AssemPro

AMIGA

```
Debugger
T·S··210···XNZVC                          00054A88 START:MOVEA.L #DATAS,A2
0000000000000000                          00054A8E  MOVE.W #$14,D2
PC  = $54ABE                              00054A92  SUB.W #2,D2
SSP = $80000                              00054A96 LOOP1:MOVE.W #0,D4
USP = $5892C                              00054A9A  MOVE.W D2,D3
L:D0                    =          0      00054A9C LOOP2:MOVE.W D4,D5
L:D1                    =          0      00054A9E  LSL.W #1,D5
L:D2                    =        $12      00054AA0  MOVE.W 0(A2,D5.W),D6
L:D3                    =      $FFFF      00054AA4  MOVE.W 2(A2,D5.W),D7
L:D4                    =        $13      00054AA8  CMP.W D7,D6
L:D5                    =        $24      00054AAA  BLE CONT1
L:D6                    =        $64      00054AAE SWAPIT:MOVE.W D6,2(A2,D5.W)
L:D7                    =        $42      00054AB2  MOVE.W D7,0(A2,D5.W)
L:A0                    =          0      00054AB6 CONT1:ADD.W #1,D4
L:A1                    =          0      00054ABA  DBRA D3,LOOP2
L:A2                    =     $54AC4      00054ABE  DBRA D2,LOOP1
L:A3                    =          0      00054AC2  BREAKPOINT
L:A4                    =          0      00054AC4 DATAS:ORI.W #6,-(A0)
L:A5                    =          0      00054AC8  DC.W $A
L:A6                    =          0      00054ACA  ORI.B #$56,$5A(A6)
L:4                     =       $676      00054AD0  ORI.B #$14,(A0)
Segment-List: $54A84                      00054AD4  ORI.W #$1A,D6
Task: $578D8                              00054AD8  ORI.B #$32,-(A4)
```

A Data Becker Product

AssemPro
AMIGA

```
Assembler
Editor: SORT.ASM

START:  MOVE.L  #DATAS,A2           ; address of the array datas
        MOVE.W  #20,D2              ; number of elements N=20
        SUB.W   #2,D2               ; (N-1) and -1 because of DBRA

LOOP1:  MOVE.W  #0,D4               ; I array across pointer
        MOVE.W  D2,D3               ; K as upper loop pointer
LOOP2:  MOVE.W  D4,D5               ; times 2 bytes per array element
        LSL.W   #1,D5               ; by shifting register left
        MOVE.W  0(A2,D5.W),D6       ; A[I] to D6
        MOVE.W  2(A2,D5.W),D7       ; A[I+1] to D7
        CMP.W   D7,D6               ; destination -COMPARE- source->
        BLE     CONT1               ; D6 < D7? If so, no exchange
SWAPIT:MOVE.W   D6,2(A2,D5.W)       ; D6->A[i+1]
        MOVE.W  D7,0(A2,D5.W)       ; D7->A[i]
CONT1:  ADD.W   #1,D4               ; increment the array pointer
        DBRA    D3,LOOP2            ; For I = 1 to K
        DBRA    D2,LOOP1            ; For K = N-1 to 1
        RTS
DATAS:  DC.W 100,96,6,10,46,86,90,16,20,70,26,36,50
        DC.W 56,76,60,80,40,60,66
```

By Peter Schulz

A Data Becker Product

Copyright Notice

AssemPro Amiga is copyrighted by Data Becker and Abacus Software, Inc. You should carefully read all the terms and conditions of this agreement prior to using this software. By opening this sealed disk package, you are agreeing to accept the terms of this agreement, which includes the software license and software disclaimer of warranty. If you do not agree to the terms of this agreement, do not open the disk package. Promptly return the unopened disk package and other items that are part of this product to the place where you obtained them for a full refund.

1. COPYRIGHT: You may not sublicense, assign or transfer the license or the program except as expressly provided in this Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations here under is void.

2. LICENSE: You have the non-exclusive right to use this software. You may not distribute copies of the software or documentation to others. This software can only be used on a single computer. You may not modify of translate the software or user manual without the prior written consent of Abacus Software, Inc. You may not use, copy, modify, or transfer the software or any copy, modification or merged portion in whole or in part, except as expressly provided for in this license.

3. BACK-UP AND TRANSFER: You may make backup copies of AssemPro Amiga, but not of the documentation. Each backup copy must include the copyright notices. Installation of AssemPro Amiga on your hard disk is permitted subject to these conditions. You may not sell, loan or give away copies of AssemPro Amiga.

4. TERM: This license is effective until terminated. You may terminate it at any time by destroying the software together with all copies, modifications and merged portions in any form. It will also terminate upon conditions set forth elsewhere in this Agreement or if you fail to comply with any term or condition of this Agreement. You agree upon such termination to destroy the software together with all copies, modification and merged portions in any form.

5. LIMITED WARRANTY: This software is provided "as is" without warranty of any kind, either expressed or implied, including, but not limited to the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the software is with you. Should the software prove defective, you (and not Abacus Software, Inc. or an authorized personal computer dealer) assume the entire cost of all necessary servicing, repair or correction. Some states do not allow the exclusion of implied warranties, so the above exclusion may not apply to you. This warranty gives you specific legal rights and you may also have other rights which vary from state to state.

Abacus Software, Inc. does not warrant that the functions contained in the software will meet your requirements or that the operation of the software will be uninterrupted or error free. However, Abacus Software, Inc. warrants the supplied diskettes to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of delivery to you as evidenced by a copy of your receipt. LIMITATIONS OF REMEDIES: Abacus Software, Inc. entire liability and your exclusive remedy shall be: a. the replacement of any diskettes not meeting Abacus Software, Inc. "Limited Warranty" and which are returned to Abacus Software, Inc. or an authorized Computer Dealer with a copy of your receipt, or, b. if Abacus Software, Inc. or the dealer is unable to deliver a replacement diskette(s) which is free of defects in materials or workmanship, you may terminate this Agreement by returning the program.

IN NO EVENT WILL ABACUS SOFTWARE, INC. BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE SUCH SOFTWARE EVEN IF ABACUS SOFTWARE, INC. OR ANY AUTHORIZED DEALER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY. Some states do not allow the limitation or exclusion of liability for incidental or consequential damages so the above limitation or exclusion may not apply to you.

This Agreement will be governed by the laws of the State of Michigan. Should you have any questions concerning this Agreement, you may contact Abacus Software, Inc., Customer Service, 5370 52nd Street SE, Grand Rapids, MI 49512.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT, AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

Amiga and AmigaDOS are registered trademarks of Commodore-Amiga Inc.

Second Printing, 1990
Printed in U.S.A.
Copyright ©1987, 1988, 1989, 1990

Abacus, Inc
5370 52nd Street SE
Grand Rapids, MI 49512

Data Becker GmbH
Merowingstr. 30
4000 Düesseldorf,
West Germany

ISBN 1-55755-026-X

Preface

At a time when C is all the rage, you have purchased an assembler. But don't be bothered by the self-assured talk of dyed-in-the-wool C fans. C is certainly a machine-oriented high-level language, but C is still a high-level language. Even though manufacturers of C compilers would like to tell you that C is as fast as assembly language, the fact cannot be denied that assembly language is absolutely the fastest way to program a computer.

True, when you a have to do a lot of floating-point calculations in a program, writing it in C doesn't look so bad, assuming its floating-point routines were written in assembler. But normally a C program is always noticeably slower than an assembly language program. The reason is that the C compiler uses only a small set of the machine language commands available. As an assembler programmer you have the ability to obtain full use of your processor.

You often hear that you couldn't write any large programs in assembly language, but this doesn't seem true anymore once you know that AssemPro Amiga was written completely in assembly language. And it certainly isn't small, is it?

The processors in the 68xxx family are very well suited to programming in assembler. They offer a relatively compact, yet powerful instruction set, so you don't have to remember too many commands, and a large number of addressing modes, which together with the fifteen CPU registers simplify programming considerably.

To further simplify working in assembly language, there are libraries on your assembler disk which allow you to call almost any operating system function with a single command.

After so many positive things, there are still a few things to be mentioned:

AssemPro Amiga is an assembler. In order to use it, you must know how to program in assembly language. If this is not the case, you should get an introductory book and learn this language. If you don't have any programming experience at all, then maybe you shouldn't start with assembly language. Try Pascal or Modula instead.

Just as AssemPro Amiga isn't designed to teach assembly language programming, it is not intended as a tutorial about the Amiga operating system either. If you don't already have books about the operating system, there are many available.

In conclusion I would like to thank Hannes Rügheimer and Uwe Braun who did some of the various work for me and also supported me in my efforts, as well as Jochen Schneider, Barbara Schütte and Gudrun Debus for extensive editing of this manual.

Peter Schulz

Table of Contents

1. **Introduction** 1

2. **The Editor** 13
   - The Editor window 14
   - Info line 15
   - The Editor menus 15
   - The File menu 16
   - The Edit menu 18
   - The Search menu 21
   - The Block menu 23
   - The keyboard 25
   - The shortcut commands 28

3. **The Assembler** 35
   - The assembler window 35
   - The info line 36
   - Format of a line 36
   - Labels and Variables 37
   - Arithmetic expressions 43
   - Number formats 43
   - Operators 44
   - Combined calculations 45
   - The effective addressing modes 45
   - The 68000 instructions 47
   - The Assembler Pseudo-ops 48
   - The Assembler menus 61
   - The File menu 61
   - The Assemble menu 63
   - The Output menu 65
   - Normal Code 68
   - PC-relative Code 68
   - Absolute Code 69
   - 68010 Code 69
   - Including assembly language programs in BASIC 69
   - Differences from other assemblers 70

4. **The Debugger** 71
   - The debugger window 72
   - The output display 72
   - Hex/ASCII dump 72
   - Disassembled output 73
   - Numeric Output 74
   - Display register 74

The Debugger menus 76
The Debugger menu 76
The Commands menu 77
The Parameter menu 80

5. Disassembler and Reassembler 83
The disassembler 83
The reassembler 84

6. The Tables 87
Effective addressing modes 87
Operating system calls 87
Construction of the table files 88

7 The Libraries 91
Operating system calls 91

Appendices 93
Appendix A Error Messages 93
Appendix B Directives and Pseudo-ops 99
Appendix C Editor Short-cuts 101

Index 103

1. Introduction

The complexity of operating systems for computers in the "new" generation (multi-tasking, more than 500K RAM) like the Commodore Amiga makes it harder to learn how to program these computers (regardless of the programming language used). This is because the operating system routines will have to be accessed for input and output of data, if not in other areas as well. This means that the Amiga programmer must become familiar with the window structure as well as the message system, which can overwhelm an Amiga beginner and budding assembler programmer.

AssemPro Amiga makes it possible to use 68000 machine language without this operating system overhead. That is, you can first learn machine language programming and then worry about more complex programs which make heavier use of the operating system. For these more complex programs a macro and constant library is included with AssemPro. Experienced Amiga programmers won't have to worry about the definition of operating system variables and/or macros.

Our very first program is directed at the inexperienced assembly language programmer. This will clarify the programming steps necessary, from entering the source text to testing an executable program with the integrated debugger in AssemPro. To keep the programming effort to a minimum, we will avoid screen output, which is why you can observe the operation of this program only with the debugger. If you have already had experience with the Amiga and are interested in "window handling" using machine language, take a look at the example program TEST_PRG.ASM in the DEMO subdirectory.

Our first program will sort twenty 16-bit numbers in memory (value range 0-65000). To keep the task from getting unnecessarily difficult, we will use the bubble sort algorithm, which is simple to program, but is not terribly fast.

The bubble-sort algorithm can be represented with the following structogram (with the assumption that the data to be sorted are in the array A[], the total number of data to be sorted is represented by the variable N and the array A[] to be sorted in ascending order):

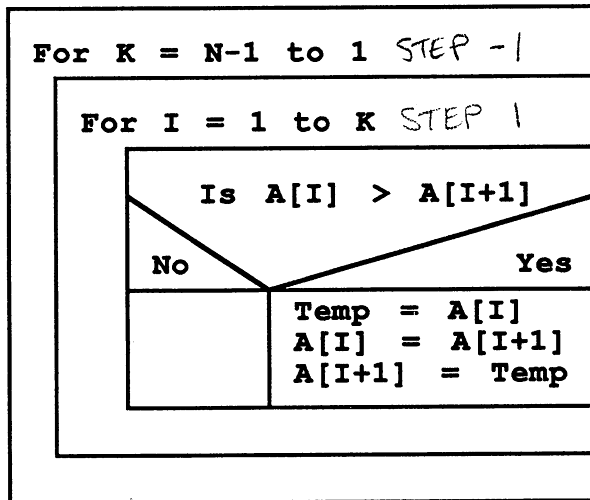


Figure 1 - Bubble-sort structogram

In each pass through the second program loop (exchange loop with the index variable I), the largest element in the remainder of the array moves to the end of the array through the exchange operations in the comparison portion of the algorithm (is A[I] > A[I+1]?). This makes it possible to decrement by one the maximum value of the loop variable for the exchange loop (I) after each pass through the main loop (for K = N-1 to 1). After the exchange loop has been called (N-1) times, the array contents are sorted in ascending order. If you want to sort in descending order, just change the condition in the exchange loop (is A[I] < A[I+1]?).

Our task is to translate this algorithm into the machine language for the 68000 processor. Although we will limit ourselves to the use of only seven assembler commands, you should keep all of the 68000 commands in mind when writing larger assembly language programs.

The large number of processor registers (eight 32-bit data and eight 32-bit address registers) make it easy to convert the problem into 68000 assembly code. We will arbitrarily choose the registers D2 and D3 to store the two loop variables I and K, and A2 to store the starting address of the data array. This data array consists simply of successive 16-bit numbers at the arbitrarily-named symbolic address DATAS.

