning of the `main()` function in module `as.c`, the new module neither has to export variables nor functions as the complete communication is done at runtime via indirect calls. They are simply done by a call to the function `AddCPU` for each processor type that shall be treated by this unit:

```
CPUxxxx:=AddCPU('XXXX',SwitchTo_xxxx);
```

`'XXXX'` is the name chosen for the processor which later must be used in assembler programs to switch AS to this target processor. `SwitchTo_xxxx` (abbreviated as the "switcher" in the following) is a procedure without parameters that is called by AS when the switch to the new processor actually takes place. `AddCPU` delivers an integer value as result that serves as an internal "handle" for the new processor. The global variable `MomCPU` always

contains the handle of the target processor that is currently set. The value returned by `AddCPU` should be stored in a private variable of type `CPUVar` (called `CPUxxxx` in the example above). In case a code generator module implements more than one processor (e.g. several processors of a family), the module can find out which instruction subset is currently allowed by comparing `MomCPU` against the stored handles.

The switcher's task is to "reorganize" AS for the new target processor. This is done by changing the values of several global variables:

- `ValidSegs`: Not all processors have all address spaces defined by AS. This set defines which subset the `SEGMENT` instruction will enable for the currently active target processor. At least the `CODE` segment has to be enabled. The complete set of allowed segments can be looked up the file `fileformat.h` (`Seg....` constants).

- `SegInits`: This array stores the initial program counter values for the individual segments (i.e. the values the program counters will initially take when there is no `ORG` statement). There are only a few exceptions (like logically separated address spaces that physically overlap) which justify other initial values than 0.

- `Grans`: This array specifies the size of the smallest addressable element in bytes for each segment, i.e. the size of an element that increases an address by 1. Most processors need a value of 1, even if they are 16- or 32-bit processors, but the PICs and signal processors are cases where higher values are required.

- `ListGrans`: This array specifies the size of byte groups that shall be shown in the assembly listing. For example, instruction words of the 68000 are always 2 bytes long though the code segment's granularity is 1. The `ListGran` entry therefore has to be set to 2.

- `SegLimits`: This array stores the highest possible address for each segment, e.g. 65535 for a 16-bit address space. This array need not be filled in case the code generator takes over the `ChkPC` method.

- `ConstMode`: This variable may take the values `ConstModeIntel`, `ConstModeMoto`, or `ConstModeC` and rules which syntax has to be used to specify the base of integer constants.

- `PCSymbol`: This variable contains the string an assembler program may use to to get the current value of the program counter. Intel processors for example usually use a dollar sign.

- `TurnWords`: If the target processor uses big-endian addressing and one of the fields in `ListGran` is larger than one, set this flag to true to get the correct byte order in the code output file.

- `SetIsOccupied`: Some processors have a `SET` machine instruction. AS will allow `SET` instructions to pass through to the code generator and instead use `EVAL` if this flag is set.

- `HeaderID`: This variable contains the ID that is used to mark the current processor family in the the code output file (see the description of the code format described by AS). I urge to contact me before selecting the value to avoid ambiguities. Values outside the range of $01..$7f should be avoided as they are reserved for special purposes (like a future extension to allow linkable code). Even though this value is still hard-coded in most code generators, the preferred method is now to fetch this value from `headids.h` via `FindFamilyByName`.

- `NOPCode`: There are some situations where AS has to fill unused code areas with NOP statements. This variable contains the machine code of the NOP statement.

- `DivideChars`: This string contains the characters that are valid separation characters for instruction parameters. Only extreme exotics like the DSP56 require something else than a single comma in this string.

- `HasAttrs`: Some processors like the 68k series additionally split an instruction into mnemonic and attribute. If the new processor also does something like that, set this flag to true and AS will deliver the instructions' components readily split in the string variables `OpPart` and `AttrPart`. If this flag is however set to false, no splitting will take place and the instruction will be delivered as a single piece in `OpPart`. `AttrPart` will stay empty in this case. One really should set this flag to false if the target processor does not have attributes as one otherwise looses the opportunity to use macros with a name containing dots (e.g. to emulate other assemblers).