Our program fragment looks like this:

```
START:        MOVE.L #DATAS,A2      ; address of the data
array
     MOVE.W #20,D2 ; number of elements N=20
     SUB.W  #2,D2  ; (N-1) and -1 because of DBRA

LOOP1:        MOVE.W #0,D4   ; I array access pointer
     MOVE.W D2,D3  ; K as upper loop bound
LOOP2:        MOVE.W D4,D5   ; times 2 bytes per array
element
     LSL.W  #1,D5  ; by shifting register left
     MOVE.W 0(A2,D5.W),D6 ; A[I] to D6
     MOVE.W 2(A2,D5.W),D7 ; A[I+1] to D7
     CMP.W  D7,D6  ; destination -COMPARE- source->
     BLE    CONT1  ; D6 ≤ D7? If so, no exchange
SWAPIT:       MOVE.W D6,2(A2,D5.W)  ; D6->A[I+1]
     MOVE.W D7,0(A2,D5.W)  ; D7->A[I]
CONT1:        ADD.W  #1,D4   ; increment the array pointer
     DBRA   D3,LOOP2       ; For I = 1 to K
     DBRA   D2,LOOP1       ; For K = N-1 to 1
     RTS
DATAS:        DC.W 100,96,6,10,46,86,90,16,20,70,26,36,50
     DC.W 56,76,60,80,40,60,66
     DC.B "END OF THE ARRAY"
     END
```

Now a little bit about the commands used:

Basically all 68000 assembly language commands have the structure:

```
COMMAND.LENGTH      SOURCE,DESTINATION
```

The MOVE command moves data in memory and in the various registers. The length specifications available are byte (8 bits, MOVE.B), word (16 bits, MOVE.W), and long word (32 bits, MOVE.L). MOVE.L D1,D2 moves all 32 bits of data source register D1 to data destination register D2 and leaves D1 unchanged. If you want to load absolute numbers into a register, precede the desired number with a number sign (#): MOVE.W #20,D2 loads the number 20 as a 16-bit number into data register D2, that is, of the 32 bits available in D2, only the "lower" 16 bits are used or modified. For the command MOVE.W 2(A2,D5.W),D7 the processor calculates the source address, that is, the address whose contents are loaded into register D7, in the following manner: add the lower 16 bits of the contents of register D5 to the contents of address register A2, then add an offset of 2 to this result.

The ADD command adds, as the name indicates, the contents of a source register to the destination register, while the SUB command

subtracts the contents of the source register from the destination register.

LSL.W #1,D5 shifts the contents of the register (the lower 16 bits) one bit position to the left, which corresponds to multiplying the contents of D5 by two. In the program this is done to adapt the array index to the array structure (an array element occupies two bytes).

```
CMP.W D7,D6
BLE   CONT1
```

These two commands are almost always used together in 68000 assembly language programs in this or a similar form. BLE CONT1 means "Branch if Less than or Equal" to the symbolic address CONT1. Several different comparison operations are permitted in the BRANCH command, such as BGT = Branch if Greater Than.

The command sequence:

```
CMP.W  D7,D6
BLE    CONT1
```

can be interpreted as follows:

destination register	<comparison operation>	source register =
(contents of D6)	<less than>	(contents of D7)

if true, then branch to symbolic address CONT1.

The DBRA command makes it very easy to construct loops. DBRA D3,LOOP2 decrements the D3 register (word) by one, tests to see if the contents of D3 are less than zero, and branches if this is not the case to symbolic address LOOP2.

RTS, finally, returns from a subroutine call.

After this brief introduction to assembly language, simply insert the AssemPro disk in the drive and open the disk icon with a double click. Now you can start AssemPro by double-clicking on its icon. After the program has loaded, several windows will open. You should activate the editor window by clicking in it. Now it is possible to enter program text; you must be sure that only label names (START, LOOP1, etc.) appear in the first column of the text and that assembler commands are indented at least one space from the start of the line. Enter the sort program, as listed on the previous page, into the editor window.

```
Assembler
Editor: SORT.ASM

START: MOVE.L  #DATAS,A2          ; address of the array datas
       MOVE.W  #20,D2             ; number of elements N=20
       SUB.W   #2,D2              ; (N-1) and -1 because of DBRA

LOOP1: MOVE.W  #0,D4              ; I array across pointer
       MOVE.W  D2,D3              ; K as upper loop pointer
LOOP2: MOVE.W  D4,D5              ; times 2 bytes per array element
       LSL.W   #1,D5              ; by shifting register left
       MOVE.W  0(A2,D5.W),D6      ; A[I] to D6
       MOVE.W  2(A2,D5.W),D7      ; A[I+1] to D7
       CMP.W   D7,D6              ; destination -COMPARE- source->
       BLE     CONTI              ; D6 < D7? If so, no exchange
SWAPIT:MOVE.W  D6,2(A2,D5.W)      ; D6->A[i+1]
       MOVE.W  D7,0(A2,D5.W)      ; D7->A[i]
CONTI: ADD.W   #1,D4              ; increment the array pointer
       DBRA    D3,LOOP2           ; For I = 1 to K
       DBRA    D2,LOOP1           ; For K = N-1 to 1
       RTS
DATAS: DC.W 100,96,6,10,46,86,90,16,20,70,26,36,50
       DC.W 56,76,60,80,40,60,66
```

Figure 2 - Program text in the editor window

Select **Save** from the **File** menu and enter the name SORT.ASM in the File: gadget to save the source text. Once you have entered the name, click the **OK** gadget or press the <Return> key. Next activate the Assembler window by clicking on it (you may have to click the Editor window into the background by clicking the back gadget). Select the **Output** menu to make sure that the **Normal code** option is checked and then select **Assemble...** from the **Assembler** menu. When the Assembling requester (figure 3) appears, click the **OK** gadget and the program is assembled. You should save the finished program (select Assembler window / **File** menu / **Save as**), under the name SORT1. PRG. Next switch to the Debugger window by clicking the back gadgets of the Assembler and Editor windows. Resize this window to cover most of the screen, but keep the Assembler title line visible.

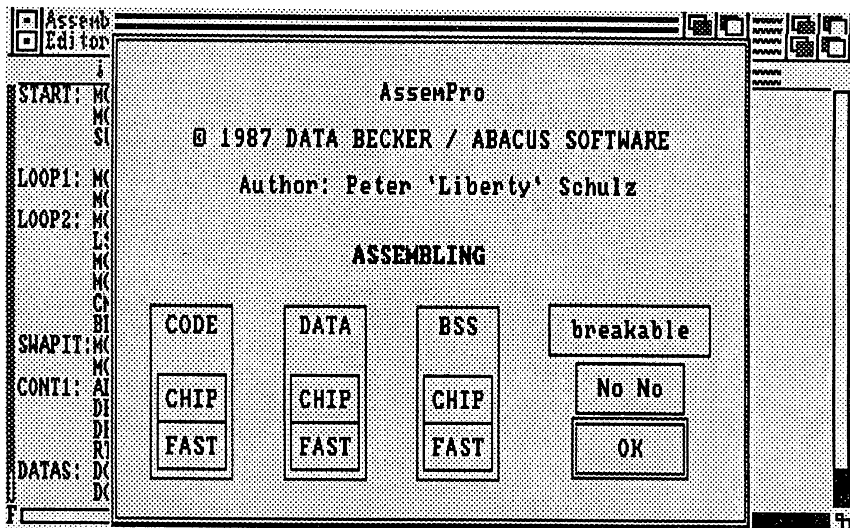


Figure 3 - Assembling requester

Select the **Debugger** menu then **Load** to load the SORT1.PRG program into the Debugger. When the Load Program: requester appears, use the down arrow to scroll through the list of files on the disk. Select the file named SORT1.PRG, it will appear in the **File:** gadget. Select the **OK** gadget to load the program. The program is displayed in the debugger window. Now select the **Parameter** menu and then the **Display** item sub-option **Symbolic.** Then select the menu item **Relocate symbols** from the **Debugger** menu. Now you can single-step through the loaded program and access program addresses using their symbolic names.

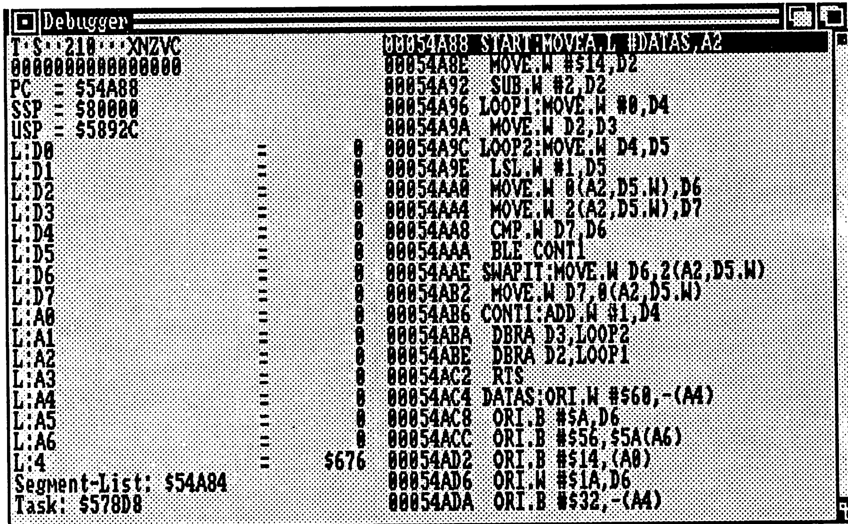


Figure 4 - The disassembled program

The address displayed in reverse video (orange) indicates the current state of the program counter. You now have several methods available for starting the loaded program:

1. Select **Next command** from the **Commands** menu.

2. Click the left mouse button on the command displayed in reverse video and hold it down. The GO icon appears. Move the mouse to a given address and when you release the mouse button the program is executed up to this address.

3. Select **Start** from the **Commands** menu after you have first placed a breakpoint at some program address.

Before we start the program with the procedure in point 3, let's first look at the data array to be sorted.

Select **Parameter-Display-From Address**, enter DATAS in the requester which appears and click on the **OK** gadget in the requester. The debugger will now show you the contents of the memory at the symbolic address DATAS. To display a hex dump of the memory contents, select **Parameter-Display-HEX-Dump** and you can check the contents of the data array to be sorted (See Figure 5).

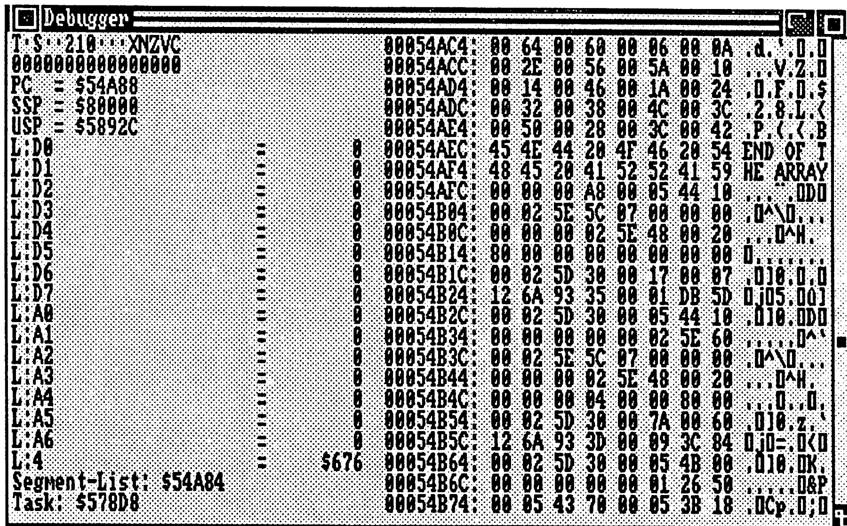


Figure 5 - Hex-dump of address DATA

The 68000 processor puts multi-byte data in the "correct" order in memory, first the high byte, then the low byte. In "our" data array the largest element (decimal 100) is in the first location and is represented as 00 64 hexadecimal.

Since we want to run our program, select **Parameter-Display-Symbolic** followed by **Parameter-Display-From Address** and enter START in the requester which appears. Position the mouse pointer at the address immediately before DBRA D2,LOOP1 and click the left mouse button. The address is displayed in reverse video. In the examples shown the address is $00054ABE (Figure 6), your address will be different. Next select **Command-Breakpoint-Set**, a breakpoint will be set at the selected address (Figure 7). If you have trouble setting this Breakpoint with the pulldown menu, click on this address again and while holding the right <Amiga> key press the <B> key.

```
Debugger
T:S··210···XNZVC                      00054A88 START:MOVEA.L #DATAS,A2
0000000000000000                      00054A8E  MOVE.W #$14,D2
PC  = $54A88                          00054A92  SUB.W #2,D2
SSP = $80000                          00054A96 LOOP1:MOVE.W #0,D4
USP = $5892C                          00054A9A  MOVE.W D2,D3
L:D0                  =          0    00054A9C LOOP2:MOVE.W D4,D5
L:D1                  =          0    00054A9E  LSL.W #1,D5
L:D2                  =          0    00054AA0  MOVE.W 0(A2,D5.W),D6
L:D3                  =          0    00054AA4  MOVE.W 2(A2,D5.W),D7
L:D4                  =          0    00054AA8  CMP.W D7,D6
L:D5                  =          0    00054AAA  BLE CONT1
L:D6                  =          0    00054AAE SWAPIT:MOVE.W D6,2(A2,D5.W)
L:D7                  =          0    00054AB2  MOVE.W D7,0(A2,D5.W)
L:A0                  =          0    00054AB6 CONT1:ADD.W #1,D4
L:A1                  =          0    00054ABA  DBRA D3,LOOP2
L:A2                  =          0    00054ABE  DBRA D2,LOOP1
L:A3                  =          0    00054AC2  RTS
L:A4                  =          0    00054AC4 DATAS:ORI.W #$68,-(A4)
L:A5                  =          0    00054AC8  ORI.B #$A,D6
L:A6                  =          0    00054ACC  ORI.B #$56,$5A(A6)
L:4                   =       $676    00054AD2  ORI.B #$14,(A0)
Segment-List: $54A84                  00054AD6  ORI.W #$1A,D6
Task: $578D8                          00054ADA  ORI.B #$32,-(A4)
```

Figure 6 - Selected address $00054ABE

```
Debugger
T:S··210···XNZVC                      00054A88 START:MOVEA.L #DATAS,A2
0000000000000000                      00054A8E  MOVE.W #$14,D2
PC  = $54A88                          00054A92  SUB.W #2,D2
SSP = $80000                          00054A96 LOOP1:MOVE.W #0,D4
USP = $5892C                          00054A9A  MOVE.W D2,D3
L:D0                  =          0    00054A9C LOOP2:MOVE.W D4,D5
L:D1                  =          0    00054A9E  LSL.W #1,D5
L:D2                  =          0    00054AA0  MOVE.W 0(A2,D5.W),D6
L:D3                  =          0    00054AA4  MOVE.W 2(A2,D5.W),D7
L:D4                  =          0    00054AA8  CMP.W D7,D6
L:D5                  =          0    00054AAA  BLE CONT1
L:D6                  =          0    00054AAE SWAPIT:MOVE.W D6,2(A2,D5.W)
L:D7                  =          0    00054AB2  MOVE.W D7,0(A2,D5.W)
L:A0                  =          0    00054AB6 CONT1:ADD.W #1,D4
L:A1                  =          0    00054ABA  DBRA D3,LOOP2
L:A2                  =          0    00054ABE  BREAKPOINT
L:A3                  =          0    00054AC2  RTS
L:A4                  =          0    00054AC4 DATAS:ORI.W #$68,-(A4)
L:A5                  =          0    00054AC8  ORI.B #$A,D6
L:A6                  =          0    00054ACC  ORI.B #$56,$5A(A6)
L:4                   =       $676    00054AD2  ORI.B #$14,(A0)
Segment-List: $54A84                  00054AD6  ORI.W #$1A,D6
Task: $578D8                          00054ADA  ORI.B #$32,-(A4)
```

Figure 7 - Breakpoint at address $00054ABE

Selecting **Command-Start** executes the program up to the breakpoint; the entire exchange loop is executed once and the largest element in the data array should have moved to the end of the array. To check this: select **Parameter-Display-From Address**, enter DATAS in the requester, then select **Parameter-Display-HEX-Dump**, and the data value 100 (hexadecimal 0064) will be at the end of the 20-value data array (Figure 8).

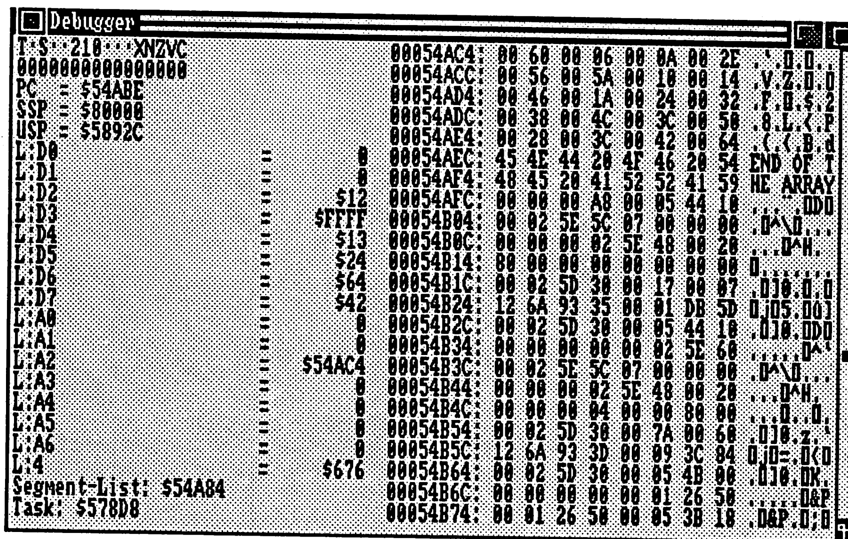


Figure 8 - Array after one loop

To run the remainder of the program the last breakpoint set must be cleared and a breakpoint must be set at the last command in the program. The process to do this will soon become second nature to you: select **Parameter-Display-From Address**, enter START in the requester, then select **Parameter-Display-Symbolic**, click the address of the set breakpoint, select **Command-Breakpoint-Erase**, click the address of RTS (which in our case is $0054AC2), select **Control-Breakpoint-Set** (Figure 9). If you have problems setting or erasing this Breakpoint use the right <Amiga> key <B> or right <Amiga> key <K>, respectively.

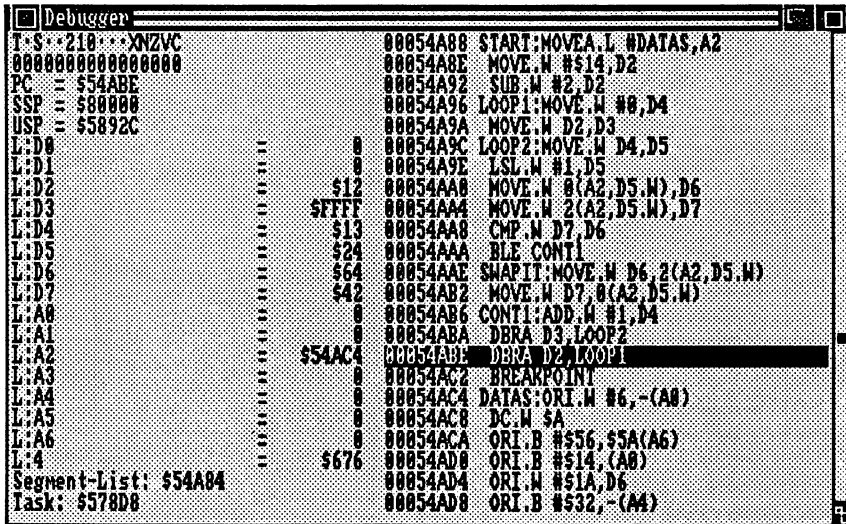


Figure 9 - Breakpoint at RTS

Start the remainder of the program by selecting **Command-Start**. After the entire program has been executed, the 20 data values in memory at the address DATAS should be sorted in ascending order, something we want to verify.

The following procedure is necessary:

1. Select **Parameter-Display-From Address**

2. Enter DATAS in the requester

3. Select **Parameter-Display-HEX-Dump**

As you can see for yourself, the twenty 16-bit numbers of the data array are sorted according to size in the array (Figure 10).

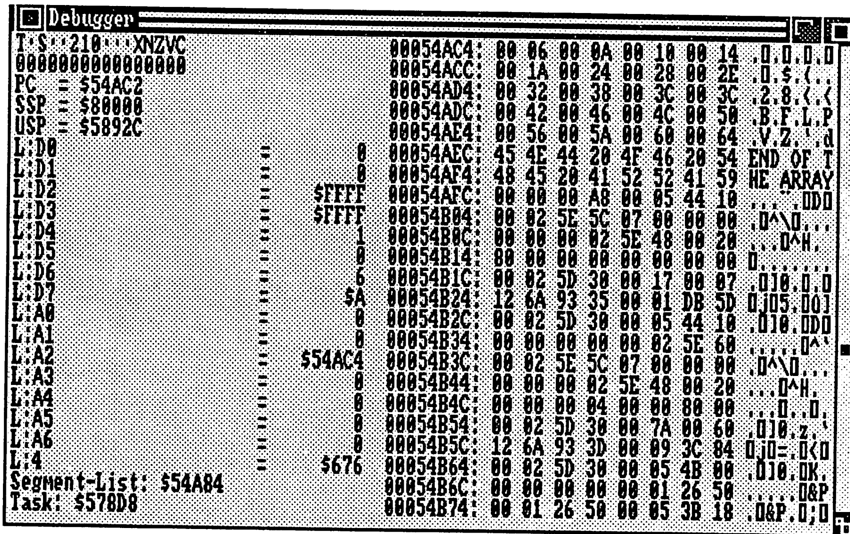


Figure 10 - Sorted array

Naturally you can load the program again and single-step through the program by selecting **Control-Next command**, allowing you to follow the loading of the processor registers with the various values.

2. The Editor

The Editor was developed especially for AssemPro Amiga. As a side note, AssemPro uses only a LF (CHR(10)), CR (CHR$(13), or CRLF (CHR$(13),CHR$(10)) as the end-of-line and null (CHR$(0)) is used as the end-of-file.

You can open as many windows as you want–the number is limited only by the available memory. Intuition will become slower as you open windows. You shouldn't overdo it.

Here we should mention a little warning: If you activate a window while Intuition is still busy redrawing other windows, the computer *may* hang up. This can only happen if you click too many windows too rapidly.

The maximum line width is 127 characters. In contrast to some editors which reserve the same amount of memory for each line, and are therefore line-oriented, AssemPro Amiga reserves only as much space for a line as it actually requires. If there are spaces (CHR$(32)) at the end of a line, they are removed, since it rarely makes sense to lengthen a program line in this manner.

Through this efficient memory management and the ability to change the size of the editor file buffer, a RAM disk can usually be kept in memory, even on small systems (512K).

The Editor window

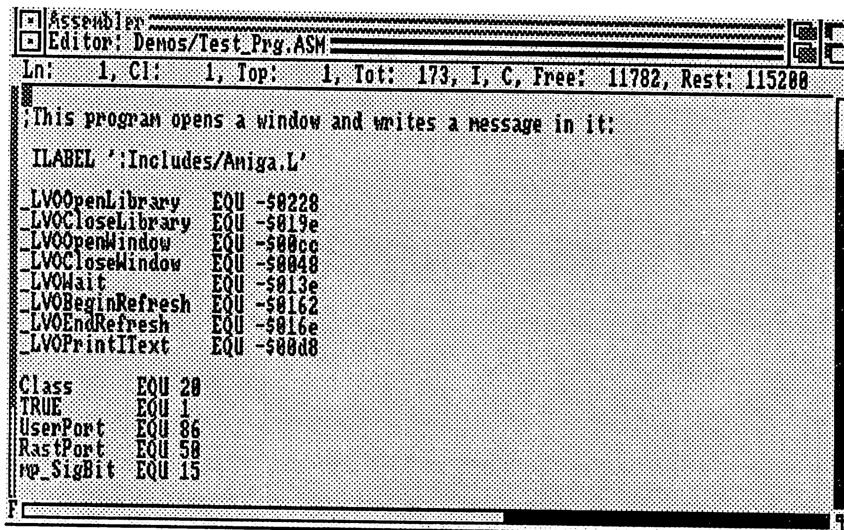


Figure 11 - The Editor Window

The Editor window contains the standard Amiga window gadgets and a few of its own. On the right side is a scroll bar with which you can move through the text. When you click on the scroll bar and move it, the window contents move along with it. The same applies to the horizontal scroll bar at the bottom of the window.

On the left side is a scroll bar which is disabled. You cannot move it, as indicated by the shaded pattern. This gadget tells you how much space in the editor buffer the text uses. The other scroll bars indicate the position of the window in relation to the entire text.

In the lower left corner of the window you find an "F". When you click on this the editor window will be expanded to maximum size and is brought to the foreground.

The name under which the current text will be saved when you select **Save** is always displayed in the title line. Directly under this you find the info line.

Info line

The info line contains the following information (Figure 11):

Ln:	Number of the line in which the text cursor is located.
Cl:	Number of the cursor column.
Top:	Number of the top line in the editor window.
Tot:	Total number of lines in your file.
I:	The editor is in insert mode.
O:	The editor is in overwrite mode.
C:	The keypad is switched to control keys (also see *The modes*).
N:	The keypad enters numbers.
Free:	Number of characters left in the editor buffer.
Rest:	Length of the largest free memory block.

When you click on the info line with the left mouse button, the tab positions are displayed instead of the info line. By clicking it again, the info line is displayed.

The Editor menus

This section presents a list of the editor window's menu titles and an explanation of each menu's item. Many menu items may also be selected from the keyboard by pressing the right <Amiga> key and the letter displayed on the right side of each menu item.

The File menu

New <Amiga><N>

Erases the contents of the editor buffer.

Open <Amiga><O>

To load a file, go to the **File** menu and select **Open**. All of the files in the current directory are displayed in the **Open Text file** requester which appears. Select a file by clicking on it with the mouse. The filename will appear in the **File:** gadget. Select the **OK** gadget to load the file. The arrow keys are used to scroll through the directory listing. If there is no text in the current editor buffer, the file is simply loaded into the buffer. Otherwise a new editor window will be opened, into which the file will be loaded.

The following applies to all File requesters: The / gadget updates the file display. The **Drive** gadget displays the current devices. The **CD** gadget will clear the pathname from the **File:** gadget. Drawers are displayed with a (DIR) preceding them.

Save <Amiga><S>

A **Save text as:** requester will appear if you select **Save** or click on the close gadget of the Editor window but the file does not have a name yet (only "Editor:" appears in the title line).

To save a file again, simply select **Save**. If you want to close the current editor window at the same time, click on the close gadget of the window. If you have changed anything in the file it is saved in the current directory and then the window is closed. If this was your last (only) editor window, the file is erased from memory after it is saved, but the window remains open.

Save as

If you want to save the file under a different name than that in the title line, select **Save as**. You will again get a **Save text as:** requester in which you can enter the name of the file. When you click **OK**, the modified name will appear in the title line of the window.

Erase <Amiga><D>

A **Erase File:** requester appears that allows you to erase files from the disk.

Rename

A **Rename:** requester appears that allows you to rename files on the disk. Enter the filename to be changed in the **File:** string gadget and select **OK**. The **Rename:** requester reappears, enter the new name in the **File:** gadget and click **OK**. The file is renamed.

Compare

A **Compare File 1:** requester appears that allows you to compare files on the diskette. Enter the first filename to be compared in the **File:** gadget and select **OK**. The **Compare File 2:** requester will appear. Enter the second filename to be compared in the **File:** gadget and click **OK**. A requester will appear that allows you to move through the files by selecting the arrows. Selecting **Compare** will automatically compare the two files. Selecting **Enough** will end the comparison.

Directory

Create a new directory: requester appears which allows you to create a new directory on the disk.

Save pre-setting

The function key definitions and the checked menu options are saved to a disk file. When AssemPro is loaded, the check options are active and the function key definitions are loaded.

Icon

When **Icon** is checked, an icon will also be created when saving so that your file can also be seen in Workbench. When you open this file from the Workbench, it will first load AssemPro, which then loads this file.