- `AttrChars`: In case `HasAttrs` is true, this string has to contain all characters that can separate mnemonic and attribute. In most cases, this string only contains a single dot.

Do not assume that any of these variables has a predefined value; set them **all**!!

Apart from these variables, some function pointers have to be set that form the link form AS to the "active" parts of the code generator:

- `MakeCode`: This routine is called after a source line has been split into mnemonic and parameters. The mnemonic is stored into the variable `OpPart`, and the parameters can be looked up in the array `ArgStr`. The number of arguments may be read from `ArgCnt`. The binary code has to be stored into the array `BAsmCode`, its length into `CodeLen`. In case the processor is word oriented like the 68000 (i.e. the `ListGran` element corresponding to the currently active segment is 2), the field may be addressed wordwise via `WAsmCode`. There is also `DAsmCode` for extreme cases... The code length has to be given in units corresponding to the current segment's granularity.

- `SwitchFrom`: This parameter-less procedure enables the code generator module to do "cleanups" when AS switches to another target processor. This hook allows e.g. to free memory that has been allocated in the generator and that is not needed as long as the generator is not active. It may point to an empty procedure in the simplest case. One example for the usage of this hook is the module `CODE370` that builds its instruction tables dynamically and frees them again after usage.

- `IsDef`: Some processors know additional instructions that impose a special meaning on a label in the first row like `EQU` does. One example is the `BIT` instruction found in an 8051 environment. This function has to return TRUE if such a special instruction is present. In the simplest case (no such instructions), the routine may return a constant FALSE.

Optionally, the code generator may additionally set the following function pointers:

- `ChkPC` : Though AS internally treats all program counters as either 32 or 64 bits, most processors use an address space that is much smaller. This function informs AS whether the current program counter has exceeded its allowed range. This routine may of course be much more complicated in case the target processor has more than one address space. One example is in module `code16c8x.c`. In case everything is fine, the function has to return TRUE, otherwise FALSE. The code generator only has to implement this function if it did not set up the array `SegLimits`. This may e.g. become necessary when the allowed range of addresses in a segment is non-continuous.

- `InternSymbol` : Some processorcs, e.g. such with a register bank in their internal RAM, predefine such 'registers' as symbols, and it wouldn't make much sense to define them in a separate include file with 256 or maybe more `EQU`s. This hook allows access to the code generator of AS: It obtains an expression as an ASCII string and sets up the passed structure of type *TempResult* accordingly when one of these 'built-in' symbols is detected. The element `Typ` has to be set to `TempNone` in case the check failed. Errors messages from this routine should be avoided as unidentified names could signify ordinary symbols (the parser will check this afterwards). Be extreme careful with this routine as it allows you to intervene into the parser's heart!

By the way: People who want to become immortal may add a copyright string. This is done by adding a call to the procedure `AddCopyright` in the module's initialization part (right next to the `AddCPU` calls):

```
AddCopyright(
   "Intel 80986 code generator (C) 2010 Jim Bonehead");
```

The string passed to `AddCopyright` will be printed upon program start in addition to the standard message.

If needed, the unit may also use its initialization part to hook into a list of procedures that are called prior to each pass of assembly. Such a need for example arises when the module's code generation depends on certain flags that can be modified via pseudo instructions. An example is a processor that can operate in either user or supervisor mode. In user mode, some instructions are disabled. The flag that tells AS whether the following code executes in

user or supervisor mode might be set via a special pseudo instruction. But there must also be an initialization that assures that all passes start with the same state. The hook offered via `InitPassProc` offers a chance to do such initializations. The principle is similar to the redirection of an interrupt vector: the unit saves the old value prior to pointing the procedure variable to its own routine (the routine must be parameter-less and `FAR` coded). The new routine first calls the old chain of procedures and afterwards does its own operations.