Back-up

When this item is checked, a backup copy of files saved is created on disk. The filename is preceded with "Back-up of". An icon is not created.

Backup Copy

When this item is checked, a requester appears where you may enter the device that the backup copy is saved to.

Quit

To return to the Workbench, select this item.

The Edit menu

Set tab <Amiga><T>

Sets a tab at the current cursor position. If there is already one here, it is removed.

List <Amiga><L>

Earlier, when it was customary to enter source programs with line numbers, you could view a subroutine at line 1000, for example, by entering "LIST 1000". Since AssemPro doesn't need line numbers, we need a new way of viewing specific parts of the program. You can do this by specifying a label name. The editor will search from the start of the file for a line which starts with the string which you have entered.

For example, if you want to see a subroutine which you have called "SUB," enter "SUB" (without quotation marks) in the requester which appears when you select **List** and then confirm this input by pressing <Return> or by clicking on the **OK** gadget. You can also enter "SUB:" because a label is always followed by a colon. This is especially useful when you have defined two labels in the program which start with the same letters, such as "SUB" and "SUB1". This search function may find "SUB1" if you just enter "SUB" if it is defined first in the program.

You can also enter something different as the search criterion if you know that a certain line starts with a given string, such as "MOVE D0,SR". The correct number of spaces must be included.

If the first character of the search string is a number, your input is interpreted as a line number and the cursor is moved to this line. The first line of the file is numbered 1.

Undo <Amiga><Q>

Undoes (almost) all changes in the current line. AssemPro Amiga uses only as much space per line as that line actually requires. To be able to edit with reasonable speed it is therefore necessary to copy the current line into a buffer when the user is entering characters and leave it there until the cursor leaves the line again. This prevents having to move the entire source text every time a new character is inserted.

As long as the line is in the buffer, the original line is still in the text memory. When you select the **Undo** item from the **Edit** menu, the buffer line is deleted and the original one is visible again. This is very helpful if you make a mistake. But as soon as you leave the line, the buffer line will be written into the file in memory and **Undo** won't restore the line.

For technical reasons, the buffer is also written to the text file by the following commands: saving the file, search operations and block operations.

Don't throw your computer out of the window if the **Undo** item doesn't seem to work.

Write mode

There are two different writing modes. To switch from one to the other you select the appropriate submenu option from the **Write mode** item in the **Edit** menu.

In the insert mode, every character you type is inserted at the current cursor position and the rest of the line is moved to the right.

In the overwrite mode, the new character replaces the old character at the cursor position.

Keypad

Here you can set what function the numeric keypad will have: whether it works normally, or whether the keys perform control functions. Control functions are enabled by selecting **Control Keys** from the **Keypad** item in the **Edit** menu. The info line will also tell you which mode is enabled, with a C for control keys or N for normal.

Owners of an Amiga 500 or 2000 already have the corresponding legends on the keys. The functions which are available to you are explained in detail at the end of *"The keyboard "* section.

Function keys

Here you change the assignment of each function key. First click on the function key you want to change and then click on the **Get** gadget so that the old contents are displayed. After you have edited the contents, confirm your input with **OK**, or if you don't want to leave the requester, click on **Save**. If you click on **Cancel** instead, you will exit the requester and nothing is changed. Nothing is changed if you click another function key, either.

You enter the assignment for the function key in the top input line in the requester. You can enter up to 120 characters. The commands which you can use are found in Appendix C "*Editor Short-cuts*". The explanation entered in the lower line of the requester serves to explain to the user what your function key macro does. This explanation will appear in the info line if you press <Shift> and the corresponding function key and stays there until you click on the info line.

Your function key definition is saved on the disk when you select **Save pre-settings** from the **File** menu.

Text color

Here you can set the color for the text in the editor window. Simply click on the desired color in the submenu.

Paper color

Used to set the background color of the editor window. Simply click on the desired color in the submenu.

Memory

A requester is displayed that allows you to change the size of the editor buffer. You are then able to load a file which doesn't fit into the current buffer. Changing the editor buffer size deletes the original contents.

The Search menu

Search for? <Amiga><W>

In this requester you can determine what you want to search for and what you want to replace it with. This doesn't start the actual search or replace option—that's taken care of by the next three options.

When you select **Variable**, AssemPro will search for only variables. These are marked by a character before and after the name which is not a valid label character. A label always begins with a letter, "\", or "_". All subsequent characters must be letters, digits, "_", "@", or a character with a code from 160 to 255, which includes foreign characters.

For example, if you search for "OV" with the normal search function, it finds every "MOVE", because it contains "OV". If you had selected **Variable**, only the variable "OV" is displayed, no matter where it appears. Example: MOVE #OV,D0 OR DC.B 4*OV+34, etc.

It should be obvious what **ignore case** does. If it is not selected, then the case of a text occurrence must match that of the search criterion exactly. Otherwise, no distinction is made.

In addition, you can select whether all occurrences are replaced, or only the first one (**one**). You can enable a query before each replacement asking you whether it should be performed or not (**chosen ones**). You can also replace every search word without confirmation (**all**).

Replace

If this is checked, calling the search function will replace occurrences found according to the settings you made in the Search requester. If it is not, no replacements will be made.

Search backward <Amiga><->

Searches or replaces backward from the current cursor position to the start of the file.

Search forward <Amiga><+>

Searches or replaces forward from the current cursor position to the end of the file.

Error file

The error-search function

A feature of AssemPro Amiga is that the assembler can create an error file which can be used by the editor to display the errors in the source text. You can find syntax errors in your programs.

The assembler produces the error file, please refer to Section 3, *"The Assembler"*, for information on creating the error file. This section explains how to use such a file in the editor.

When the assembler has produced an error file and you get the message "xx Errors." after assembling, simply click on the editor window and select the **Load** item from the **Error File** item in the **Search** menu. The editor will then load the error file. If you use include files, the editor will show you the first included file in the info line. You only have to worry about included files when your program consists of multiple source files. You can then select the source file you want to correct first with **Next source file.**

When loading, the errors are sorted according to their position in the source file. When you select **Next Error**, the cursor jumps to the first error. The position of the error, the file or the macro in which it occurred, and the error message is displayed in the info line. If you want to see the normal info line, simply click on the info line with the mouse.

A special case occurs when the error occurs in a macro. Here the name of the macro is specified in the info line and the position that is displayed there refers to the macro, not the source file, because the assembler doesn't know where you defined the macro. You must either figure out the source of the error from the error message (usually incorrect parameter types are the problem), or write down the position and look for the macro definition to see where the error occurred.

When you select **Next error** again, the next error is displayed for you. This continues until you have seen all of the errors, in which case you see the message "The rest is error free!" in the info line. You can display the previous error at any time by selecting **Last Error.**

If you delete a line when correcting errors, remember to take this into account when viewing other errors, since this can change the position of the line so that the error is located at a different position.

After you have corrected all of the errors, you should remove the error file from memory again by selecting the **Delete** submenu item.

As long as you write only programs which consist of one module (one source file), you can skip ahead to the next section.

If your program consists of several modules (it is chained), you will have to take another point into account when searching for errors.

All of the errors, from all of the modules, are written in the same error file, so you will have to determine the correct include file. The included file is the name of the module you want to debug first. After loading the error file the name of the first erroneous included file (in alphabetical order) is displayed.

When you have corrected this module, select **Next Source Text** in the **Error File** item of the **Search** menu. In the info line, you will see either the name of the next module with errors, or if the previous module was the last one, the name of the first module again. If the name of another module is displayed, proceed as above until you have corrected all of the modules.

Naturally, you can correct the modules in a different order, but you have to make sure that you don't forget any.

Last error **<Amiga><R>**

Displays the previous error.

Next error **<Amiga><F>**

Displays the next error.

The Block menu

Blocks

Text blocks are defined by setting a block-start and a block-end. This can be done with the appropriate menu options, the short-cut commands, or the mouse. Place the mouse pointer at the start of the block. Press the left mouse button, hold it down, and move the mouse to the end of the block. The marked area is highlighted on the screen. When the mouse pointer is at the end of the block, you can release the left mouse button again. The block is now shown in a different color.

The block boundaries, by the way, are not limited to the start or ends of lines–they can occur anywhere in the line. A block can even consist of just one word, or even one character. If the block start or block end

comes after the last character on a line, the LF (end-of-line marker) also belongs to the block.

When you no longer need a block, you can remove the block markers, unmarking the block. A block is also unmarked when you delete it, mark a new block, enter something in the line containing the start or end of the block, or call the **Replace** item.

You can **Copy** a block, copying it into the block buffer, and then paste it at the current cursor position with **Insert**. This block buffer is accessible to all of the files which you are editing, so you can copy blocks between files. Every time you copy a block, the old block in the block buffer is erased. The block is not erased until you copy a block which consists of just one character or you exit AssemPro. You can tell whether there is a block in the block buffer or not by whether **Insert** in the **Block** menu is displayed in normal or ghost (light) type.

If you don't know whether you have defined a block or not, just look in the **Block** menu. If most of the items there are disabled (displayed in light type), then no block is defined.

Mark start **<Amiga><A>**

Sets the block-start to the current cursor position.

Mark end **<Amiga><E>**

Sets the end of the block to the current cursor position.

List **<Amiga><V>**

Sets the cursor to the start of the block. If it is already there, it will move to the end of the block.

Move **<Amiga><M>**

If a block is marked, it is moved to the current cursor position, that is, it is removed from its old position and inserted at the cursor position. The markers are also moved.

Copy **<Amiga><C>**

If a block is marked, it is copied to the current cursor position, that is, it is inserted here, but it will also remain at the old position. The markers will stay at the old position.

Delete **<Amiga><K>**

If a block is marked, it is permanently deleted and its markers removed.

Cut **<Amiga><X>**

If a block is marked, it is transferred into the buffer. The original contents of the block buffer are erased.

Insert **<Amiga><P>**

If there is a block in the buffer, it is inserted into the current cursor position.

Save

If a block is marked, it can be saved to the disk as a text file.

Load

When you **Load** a block from disk, it simply inserts text from the disk into the current file. No block is marked after the loading and an existing block will not be erased.

Hide **<Amiga><H>**

If a block is marked, its markers are removed.

The keyboard

The text cursor can either be placed directly at a location with the mouse, by placing the mouse pointer at this position and then briefly clicking the left mouse button, or it can be moved through the file using the cursor keys:

Key	Normal	SHIFT	CONTROL
left	1 character	end of word	start of line
right	1 character	start of word	start of line
up	1 line	label definition	start of the file
down	1 line	label definition	end of the file

The meaning of "label definition":

The cursor moves to the next/previous line in which a label is defined. Such a line has a letter, "_", or "\" in the first column. If you press the <Alt> key at the same time, the cursor moves to the next/previous line in which a global label is defined. This is a label which starts with a letter or "_".

The cursor can also be moved with the following keys:

<TAB>

Normal: Moves the cursor to the next tab position (to the right). If there are no more tab stops to the right of the cursor, and if a tab stop is set in the first column, the cursor is moved to the start of the next line. If this is not the case, nothing will happen.

Shift: Moves the cursor to the previous tab position to the left of the current cursor position.

<DEL>

Normal: Deletes the character under the cursor and moves the rest of the line one position to the left.

Shift: Deletes all of the characters under and to the right of the cursor up to the next end-of-word, or at most up to the end of the line. The end of a word is marked by a space.

Control: Deletes all characters under and to the right of the cursor up to the end of the line.

Alt: Deletes the entire line in which the cursor is located.

<BACKSPACE>

Normal: Deletes the character to the left of the cursor and moves the rest of the line one place to the left.

Shift: Deletes all of the characters to the left of the cursor up to the next word start, but at most up to the end of the line. The word start is defined by a space.

Control: Deletes everything to the left of the cursor up to the start of the line.

Alt: Deletes the entire line in which the cursor is located.

When the overwrite mode is enabled, the appropriate text is deleted, but the remaining text on the line is not moved to the left.

<HELP>

When you press <Help>, the table window is expanded to maximum size and brought to the front. If you place the cursor on the first character of a command before pressing this key, information about that command is displayed in the window. If the cursor is on a command which is not in the current table, the old contents are displayed.

When you have read the help text, click on the close gadget of the table window. For additional information, see *"The Tables."*

<ESC>

With the <ESC> key you can enter the short-cut commands, they can also be entered as function key definitions. Your input is written into a command buffer and stays there until it is erased or overwritten.

Normal: Deletes the command buffer and waits for a command sequence to be entered. After terminating with <Return> or <Enter>, this is executed. If you don't enter anything, nothing will happen.

Shift: The command buffer is displayed in the info line and you can change it. After confirmation, the command sequence is executed.

Control: Executes the command sequence in the command buffer.

<Function keys>

Normal: Executes the corresponding command sequence stored in the function key.

Shift: Displays the explanation of the function key in the info line.

Numeric keypad

When you use your numeric keypad as control keys, the following functions are available:

Key	Function
<0> (= Insert)	Inserts a space at the current cursor position.
<0>+<Shift>	Switches between insert and overwrite modes.
<0>+<Alt>	The cursor is at the start of the line, a blank line is inserted before the current line, otherwise after it.
<.> (= Delete)	Works like the normal <Delete> key.
<1> (= End)	Puts the cursor at the end of the line.
<1>+<Shift>	Moves the cursor to the bottom line in the editor window.
<1>+<Control>	Moves the cursor to the end of the file.
<2>	Moves the cursor one line down.
<3> (= PgDn)	Moves the cursor one screen page down.
<4>	Moves the cursor one character to the left.
<5>	No action.
<6>	Moves the cursor one character to the right.
<7> (= Home)	Moves the cursor to the start of the line.
<7>+<Shift>	Moves the cursor to the start of the first line in the editor window.
<7>+<Control>	Moves the cursor to the start of the file.
<8>	Moves the cursor up one line.
<9> (= PgUp)	Moves the cursor up one screen.

The shortcut commands

The short-cut commands allow you to access the editor functions without using the menus. This is of special interest to touch typists, because you won't have to take your hands off of the keyboard in order to perform a command. A command sequence is either called with the function keys, where the command is already defined, or with the <Esc> key.

As in a programming language, it is also possible to perform multiple commands in sequence, or more than once.

But first an overview of the commands. No distinction is made between uppercase and lowercase. Most of the short commands work just like the versions that you can select from the menus with the mouse. If there are differences, they will be noted.

Cursor control

CO	Cursor up one line
COA	Cursor to start of file
COB	Cursor to start of block
COF	Cursor to previous error
COS	Search/replace backward
COV	Cursor to last label definition
CU	Cursor one line down
CUB	Cursor to end of block
CUE	Cursor to end of file
CUF	Cursor to next error
CUS	Search/replace forward
CUV	Cursor to next label definition
CR	Cursor one character right
CRB	Cursor to end of block
CRE	Cursor to end of line
CRW	Cursor to start of next word
CL	Cursor one character left
CLA	Cursor to start of line
CLB	Cursor to start of block
CLW	Cursor to end of last word
C(col,line)	Set cursor to (col,line). Example: C(5,20)=column 5 in the 20th line.
CN	Cursor to the start of the next line
CV	Cursor to the start of the previous line

Block commands

BA	Set block-start to current cursor position
BC	Copy block to current cursor position
BE	Set end of block to the current cursor position
BH	Unmark block
BK	Delete block
BM	Move block to the current cursor position
BP	Copy block from the block buffer to the current cursor position
BV	Display the text where the block is located
BX	Copy block into the block buffer

Replace

E[?AVU]r"string1" "string2"

Replace string1 by string2. The following settings are optional:

?	Query before replacement
A	Replace all
V	Search for labels
U	Distinguish between upper and lowercase

If nothing is specified, no distinction is made between upper and lowercase, AssemPro will not search for labels, and only the next occurrence is replaced.

The order of the attributes is arbitrary. An attribute can be specified more than once, but this will have no effect.

r	Specifies whether to search forward or backward
=L	Search for last occurrence
=N	Search for next occurrence

Example:

EAVN "OV" 'OVERFLOW'

Replace all occurrences of the label OV by OVERFLOW, from the current cursor position to the end of the file.

EVAN 'OV' >OVERFLOW>

means the same thing.

EL |.W| ##

Replace the last (previous) '.W' with nothing, that is, delete it.

EA?Nx .Wx @@

Delete all '.W', but ask first.

Delete commands

LA	Delete everything from the current cursor position to the start of the line.
LE	Delete everything from the current cursor position up to and including the end of line.
LL	Delete a character to the left of the cursor (like <Backspace>).
LR	Delete a character under the cursor (like <Delete>).
LZ	Delete the current line.

Search commands

S[VU]r 'String'

Search string. The following attributes are allowed:

V	Search for labels
U	Distinguish between upper and lower case

If nothing is specified, character case is ignored and the program will not search for labels.

The attributes can be in any order. An attribute can be specified more than once, but this has no effect. You can also use the ? and A attributes, just as you can set these attributes in the **Search for?** requester, but they have no effect on the search.

r	Specifies whether to search forward or backward
=L	Search for last occurrence
=N	Search for next occurrence

Instead of the apostrophes you can use any character as the string delimiter, provided it does not occur in the search string.

Example:

SVN 'OV'

Search for the next occurrence of the label OV.

SUL #.W#

Search for the previous occurrence of .W and pay attention to case, that is, do not find '.w'.

SN !""!

Search for the next "".

Tab commands

TL	Delete the tab at the cursor position
TS	Set tab at the current cursor position

Miscellaneous commands

Z 'string'	Insert the string at the cursor position
ZN 'string'	Insert the string on a new line after the current line
ZV 'string'	Insert the string on a new line before the current line

Instead of the apostrophes you can use any character as the string delimiter, provided it does not occur in the string.

Generating command sequences

Two successive commands are always separated by a colon. Any number of commands can be combined with parentheses, e.g. >(LZ:LZ:LZ)<.

A command or parenthesized sequence of commands can be repeated by replacing a repetition count (maximum of five digits) in front of it. Thus >7CR<, for example, moves the cursor seven places to the right.

By placing a W in front of a command or sequence of commands, it will be executed until an error occurs or until you press both <Shift> keys simultaneously. >WCU< moves the cursor to the end of the file.

You can cancel the execution of any command sequence by pressing both <Shift> keys together.

Examples:

    COA:WEN'*";'

Go to the start of the file and replace all * with ;.

    COA:EAN'*";'

has the same effect.

    CLA:128(TL:CR)

Delete all tabs.

    CLA:16(TS:8CR)

Sets a tab every eight places.

3. The Assembler

The assembler is probably the heart of this program. It is a two-pass assembler, which means that in the first pass through the source text the assembler just calculates all of the label values and the actual program is not created until the second pass.

If you have not saved the created code on disk, you will get a warning message if you are about to erase the code, such as when you exit AssemPro. You will not get a warning message when you assemble a new program, however, even if you have not saved the old one. Experience has shown that a warning message, in this case, is just annoying.

The assembler sticks to the usual 68000 conventions for input format, instructions, and addressing modes, so anyone who has programmed in 68000 assembly language before won't have any difficulties working with AssemPro Amiga.

Here a few notes about the notation used in the syntax definition:

(A1|...|An) One of the expressions A1 to An can be evaluated.

(0|1) Means, for example, that either the value 0 or the value 1 must be used.

[...] The material in square brackets is optional.

{...}* The material in curly braces can be repeated as often as desired, including zero times (meaning it can be omitted).

The assembler window

The assembler window can be moved and resized as usual. It is only used by the assembler to show what is currently being assembled (if **File name** under **Output** is checked) and after the assembly to output the errors which occurred (if you create an error file).

The info line

The following information is displayed in the info line:

CODE: Length of the CODE segment of the last-assembled program.

DATA: Length of the DATA segment of the last-assembled program.

BSS: Length of the BSS segment of the last-assembled program.

Free: Number of bytes of the assembler memory which are still free. All of the defined labels and macros as well as the linked libraries are stored in the assembler memory. Since your program will continually increase in size during development, you should always check to see if there is enough room for the next change after assembly. You can avoid the "Not enough memory" error message by increasing the assembler memory at the appropriate time.

Rest: Length of the largest contiguous memory block which the operating system has available.

All specifications are in decimal. If you don't know what CODE, DATA, and BSS mean, they are explained in the context of the appropriate commands.

Format of a line

A line has the following basic format:

[label:][instruction[source operand[,destination operand]]][;comment]

The elements in square brackets are optional. The label must be in the first column. Only one instruction is allowed per line and it may not take more than one line. An example of such a line would be:

```
LABEL_1:MOVE.L D0,A0;here is the comment
```

This is not very readable, however, so you should separate various elements from each other on a line with spaces:

```
LABEL_1:    MOVE.L    D0    ,    A0    ;    here is the
comment
```

This isn't great either, but is does show you where you may insert spaces and where you may not. You may not insert spaces within elements, for example:

```
LABEL_1 :M OVE .L D 0,A    0 ;here is the comment
```

What you may do is replace a colon after a label with one or more spaces:

```
LABEL_1    MOVE.L D0,A0 ;here is the comment
```

A special case occurs when a line is composed of just a label and a comment. Then you can also write:

```
LABEL_2;here is the comment
```

The colon can be omitted.

If you have not defined a label, at least one space must appear before the pseudo-op or the assembler thinks the command is a label (a · is used to indicate a space):

```
·MOVE.L D0,A0
```

The length of a line is limited to 128 characters, but this isn't really a limitation. The end-of-line marker is included in these 128 characters. Since a line can be terminated by an LF, a CR, or a CR/LF, the actual line length is 127 or 126 characters (the built-in editor terminates lines with an LF, so your lines may not be longer than 127 characters).

Labels and Variables

Labels or variable names begin with a letter, an underscore (_), or a backslash (\). Subsequent characters can be letters, digits, underscore, @, or characters whose ASCII values are between 160 and 255:

```
variable::=('A'..'Z'|'_'|'\'){Sc}*

Sc::=('A'..'Z'|'0'..'9'|'_'|'@'|CHR$(160)..CHR$(255))
```

The length of a variable name is at least one character and a maximum of 126 characters. The upper limit is the maximum line length minus one (for the colon).

The variable name is stored in its entire length, and not, as is customary, just the first eight characters. This way you can use highly descriptive names for your subroutines, such as 'CALCULATE_SINE_OF_D1', without worrying that the assembler

will confuse this with 'CALCULATE_SINE_OF_D0'. On the other hand, short variable names take up less memory.

You probably already know what labels are. Labels are used to designate memory locations, such as branch destinations, addresses and data. When you want to branch to a subroutine or access an address, you can use the name of the label to designate this location. For example:

```
BSR SUB_1
...
SUB_1; subroutine 1
or
·MOVE X_POSITION,D0
...
X_POSITION: DC.W 0
```

Thus labels represent addresses.

Constants are different. The value of a constant must be specified explicitly. The equals sign, or, as is more common on assemblers, the EQU pseudo-op, is used for this:

```
CONSTANT=13
```

Alternatively you can also write:

```
CONSTANT:=13
```

or

```
CONSTANT = 13
```

or

```
CONSTANT EQU 13
```

You can insert any number spaces wherever a single space may occur.

Constants are used for values which are used often and have a specific meaning. For example:

```
CR=13 for carriage return
LF=10 for linefeed
```

For example, you can define a constant for instructions containing an index whose values can change, such as in a data structure where the fifth byte contains a counter, instead of accessing >5(An)<:

```
COUNTER=5
```

and then access >COUNTER(An)<, where An points to the base of the data structure. You should always define addresses which lie outside the program, such as operating system variables:

```
AbsExecBase = 4
```

at the beginning of the program, because the program assumes during pass 1 that any unknown labels are addresses, and this can lead to unnecessary error messages.

When it is necessary to define a constant at a later time, you should put a percent sign (%) immediately after the constant name, just to be sure (also see the section *"Combined calculations"*).

AssemPro supports a special kind of variable called a "replacement symbol." A replacement symbol is assigned a string by the EQUR pseudo-op:

```
SYMBOL EQUR 12(A0,D0.L)
```

When you write the following:

```
·MOVE.L SYMBOL,D1
```

the assembler translates it to:

```
·MOVE.L 12(A0,D0.L),D1
```

This allows you to assign a name to complicated addressing modes or other strings that you don't want to retype. You can even assign an entire command to a replacement symbol:

```
SAVE_REGISTERS EQUR MOVEM.L D0-A6,-(SP)
```

When you enter:

```
·SAVE_REGISTERS
```

the assembler translates it to:

```
·MOVEM.L D0-A6,-(SP)
```

This gives you sort of a "mini-macro." The only problem is that it costs some time.

Symbols in the first column are not replaced; thus the assembler will not replace LABEL1 by Hello in the following line:

```
LABEL1 EQUR Hello
```

since this would lead to an error message. If you want to replace a symbol which starts in the first column, just put the $ character immediately in front of this symbol. Example:

```
DEFINE:MACRO \$1,\$2 ; these $'s mean that \1 is a symbol

$\1=$\2

•ENDM
```

The expression:

```
•DEFINE CONST,7
```

would be transformed into:

```
•CONST=7
```

Admittedly, this doesn't make much sense in this special example, but there are more complex applications (such as converting C structures to assembler format) where this can be used.

Since a replacement can occur almost anywhere in the source text, the assembler must always scan the entire line for replacement symbols, which can take a perceptible amount of time in long programs. But if you don't define any replacement symbols, or at least only local ones (explained shortly), it won't take up any extra time (local variables take up time only when they are accessible).

The last variable type is the macro. How macros are defined are explained together with the corresponding command.

The assembler makes a distinction between global and local variables and redefinable variables.

Global variables begin with a letter or an underscore (_). They can be defined only once and can be accessed at any time.

Local variables begin with a backslash (\). The scope of a local variable is limited. You can access a local variable when there are no global variables defined between its definition and access to it.

Example:

```
GLOBAL_1:BSR.S \LOCAL
;Here you can access \LOCAL
...
•RTS
\LOCAL:
```

```
·BNE.S \LOCAL
...;here too you can access \LOCAL
·RTS
GLOBAL_2:BSR.S \LOCAL
;Here you will get an error message because the
definition
;of GLOBAL_2 prevents access to \LOCAL
...
GLOBAL_3:
...
\LOCAL
;Here you can use the same name again without any
problems
...
·BRA \LOCAL
\LOCAL=2
;Here you get an error message because \LOCAL is already
 ;defined
```

Important: Global variables may not be defined where you see "...".

Redefinable global variables do not affect the scope of local variables if they have been already defined. Thus the following program fragment is correct:

```
REDEFINE@ = 0
GLOBAL1:
\LOCAL1 = 27
REDEFINE@ = \LOCAL1
\LOCAL2 = 1 << \LOCAL1
GLOBAL2:
```

The operation of local variables is somewhat similar to Pascal. For example, you can write the following:

```
DELAY_BY_D0: MACRO ;Delay loop
\LP: DBRA D0,\LP
·ENDM
...
·MOVEQ #64,D0
·DELAY_BY_D0
·MOVEQ #19,D0
·LEA SOURCE,A0
·LEA DEST,A1
\LP: MOVE.B (A0)+,(A1)+ ;Transfer 20 bytes
·DBRA D0,\LP
·MOVE.W #$7000,D0
·DELAY_BY_D0
```

Here the local variable \LP is defined three times in a row without any access conflicts occurring. This is because variables which are defined

in a macro are valid only within the macro. As soon as the macro is terminated with ENDM, its local variables can no longer be accessed. This also holds for nested macros:

```
M1:MACRO
\LP:DBRA D0,\LP
·M2
·ENDM
M2:MACRO
\LP:DBRA D1,\LP
·ENDM
...
·M1
```

Local variables which were defined before are defined in a macro, provided that no local variables with the same names are defined in the macro:

```
M1:MACRO
·MOVEQ #\VALUE,D0
\LP:DBRA D0,\LP
·ENDM
...
\VALUE=20
·M1
```

We can explain how it works internally. If you already understand how it works, you're better off not reading this.