The function chain built up via `CleanUpProc` works similar to `InitPassProc`: It enables code generators to do clean-ups after assembly (e.g. freeing of literal tables). This makes sense when multiple files are assembled with a single call of AS. Otherwise, one would risk to have 'junk' in tables from the previous run. No module currently exploits this feature.

Writing the Code Generator itself

Now we finally reached the point where your creativity is challenged: It is up to you how you manage to translate mnemonic and parameters into a sequence of machine code. The symbol tables are of course accessible (via the formula parser) just like everything exported from `ASMSUB`. Some general rules (take them as advises and not as laws...):

- Try to split the instruction set into groups of instructions that have the same operand syntax and that differ only in a few bits of their machine code. For example, one can do all instructions without parameters in a single table this way.
- Most processors have a fixed spectrum of addressing modes. Place the parsing of an address expression in a separate routine so you an reuse the code.
- The subroutine `WrError` defines a lot of possible error codes and can be easily extended. Use this! It is no good to simply issue a "syntax error" on all error conditions!

Studying other existing code generators should also prove to be helpful.

Modifications of Tools

A microscopic change to the tolls' sources is still necessary, namely to the routine `Granularity()` in `toolutils.c`: in case one of the processor's address spaces has a granularity different to 1, the swich statement in this place has to be adapted accordingly, otherwise PLIST, P2BIN, and P2HEX start counting wrong...

I.9 Localization to a New Language

You are interested in this topic? Wonderful! This is an issue that is often neglected by other programmers, especially when they come from the country on the other side of the big lake...

The localization to a new language can be split into two parts: the adaption of program messages and the translation of the manual. The latter one is definitely a work of gigantic size, however, the adaption of program messages should be a work doable on two or three weekends, given that one knows both the new and one of the already present messages. Unfortunately, this translation cannot be done on a step-by-step basis because the resource compiler currently cannot deal with a variable amount of languages for different messages, so the slogan is 'all or nothing'.

The first oeration is to add the new language to `header.res`. The two-letter-abbreviation used for this language is best fetched from the nearest Unix system (in case you don't work on one anyway...), the international telephone prefix from a DOS manual.

When this is complete, one can rebuild all necessary parts with a simple *make* and obtains an assembler that supports one more language. Do not forget to forward the results to me. This way, all users will benefit from this with the next release :-)

Bibliography

[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).
(part of the AVR tools documentation)

[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.
published by Hitachi

[21] Hitachi Microcomputer:
H8/300H Series Programming Manual.
(21-032, no year of release given)

[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, no year of release given)

[26] Hitachi Ltd.:
H8/534,H8/536 Hardware Manual.
(21-19A, no year of release given)

[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.
(no year of release given)

[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.
(available via World Wide Web:
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] documentation about the M16/M32-developer's package from 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.
Second Edition, Prentice-Hall, Englewood Cliffs 1989

[50] Motorola Inc.:
MC68851 Paged Memory Management Unit User's Manual.
Second Edition, Prentice-Hall, Englewood Cliffs 1989,1988

[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.:
CPU12 Reference Manual.
1st edition, 1996

[60] Motorola Inc.:
CPU08 Reference Manual.
Rev. 1 (no year of release given im PDF-File)

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

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

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

[64] Motorola Inc.:
DSP56300 24-Bit Digital Signal Processor Family Manual.
Rev. 0 (no year of release given im PDF-File)

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

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

[67] National Semiconductor:
COP800 Assembler/Linker/Librarian User's Manual.
Customer Order Number COP8-ASMLNK-MAN
NSC Publication Number 424421632-001B
August 1993

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

[69] National Semiconductor:
SC14xxx DIP commands Reference guide.
Application Note AN-D-031, Version 0.4, 12-28-1998