Every local variable is assigned a priority. When you start the assembly process, the priority level is set to one. This priority level is increased by one every time a macro is encountered in the program or an external source file is included (INCLUDE), and it is decreased by one when a macro or external file ends (END, ENDM). You can access only those local variables whose priority is less than or equal to the current priority level. If several labels have the same name, the one with the highest priority is used.

You can also define local variables between the start of the program and the first global variable, as well as between the last global variable and the end of the program.

The variables are not dependent on the division of your program into CODE, DATA, and BSS segments, only on the order in which they are defined in the file.

Note that when an external file is inserted with INCLUDE, it is treated exactly as if you had called a macro (the priority level is incremented by one). Thus you can restrict yourself to using local variables in the

external source file and you won't have to worry about conflicts with the program in which it is used.

Redefinable variables may be defined multiple times. A redefinable variable ends with the "@" character.

Redefinable variables can be redefined as often as desired. They retain the value that was last assigned to them. One example for using a redefinable variable is in the REPEAT pseudo-op.

Here are few restrictions you must keep in mind:

- Macro names must be global and cannot be redefined.
- Replacement symbols can be local, but cannot be redefined.
- Only constants and address labels can be local and/or can be redefined.

Arithmetic expressions

Number formats

The following number formats are supported by AssemPro Amiga:

- Decimal
- Hexadecimal, indicated by a $ before the number (max. 8 digits)
- Binary, indicated by a % in front of the number (max. 32 digits)
- String, enclosed in quotation marks (maximum of 4 characters)

Both upper and lowercase may be used in hex numbers. All numbers are properly sign-extended to 32 bits. An error message is given if a number larger than $FFFFFFFF (4294967295) or less than -80000000 (-2147483648) is used. AssemPro stores positive numbers as 32-bit unsigned numbers and negative numbers as signed 32-bit numbers. This may seem somewhat confusing, but from experience, it works very well.

Examples (the corresponding decimal number is in parentheses):

Decimal:	12, 1029, 65536, 3465123624
Hex:	$A (10), $FFFF (65535), $10000000 (268435456), $AFFE (45054)
Binary:	%101 (5), %11111111111111111111111111111111 (-1), %1001001100111 (4711)
String:	'AFFE' (1095124549), " =)" (540876841), 'a' (97), 'ST' (21332)

Operators

The following operators are available for computation:

<table>
<tr><td>+, -, *, /</td><td>The four basic operators</td></tr>
<tr><td>a<<b</td><td>Shift a (binary) b bits to the left. This corresponds to a*2^b.</td></tr>
<tr><td>a>>b</td><td>Shift a (binary) b bits to the right <=> ABS(a/2^b).</td></tr>
<tr><td>?a</td><td>The highest-order bit of a. This corresponds to the base 2 logarithm of a.</td></tr>
<tr><td>&</td><td>Logical AND</td></tr>
<tr><td>|</td><td>Logical OR</td></tr>
<tr><td>^</td><td>Logical exclusive OR (XOR)</td></tr>
<tr><td>-</td><td>Unary minus (2's complement; sign)</td></tr>
<tr><td>~</td><td>Logical not (1's complement)</td></tr>
</table>

All calculations are performed using 32-bit arithmetic. The priority of the operators is as follows:

<table>
<tr><td>Highest priority:</td><td>- (sign)</td></tr>
<tr><td></td><td>* or /</td></tr>
<tr><td></td><td>+ or -</td></tr>
<tr><td></td><td><<</td></tr>
<tr><td></td><td>>></td></tr>
<tr><td></td><td>~</td></tr>
<tr><td></td><td>&</td></tr>
<tr><td>Lowest priority:</td><td>| or ^</td></tr>
</table>

Combined calculations

Arithmetic expressions can be formed with the operators listed above. They can also be parenthesized, of course, up to 32 levels. The memory for storing calculations is set up such that you should never have problems with an expression that is too long to be evaluated. The expressions are put on a stack as usual and evaluated as soon as possible. The stack is set up for 20 operands, meaning that a maximum of 19 calculations may be open before an error message will result.

Labels followed by a percent sign (%) are treated as constants. This is important for certain addressing modes.

Any number of spaces may appear before and after the operators.

Examples:

```
1<<3 = 8
1024>>5 = 32
?8 = 3
?1024 = 10
?32 = 5
%1011<<3 = %1011000
%1010>>3 = %1
?%1010>>3 = 3
?(1<<a) = a
1 + 2 * 3
4711 * (3 - (123 | %1101) ^ $F) / 2
'aBcD' &$ff00ff00 - 1234
LABEL_1 - CONSTANT_2
(-0) * 32 / (((17 + 4) * 3) & %11110000
```

The effective addressing modes

An effective addressing mode represents the way in which a memory location or register is accessed. The following groups can be made:

Register direct:	Dn, An, CCR, SR, USP
Memory indirect:	(An), (An)+, -(An), d(An), d(An,Rm.x)
Memory direct:	a16, a32
Program counter relative:	d(PC), d(PC,Rm.x)
Immediate:	#k

where:

Dn	is a data register
An	is an address register
Rm	is either a data or an address register
x	is either W (word operand) or L (longword operand)
n, m	are integers between 0 and 7
a16, a32, d, k	are arithmetic expressions

For the PC-relative addressing modes d(PC) and d(PC,Rm.x) there are two different ways of using the arithmetic expression d:

If no address label occurs in d, it is used directly as an index, that is, the memory location which is d bytes from the current PC (for d(PC,Rm.x) the contents of the register Rm must also be added) are accessed.

But when at least one address label occurs in the expression d, the address of this label is viewed as an access address and the PC is subtracted before the index is generated (this also holds for d(PC,Rm.x)).

Examples:

```
·MOVE.W 2(PC),D0
; loads the opcode of the instruction which follows this
MOVE        instruction into register D0
·MOVE.L LABEL+2(PC),D0
; loads the contents of long word at address LABEL+2 into
D0
·MOVE.B LABEL(PC,D1.L),D0
; loads into D0 the contents of the memory location which
is D1       bytes beyond LABEL
```

To repeat, the determining factors is whether an address label occurs in the expression or not, not just any ordinary variable like a constant.

The following alternate notations are allowed in the effective address:

Instead of	Alternative
A7	SP
0(An,Rm.x)	(An,Rm.x)
d(An,Rm.x)	d(An,Rm)

If you want to get an overview of the addressing modes allowed for a certain instruction, you can do this by pressing the <Help> key when the cursor is directly over the first letter of the instruction in question. More information is found in Section 6, *"The Tables"*.

The 68000 instructions

AssemPro Amiga supports all of the 68000 instructions. The EA table mentioned in the previous section indicates which operand sizes can be specified for a particular instruction.

In general, all instructions where the operand size is fixed by hardware do not require an explicit operand size, such as the BCD or MOVEQ instructions.

If you omit the operand size of other instructions, then word (.W) size operand is generally assumed. Only with the EXT instruction and some assembler directives are you required to specify an operand size to prevent mistakes.

AssemPro Amiga adheres to the Motorola standard, so there should be no problems.

In addition, AssemPro allows several alternatives for some of the standard Motorola instructions:

Instead of	alternative
ADDA, ADDI	ADD
SUBA, SUBI	SUB
CMPA, CMPI	CMP
ANDI	AND
EORI, EOR	XORI, XOR
EOR	XOR
ORI	OR
MOVEA	MOVE

In cases for which you must specify for the Bcc, Scc, and DBcc instructions, you can use HS (Higher or Same) instead of CC and LO (LOwer) instead of CS.

AssemPro also allows you to translate 68010 assembler instructions if you have one installed in your Amiga. If the corresponding menu option is selected (checked), then you can use the four additional 68010 instructions, but only if you have a 68010.

The Assembler Pseudo-ops

CODE, DATA, and BSS

Syntax: `·Segment`

Segment can be either CODE, DATA, or BSS. You can use this to divide your program into three segments:

The actual program is found in the CODE segment.

The initialized data are stored in the DATA segment. These are data which are already defined when the program starts, such as strings which are printed.

In the BSS segment (Block Storage Segment) space is simply reserved for data which is placed there by the program and which does not have to be available at the start of the program. If you create absolute or PC-relative code, you cannot define a BSS segment!

Although the division between CODE and DATA segments is a matter of taste, the BSS segment is something different. It is not actually stored along with everything else and thus does not take up space on the disk.

To switch to the appropriate segment you just need to write the name of the segment, CODE, DATA, or BSS, as a directive on a line and all of the instructions following it are assembled to this segment.

The order in which the segments are located in memory is not set. As we mentioned, this division has no effect on the access range of local variables, so you can define the following macro, for example:

```
PRINT:MACRO \$MESSAGE
·PEA \ADDR
·BSR PRINT_TEXT ;outputs a string
·ADDQ.L #4,SP
·DATA
\ADR:DC.B \MESSAGE,0
·ALIGN
·TEXT
·ENDM
```

which you then call as follows:

```
PRINT 'This is a string'
```

You then have to write the subroutine which outputs a string terminated by a zero byte whose address is on the stack.

Instead of CODE you can also use TEXT.

DC (Define Constant)

To define data in a program, use the DC pseudo-op.

Syntax: `·DC.x expression{,expression}`

where x is either B (byte), W (word), or L (long word). expression is an arithmetic expression. The data are then stored in memory as a byte, word, or longword. This pseudo-op is similar to the POKE command which you may recognize from BASIC, except you specify more than one data value. When you define data with DC.B you can also use a string of any length, enclosed in either apostrophes or quotation marks, instead of an arithmetic expression.

Examples:

```
·DC.B 'This string is constant'
·DC.L 0,0,7
·DC.W 4711,'ST',0
·DC.B 0,6,"AMIGA",0
```

The operand size is required for the DC pseudo-op. You cannot write DC but must write DC.W. However, there are some alternatives:

Instead of	alternative
DC.B	DEFB, DEFM
DC.W	DEFW
DC.L	DEFL

DS (Define Space)

Syntax: `·DS.x number[,fill_value]`

x may be either B, W, or L and **number** and **fill_value** are arithmetic expressions. With this pseudo-op you define an area of memory whose length depends on the operand size. Either **number** bytes, words, or longwords are reserved. In the CODE and DATA segments this area is filled with the **fill_value**, but the **fill_value** is ignored in the BSS segment and the space is only reserved. If **fill_value** is omitted, the area is filled with zeros. Instead of:

```
·DS.x A,B
```

you can also write A times:

```
·DC.x B
```

Examples:

```
·DS.B 20
; defines 20 bytes
·DS.W 30,-1
; defines 30 words (=60 bytes) and fills these with -1
·DS.L 40,$1234
; defines 40 long words (=160 bytes) and fills these with
; $1234
```

The operand size is mandatory for this pseudo-op as well. You can also write DS as BLK and DS.B as an alternative to DEFS.

EQU

Syntax: `name=val` or `name EQU val`

name is a variable name and **value** is an arithmetic expression. The constant variable **name** is assigned a value of **val.**

EQUR

Syntax: `name EQUR string`

Defines a replacement variable. See the section on variables and labels for more information.

For defining constants you should always use the equals sign and not the EQUR instruction. Although this does not affect the function of the program, it does affect the speed of the assembler.

ORG

Syntax: `.ORG address`

address is an arithmetic expression. If you are writing a program in absolute code (a program where you know ahead of time where it is placed in memory), you can use this to set the address at which the instructions following it are assembled. The program can then be executed at this address.

This directive has no effect on where the assembler puts the program while it assembles it. Also, you normally cannot test an absolute program with the debugger.

Normally the ORG pseudo-op is used only once, at the beginning of the program. If you divide your program into segments, you cannot use ORG in the DATA segment because the segments are always packed one after the other and the start addresses of these two segments is set by this. Note that a BSS segment in an absolute program is not allowed. If you don't use the segmentation, you can use other ORG directives in your program. This can be useful if your program is burned into an EPROM chip, for example.

Example:

```
.ORG $F80000
...
The following program will run at address $F80000
```

END

Syntax: `.END`

You must always terminate your source text with an END so that the assembler knows that nothing more follows. If you forget the END, you will get an error message. It is important to conclude each source file with an END if your program consists of multiple source files or modules. When you bind in a source file with INCLUDE, the assembler needs the END to return to the source file which contained the END.

Example:

```
;Here is your program
...
.END
;Everything after this point will be ignored by the
assembler
```

ALIGN

Syntax: `.ALIGN.x`

In this case, x may be either W or L. Words and longwords on the 68000 can be accessed only on even addresses, so your program has to make sure that memory locations which contain word or longword data begin at an even address. To make sure that the next instruction is placed at such an address, use the ALIGN.W pseudo-op in the line before the instruction.

To make sure that an instruction is assembled to an address which is divisible by four, use the ALIGN.L pseudo-op in the line before the instruction.

The ALIGN directive is most often used before a DC.B pseudo-op to make sure that the following instructions are located at even addresses.

INCLUDE

Syntax: `.INCLUDE file`

file is the name of a file in the current directory which is inserted at the current point in the source file. You can also enter a complete pathname to specify a file outside of the current directory:

```
.INCLUDE ":Includes/Amiga.ASM"
```

The file or pathname can optionally be enclosed in apostrophes or quotation marks. If the file is not in the current directory or the directory specified, the assembler will first search in the global directory before giving an error message.

The file must contain source code terminated by an END statement. Assembly will branch to this file and continue with the original source file after the END.

You can, for instance, pack all of your subroutines into one file and your main program into another. Then you just have to put a corresponding INCLUDE pseudo-op somewhere in your main program, such as at the end, in order to combine the two source files.

It also makes sense to divide your program into multiple source files when it exceeds a certain size. This way you don't have to load and edit the entire program when an error occurs.

If you nest multiple INCLUDEs inside of each other (you INCLUDE a source file which INCLUDEs another, which ...), you should be aware that a maximum of only 30 nestings is allowed. You will have to decide for yourself what size you want to make your sections.

IBYTES

Syntax: `.IBYTES file[,length]`

file is again a filename, which can optionally be enclosed in apostrophes or quotation marks, and **length** is an arithmetic expression. IBYTES allows you to bind in (include) data from a disk file. The file must be in the current directory. If it is not there, the assembler will first search the global directory for it before reporting an error.

The data are inserted into the program at the location where the IBYTES is found in the program, in the manner in which they are stored on the disk. If you don't want to insert all of the data, you can specify an option length argument which specifies the number of bytes to insert.

For example, if you have a picture which you created with a drawing program and you want to insert it into your program so that you can display it, you just have to write the following:

```
.IBYTES PICT.PIC
```

or

```
.IBYTES PICT.PIC,32000
```

if your picture is 32000 bytes long.

An example of the IBYTES pseudo-op is found in the demo program DET_TEST.ASM. If you want to include program fragments, such as subroutines in this manner, you must assemble them in PC-relative mode.

SLABEL

Syntax:

```
·SLABEL file
·Name_1
...
·Name_n
·ENDS
```

You can specify as many variable names between SLABEL and ENDS as you want. At least one space must appear before each name and only one name is allowed per line. Comments can appear after the name, separated from it by a semicolon. You can also have lines which consist only of a comment.

file is again a filename, which can optionally be enclosed in either apostrophes or quotation marks. The file is written in the current directory.

You can use this directive to create a library that contains the variables that a library of functions needs. This library can then by included in any program with ILABEL.

You can save address labels, constants, replacement symbols, and macros, but they must be global variables. The advantage of such a library is that it can be bound in less time than an INCLUDE file.

Example:

```
M1:MACRO
....
·ENDM
K1=10
S1 EQUR D0-A6
L1:
·SLABEL LIBRARY.L
·Label:
·L1
;Macros
;Constants:
·K1
;Symbols
·S1
·ENDS
```

Another example for the SLABEL is found in the file, "Amiga_SLABEL.ASM", in the **Includes** directory. The mini-library, "Amiga.L", was created with this file.

ILABEL

Syntax: `.ILABEL file`

file is again a filename optionally enclosed in apostrophes or quotation marks. The file must contain variables created by the SLABEL pseudo-op. If the file is not in the current directory, the assembler will search in the global directory before reporting an error.

You can use this pseudo-op to include libraries. For example:

```
.ILABEL LIBRARY.L1
```

A special case occurs when you have saved address labels. When these are bound in with ILABEL the address of the ILABEL pseudo-op is added to the value of each address label. For example, if you have saved an address label which is located at address 20 and the ILABEL directive is at address 148 in your program, then this label is assigned the value 168. This, together with the IBYTES directive, makes it possible to insert already-assembled programs into your programs.

Examples of the ILABEL pseudo-op are found in most of the programs in the DEMO folder. Here some system constants are bound in with >ILABEL ":Include/Amiga.L"<.

MACRO

Syntax:

```
name:MACRO [parameter{,parameter}*]
...;here is the program
.ENDM
```

name and **parameter** are variable names. **name** is the name of the macro, and the **parameters** are the local parameters passed to the macro. The following parameter types can be declared:

$parameter:	symbol
%parameter:	constant
*parameter:	address label

parameter: Address label or constant depending on whether an address label appears in the expression passed to the macro or not.

A macro is invoked by writing its name and parameters:

```
PRINTLINE:MACRO $\ADDR
·PEA \ADDR
·BSR PRINT_TEXT
·ADDQ.L #4,SP
·ENDM
...
·PRINTLINE TESTTEXT
...
TESTTEXT:DC.B 'This is a string',0
```

The number of parameters is not limited. However, all parameters must appear on the line containing the macro call. You can also pass fewer parameters to a macro than you have defined, but not more:

```
TEST:MACRO %\1,%\2
·DC.B \1
·IFD \2
·DC.B \2
·ELSE
·DC.B 0
·ENDIF
·ENDM
```

You can call this macro with one or two parameters:

`·TEST 1` or `·TEST 0,5`

without getting an error message. Only those parameters which are passed are defined. From within a macro you can also call another one, or even the same one. 30 nestings are allowed. You should remember that a macro call is handled internally like an INCLUDEd external source file. So if you call a macro from an external source file, only 28 nestings are possible, and if this source file was called from another one, only 27 levels are possible, and so on. It is rather improbable that you will exceed this limit in a normal program.

It is not legal to define a macro within another one.

IF (conditional assembly)

Syntax:

```
·IFcc expression, expression
....
·[ELSE]
....
·ENDIF
```

For cc you can use any condition code except T, F, and RA. expression is an arithmetic expression. The condition is evaluated exactly as for the Bcc instruction. If the condition is fulfilled, the assembly continues with the next instruction, up to ENDIF. If the assembler encounters an ELSE along the way, assembly continues after the ENDIF. If the condition is not fulfilled, assembly continues after ELSE, if present, if after ENDIF.

Example:

```
·IFHI 0,3
·NOP ;this will be assembled because 3>0
·ELSE
·DC.W 2
·ENDIF
·IFLT 0,1
·NOP
·ELSE
·DC.W 4 ;this will be assembled because 1<0
·ENDIF
```

The IF pseudo-op becomes interesting when variables appears in the arithmetic expressions. Such an example is given for the REPEAT pseudo-op.

There are two additional pseudo-ops available:

1. IFD variable

Tests whether the variable is defined and then behaves just like the normal IF pseudo-op. The variable cannot be a replacement symbol, because this, if defined, would be replaced by the corresponding replacement string and the assembler would do the wrong thing.

2. IFND variable

This is the opposite of IFD. This directive tests whether or not the variable is undefined. The same limitations apply here as for IFD. An arithmetic expression can be specified with both of these pseudo-ops. The pseudo-op then tests whether all of the variables which appear in the expression are defined/undefined.

It gets difficult when the variable is only undefined in pass 1. Then the program segment enclosed by IFND - ELSE (or ELSE - ENDIF for IFD) is executed only during pass 1. This is allowed only when this program segment contains only variable definitions and no 68000 instructions. No forward references are allowed in the variable definitions:

```
·IFND FLAG
FLAG=1
M:MACRO
;
·ENDM
K1=15
K2=LABEL ; K2 contains the value 0 because LABEL
; is not defined in pass 1
·ENDIF
\1=0
LABEL:
```

You have to be careful of a local variable defined directly following ENDIF. These then belong to the last defined global variable. In the example above, this is the constant K2 during pass 1, but during pass 2, it is the last global variable before the IFND. The result is that the assembler will not find \1 during pass 2.

If you want to program something like this, you must make sure that a global variable follows the ENDIF and not local.

You cannot nest more than 16 IF constructs deep.

REPEAT (repeated assembly)

Syntax:

```
·REPEAT
...
·UNTILcc expression,expression
```

cc may again be any condition except T, F, or RA and expression is a numeric expression. To spare you long-winded explanations, here is an example:

```
N@='0'
·REPEAT
·DC.B N@
·IFEQ N@,'9'
N@='A'
·ELSE
N@=N@+1
·ENDIF
·UNTILHI 'F',N@
```

This little program creates the same code as the following line:

```
·DC.B '0123456789ABCDEF'
```

The redefinable variable name must begin in column 1 in order for us to assign a value to it.

A maximum of eight REPEAT pseudo-ops may be nested inside each other. There is a more extensive example on the AssemPro disk in the HAM-Demo.ASM program.

INPUT

Syntax: `·INPUT [message,]variable`

message is a string and **variable** is a variable. The important thing about this pseudo-op is that no variable name may appear in front of it! The variable may be global, local, and redefinable. You can declare the following types:

$variable :	symbol
%variable:	constant
*variable :	address label

variable : Address label or constant depending on whether an address label appears in the expression passed to the macro or not.

When the assembler comes to the INPUT pseudo-op during pass 1 (nothing happens during pass 2 since you already entered it), a requestor containing a **message** (if you didn't specify one, a ? appears) appears which asks you to enter a value. Whether you enter an arithmetic expression or a string depends on whether you want to define a constant or a replacement symbol.

The INPUT pseudo-op has the same effect as the normal definition of a variable with = (constant) or EQUR (replacement symbol), except that the value can be set during assembly. Instead of

```
CONSTANT=3
```

you can also write

```
.INPUT 'CONSTANT =',CONSTANT
```

and then enter 3 in the dialog box.

The INPUT pseudo-op is most useful when you are preparing different versions of a program. You then write the code for all versions in a single source file and use conditional assembly pseudo-op to generate the appropriate version. At the beginning of your program, you insert an INPUT pseudo-op to determine which version should be assembled.

PAGE, LIST, and NOLIST

These directives affect only program listings, if you create one. They have the following meaning:

`PAGE:` Inserts a form feed

`NOLIST:` Turns the listing off

`LIST:` If the listing was turned off with NOLIST, this turns it back on.

None of these directives will appear in the output. If you don't produce a listing, they will have no effect. In particular, you cannot use LIST to override the option if it is turned off in the menu.

The Assembler Menus

The File menu

New <Amiga><N>

Erase the program code which you last created.

Open <Amiga><O>

Opens a previously created file.

Save <Amiga><S>

Saves the program code in the current directory. The name in the title line of the current editor window is used as the name, except that the extension is removed. The extension consists of the characters after and including the period. If there is no extension, .PRG is simply added.

TEST.ASM will become TEST

DEMO will become DEMO.PRG

Save as

Here you can enter the name under which the program is saved.

Erase <Amiga><D>

A **Erase File** requestor appears which allows you to erase files from the disk.

Rename

A **Rename file** requestor appears that allows you to erase files from the disk. Enter the filename to change in the **File** gadget and select **OK**. The **Rename** requestor reappears, enter the new name in the **File** gadget and click **OK**. The file is renamed.

Compare

A **Compare file** requestor appears that allows you to compare files on the disk. Enter the first filename to be compared in the **File** gadget and select **OK**. The **Compare file** requestor reappears, enter the second filename to be compared in the **File** gadget and click **OK**. A requestor appears that allows you to move through the files by selecting the arrows. Selecting **Compare** will automatically compare the two files. Selecting **Enough** ends the comparison.

Directory

A **Create a new directory** requestor appears that allows you to create a new directory on the disk.

Save pre-setting

The function key definitions and the checked menu options are saved to a disk file. When AssemPro is loaded the check options are active and the function key definitions are loaded.

Icon

If you want your program to be executable from the Workbench, it needs an icon. If **Icon** is checked, an icon will be created when you save the program.

Back-up

When this item is checked a backup copy of files saved is created on disk. The filename is preceded with "Back-up of". An icon is not created.

Backup Copy

When this item is checked a requestor appears where you can enter the device that the backup copy is saved to.

Quit

Quits the AssemPro program.

The Assemble menu

Assemble... <Amiga><A>

When you want to assemble your program, select this item. Make sure that you have first set all of the other menu options properly.

The error messages which you can get are listed in Appendix A. If you want to stop the assembly process for some reason, simply press both <Shift> keys simultaneously, assuming that you have selected breakable in the **Assemble...** requestor. You will then get an appropriate (error) message and you can either return to the menu or continue the assembly.

If you don't want the error messages printed or sent to an error file, or an uncorrectable error occurs, you will get a requestor which indicates the error. The error message and the erroneous line is printed, the cursor points to the point at which the error was discovered. The file in which the line is located and the line number is also printed.

You can now correct the error by editing the line, if this is possible. If you then click on Try again, the assembler tries to assemble the line again. If it is successful, it continues to assemble the program (up to the next error).

If the file in which the error occurred is in the editor buffer, you can Save and try again before the assembler tries to assemble the line again. If you cannot correct the error, the assembly process can be terminated.

Source:

Here you can set which source file is assembled. This can either be in editor memory or in a file in the current directory on the disk. The corresponding filename is displayed in the menu. If you want to assemble the file in the editor memory, a filename is not displayed in the menu. If you have multiple editor windows open, the source file is assembled from the last active window.

Filename output

If, during the assembly, you want to know what is currently being assembled, you should check this menu option. The open files, macros, and line numbers are then printed in the assembler window as they are processed.

If you want to assemble as quickly as possible, you should not enable this option since it slows the assembler down by about 50%.

If you want to produce a listing of your program, the filenames are printed only during pass 1.

Optimize Backward BCC's

To be able to produce optimized code with an assembler is a welcome option. These possibilities are severely limited in a 2-pass assembler which allows division of the code into three segments. Thus out of all the options, only one is left to us.

All Bcc instructions for which an .S isn't given and for which the jump distance is between -128 and -2 are assembled as short jumps, saving 2 bytes, if you have checked this option.

Flag undef. variables

Normally this menu option is checked so that undefined variables are displayed with an error message. But if you just want to check your program for correct syntax, you can turn this option off. Non-existent instructions—undefined macros—are displayed in any event because these can change the length of a program dramatically.

Original Line

Here you can choose between the original line and the assembler line in the output and in the error messages. To do this you have to know what the difference between the two is.

The original line is identical to the line which was typed in. But since the assembler can't work with this form "as is," all lowercase letters (with the exception of constants and strings) must be converted to uppercase. Also, text replacement through replacement symbols is performed on the line.

Example:

You have defined the replacement ALL_REGISTERS as follows:

```
ALL_REGISTERS: EQU D0-A6
```

The original line

```
·movem.l all_registers,-(sp)
```

is transformed into the following assembler line:

```
·MOVEM.L D0-A6,-(SP)
```

It is important to note that all error messages always refer to the assembler line. You will have to decide for yourself which of the two representations is more useful.

Memory

This requestor allows you to set the size of the assembler memory.

The Output menu

Listing

With the submenu options yes and no you can determine whether a listing is produced or not. If you select **Parameter**, you will get the following requestor which allows you to format the listing.