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

[71] NEC Corp.:
µpD70108/µpD70116/µpD70208/µpD70216/µpD72091 Data Book.
(no year of release given)

[72] NEC Electronics Europe GmbH:
User's Manual µCOM-87 AD Family.
(no year of release given)

[73] NEC Corp.:
μCOM-75x Family 4-bit CMOS Microcomputer User's Manual.
Vol. I+II (no year of release given)

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

[75] NEC Corp.:
Digital Signal Processor Product Description.
PDDSP.....067V20 (no year of release given)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[112] 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

[113] data sheets from Zilog about the Z80 family

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

[115] Zilog Inc.:
Discrete Z8 Microcontrollers Databook.
(no year of release given)

[116] Zilog Inc.:
Z380 CPU Central Processing Unit User's Manual.
(no year of release given)

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

Index

ADR, 113
ALIGN, 119
ASCII, 117
ASCIZ, 117
ASSUME, 100

BFLOAT, 115
BIGENDIAN, 96
BINCLUDE, 151
BIT, 68
BLOCK, 118
BRANCHEXT, 108
BSS, 118
BYT, 113
BYTE, 113

CASE, 136
CHARSET, 71
CODEPAGE, 72
CONSTANT, 65
CPU, 79

DATA, 116
DB, 111
DBIT, 69
DC, 109
DC8, 113
DD, 111
DEPHASE, 98
DFS, 117
DOTTEDSTRUCTS, 131
DOUBLE, 114, 115
DQ, 111
DS, 110, 112
DS16, 119
DS8, 112
DSB, 118
DSW, 118
DT, 111
DW, 111
DW16, 114

EFLOAT, 115
ELSE, 134
ELSECASE, 136
ELSEIF, 134
END, 154
ENDCASE, 136
ENDIF, 134
ENDM, 120
ENDS, 131
ENDSECTION, 143
ENDSTRUCT, 131
ENUM, 73
EQU, 65
ERROR, 152
EXITM, 128
EXTENDED, 114
EXTMODE, 95

FATAL, 152
FB, 116

FCB, 113
FCC, 117
FDB, 113
FLOAT, 115
FORWARD, 148
FPU, 92
FULLPMMU, 93
FUNCTION, 129
FW, 116

GLOBAL, 147

IF, 134
IFB, 135
IFDEF, 135
IFEXIST, 135
IFNB, 135
IFNDEF, 135
IFNEXIST, 135
IFNUSED, 135
IFUSED, 135
INCLUDE, 150
IRP, 126
IRPC, 126

LABEL, 68
LISTING, 139
LIV, 70
LONG, 114
LQxx, 115
LTORG, 119
LWORDMODE, 95

MACEXP, 139
MACRO, 120
MAXMODE, 94
MAXNEST, 129
MESSAGE, 152

NAMEREG, 70
NEWPAGE, 138

ORG, 74
OUTRADIX, 141

PACKING, 94
PADDING, 93
PAGE, 137
PHASE, 98
PMMU, 92
POPV, 73
PORT, 70
PRTEXIT, 140
PRTINIT, 140
PUBLIC, 147
PUSHV, 73

Qxx, 115

RADIX, 141
READ, 153
REG, 70
register symbols, 61
RELAXED, 154
REPT, 127
RES, 118
RESTORE, 99
RIV, 70
RMB, 117
RSTRING, 117

SAVE, 99
SECTION, 143
SEGMENT, 97
SET, 65
SFR, 67
SFRB, 67
SHARED, 30, 62, 150
SHIFT, 128

SINGLE, 114
SPACE, 118
SRCMDE, 95
STRING, 117
STRUC, 131
STRUCT, 131
SUPMODE, 92
SWITCH, 136

TFLOAT, 115
TITLE, 141

UNION, 131

WARNING, 152
WHILE, 127
WORD, 114
WRAPMODE, 96

XSFR, 67

YSFR, 67

ZERO, 116