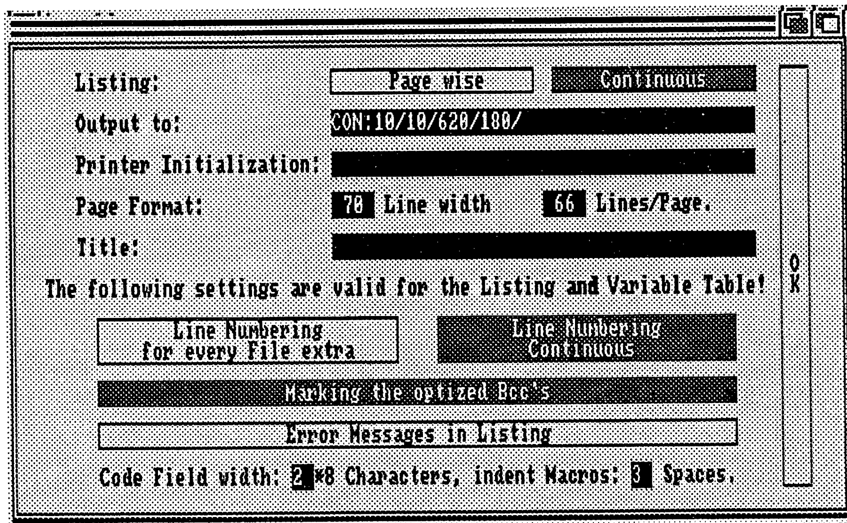


Figure 12 - Listing requestor

In the top line you can choose between **Page wise** and **Continuous** output. With continuous output, the heading is printed only at the beginning. With **Page wise** output, the heading appears at the top of each page.

Output to lets you select where the output is sent. In addition to filenames you can also specify logical devices like the printer PRT: or the serial interface SER:.

If you are addressing a printer, you should be sure to set the total width of a line correctly in order to avoid the end of a line simply being printed at the start of a new line. The output is column-oriented so that lines which are too long will be printed so that opcodes will appear under opcodes, and source text under source text.

The number of lines per page needs to be specified only if you are printing pagewise.

The created code is printed in hexadecimal in the opcode field. If you specify a **Code Field width** of zero, neither the opcodes nor the addresses will be printed in the assembly listing.

For the line numbering you can specify whether the lines are numbered according to the line numbers in source files (**Line Numbering for every File extra**) or whether the lines are numbered continuously (**Line Numbering continuous**). If you don't see the difference, try using both options with a program which either consists of more than one source file or in which you use macros.

If you want, you can mark the optimized Bcc instructions with a '>' in front of the instruction, and you can control the output of error messages in the listing, if you don't want them printed on the screen.

To make the listing easier to read you can have the text of macros and external source files indented by up to nine spaces to the right.

In the printer initialization you can enter up to 40 characters to send to your printer, including ESC codes. The data are defined just as in the DC.B instruction, for example: >27,$F,27,"E"< which switches an EPSON printer to bold, compressed print.

Variable table

Figure 13 - Variable table requestor

This requestor is similar to the one for the assembler listing. The settings above are identical to those for the listing, so we won't repeat the descriptions. Note that if you have selected both listing and symbol table output these settings will be the same for both and cannot be set independently of each other.

If you have already selected pagewise output and then select a continuous symbol table, the listing is printed continuously. This also applies to the title, the **Output to**, the **Printer Initialization** (which is only sent once), the **Line width**, and the **Lines/Page.**

There are some settings which apply only to the variable table. You can select whether the variables are printed in the order in which they were defined or whether they are sorted alphabetically. Local variables are always printed in the order in which they were defined and always behind the global variables to which they belong. In the next line you can set whether or not local variables are printed.

You can also set whether one variable is printed per line or whether as many variables as possible are packed into a line. The amount of space a variable requires depends on its length and on the printer tab setting. After a symbol has been written on a line, it is followed by as many spaces as necessary to advance to the next tab stop. Only address labels and constants are printed, with their values in hexadecimal.

Error file

If you can't wait while the assembler displays one error message after the other during assembly, you can calm your nerves with this option.

You simply specify a file and all of the error messages (from correctable errors!) are written into it.

After the assembly you will get a message in the assembler window which tells you how many errors occurred. If your program was not, contrary to your expectations, free of errors, activate the editor and load the error file. Now you can correct the errors in peace.

This option is especially useful for longer programs, because you can get a cup of coffee while the program is assembling. Only correctable errors are written into the error file. Uncorrectable errors are displayed on the screen, because the assembly must stop when such an error is encountered. Appendix A lists all of the possible errors and indicates which ones are uncorrectable.

Normal Code

If checked, normal code will be created, like other assemblers.

Normal code is like the code which other assemblers can also produce, and should generally be checked because only normal code can be started from the Workbench or the CLI.

PC-relative Code

If checked, PC-relative code is created.

For PC-relative code the assembler always mentally places a "PC" behind every label. This will lead to an error message if the label is the destination of an instruction. In spite of this, PC-relative code can be used when you want to include assembly language programs into BASIC.

The demo programs Determinants.ASM and AmigaFFP.LST show how this is done. PC-relative programs cannot be started from either Workbench or the CLI.

Absolute Code

When you want to produce absolute code, this option must be checked.

If you want to burn programs into EPROM's, you can use absolute code, because then the address at which the program is to run is set. Absolute programs cannot be started from either Workbench or the CLI.

68010 Code

If you have a installed a 68010 in your Amiga, you can also use its additional instructions in your programs if this menu option is checked. The disassembler will also disassemble the 68010 instructions if a 68010 is installed.

Including assembly language programs in BASIC

Since AmigaBASIC isn't exactly a fast language, it can be useful to write time-critical routines in assembly language and then include them in the BASIC program. But how?

First you have to choose the routines which take up the most time and rewrite them in assembly language. You must pay attention to the parameter passing conventions described in the section *"Calling Assembly Language Routines"* of the AmigaBASIC manual.

You must assemble the PC-relative program. If you create something other than PC-relative code, you will have little success. If your program consists of multiple subroutines which you want to call from BASIC, you should create a symbol table so that you know the addresses of these subroutines relative to the start of the program. After you have saved the program, load your BASIC program. First write a subroutine which reads the machine language program from the disk into an array (such as an integer array). You can also convert your program into DATA lines and include them in the BASIC program, but then you still need a routine which puts these DATA lines in an array.

An example of this is found in the DEMOS directory on your AssemPro disk. There the PC-relative-assembled program Determinants.B is bound into the BASIC program AmigaFFP.LST. You then have three subroutine calls available to open and close the MathFFP library, and to calculate the value of a determinant. The only problem is that AmigaBASIC and the MathFFP library use different formats for storing numbers.

Differences from other assemblers

The differences from other assemblers are relatively small. While there are no differences in the 68000 instructions, there are differences in the assembler directives. If you want to enter listings from magazines or books, you will probably have trouble with the command that AssemPro calls ALIGN. Other assemblers may use EVEN instead of ALIGN, or even CNOP 0,2. Instead of ALIGN.L you would then have CNOP 0,4. Here you might want to use macros to get around this:

```
EVEN:MACRO
·ALIGN
·ENDM
```

and

```
CNOP:MACRO %\1,%\2
·IFNE 0,\1
·DS.B \1
·ENDIF
·IFEQ 2,\2
·ALIGN
·ELSE
·ALIGN.L
·ENDIF
·ENDM
```

If you want to burn programs into EPROM's, you can use absolute code, because then the address at which the program is to run will be set. Absolute programs cannot be started from either Workbench or the CLI.

This brings us to the second problem, namely the parameters passed to macros. With some assemblers these do not have to be specified in the definition. In the macro itself they are called \1, \2, \3, etc. or ?1, ?2, ?3, etc. In addition, these are usually variables, so that in AssemPro you have to specify $\1, $\2, $\3, etc. in the macro definition, depending on how many parameters are used by the macro.

In addition, every source file, even those which are only INCLUDEd, must be terminated with END.

These are the essential differences which you have to keep in mind if you want to convert programs written for other assemblers to AssemPro. There are a few differences, such as the IF command, but these occur only rarely and are never used in published listings, so you probably won't have to worry about them.

4. The Debugger

The debugger is used to detect programming errors. You can also use it to investigate the operating system. The multi-tasking capabilities of the Amiga are naturally retained in the debugger. You have to pay attention to a few things, however.

You cannot debug using single-step or Start breakable if your program turns off multitasking with DISABLE or FORBID. Instead you'll have to put a breakpoint after this code and then call Start or the system will hang.

You cannot debug using single-step or Start breakable if your program uses operating system routines that output to the screen. These routines prevent other programs (including AssemPro) from writing to the screen.

Be especially careful if your program runs in the supervisor mode. Avoid supervisor mode if possible.

Guru Meditation messages are trapped by the debugger, so you can track down the source of the error. The location at which the error occurred and the contents of the registers are displayed.

The debugger window

To redisplay the contents of the debugger window, press the <Return> key. The contents are then refreshed.

```
Debugger
T·S··210···XNZVC                          0001B408 MOVE.L A7,INITIAL_SP
0000000000000000                          0001B40E MOVEA.L 4,A6
PC  = $1B408                              0001B412 MOVE.L A6,EXECBASE
SSP = $80000                              0001B418 SUBA.L A1,A1
USP = $62E5C                              0001B41A JSR -$126(A6)
L:D0                       =          0   0001B41E MOVE.L D0,OWN_TASK
L:D1                       =          0   0001B424 MOVEA.L D0,A4
L:D2                       =          0   0001B426 TST.L $AC(A4)
L:D3                       =          0   0001B42A BNE.S \1
L:D4                       =          0   0001B42C LEA $5C(A4),A0
L:D5                       =          0   0001B430 JSR -$180(A6)
L:D6                       =          0   0001B434 LEA $5C(A4),A0
L:D7                       =          0   0001B438 JSR -$174(A6)
L:A0                       =          0   0001B43C MOVE.L D0,WBENCHMSG
L:A1                       =          0   0001B442 \1:LEA INTUITIONNAME,A1
L:A2                       =          0   0001B448 MOVEQ #0,D0
L:A3                       =          0   0001B44A JSR -$228(A6)
L:A4                       =          0   0001B44E MOVE.L D0,INTUITIONBASE
L:A5                       =          0   0001B454 BEQ EXIT
L:A6                       =          0   0001B458 LEA MYNEWWINDOW,A0
L:4                        =       $676   0001B45E MOVE.L A6,-(A7)
Segment-List: $1B404                      0001B460 MOVEA.L INTUITIONBASE,A6
Task: $61E08                              0001B466 JSR -$CC(A6)
```

Figure 14 - The debugger window

The output display

The memory contents are displayed on the right side of the debugger window. You can choose one of two ways:

- Hexadecimal (memory dump)
- disassembled code

Hex/ASCII dump

If you want to display the contents of memory as a Hex/ASCII dump, each line is constructed as follows:

address - contents in hex - contents as ASCII characters

In the ASCII text display, zero bytes are displayed as periods. The address, which always refers to the first byte on the line, is given in hex. A cursor is displayed which you can control with the cursor keys or with the mouse.

You can change the contents of memory by moving the cursor to the corresponding location and then entering the new contents. You can do this in both the hex and ASCII fields. If the change is not visible on the screen, it may be because you are trying to write to ROM.

If you want to look at a different area of memory, you can do this by either entering the desired address directly by selecting the **Parameter-Display-From address** option or by pressing one of the following key combinations:

<Shift>+<cursor up>	one page back
<Shift>+<cursor down>	one page forward
<Control>+<cursor up>	8 pages back
<Control>+<cursor down>	8 pages forward

Disassembled output

If you want to see a disassembly of the memory contents, you have two options: symbolic or not symbolic. The difference lies only in the division of the line:

Normal:	address - hex code - disassembled command
Symbolic:	[label:] disassembled command

and in the number display. If you have assembled a program before, the address labels defined in it are also used by the debugger, assuming that you selected **Relocate variables** from the **Debugger** menu. You can also define labels which are displayed in this manner as well.

There is also a cursor in the disassembled output, and its position is determined by the value of the program counter (PC). If the line to which the PC points is in the output field, the line is shown in reverse video.

If you move the mouse to the line in reverse video and press the left mouse button, the mouse pointer changes into the GO symbol. If you move the mouse to another line holding the mouse button down (drag), and then release the button, all of the instructions up to, but not including, the instruction where you released the button, are executed.

You can use the cursor keys to change the position of the display. The following key combinations are allowed:

<cursor up>	one instruction back
<cursor down>	one instruction forward
<Shift>+<cursor up>	one page back
<Shift>+<cursor down>	one page forward
<Control>+<cursor up>	1024 bytes back
<Control>+<cursor down>	1024 bytes forward

You can see that the layout is only slightly different than in the Hex/ASCII mode. You should remember that you cannot disassemble backwards, so that cursor up does not move exactly one instruction/page backwards.

The scroll bar at the right of the window shows you the position of the output field relative to a given memory range. You can define this area with **Range** in the **Parameter** menu. If the memory area displayed is outside this range, the scroll bar will fill completely. Otherwise the height of the bar in relationship to the height of the window indicates how much of the memory range can be seen on the screen at the moment.

You can view other locations of the memory by moving the scroll bar. You are not guaranteed exact placement with this method, especially with a large range.

To specify the position exactly you can use **Display-From address** in the **Parameter** menu. A requester then appears in which you can specify the address at which the memory contents are displayed.

Numeric Output

You can specify that numbers are to display in hexadecimal or decimal. Numbers larger than 1000000 are always displayed in hexadecimal.

Display register

The numeric output setting affects the output of registers on the left side of the window. If **Display register** is checked, the contents of the registers are displayed in decimal or hexadecimal, assuming that the window is large enough.

At the top you see the status register and, in the line under it, its contents in binary. The meaning of the individual bits is explained in the 68000 literature. The PC (program counter), SSP (supervisor stack pointer), and USP (user stack pointer) are printed in the following lines.

Below this, you will normally see the contents of the CPU registers. The output of the status register, PC, SSP and USP is fixed. You can display the contents of any address in the next 16 lines, not just the registers. Each line is set up as below:

size:<ea> = contents

whereby size specifies whether the output has byte, word, or longword format, and is correspondingly B, W or L. For <ea>, effective address, you can specify any address, except CCR, SR and USP. To display the address instead of the contents of an effective address use:

<ea> ^ address

The address is always printed in longword format. All effective addresses that are allowed for the PEA instruction are allowed here. You cannot specify Dn, An, (An)+, -(An), #k, CCR, SR, or USP.

Register A7 may not appear anywhere. The reason for this is that AssemPro uses A7 when printing contents and addresses and not from the program being tested. This means that an incorrect value in A7 would crash the whole system.

To change the effective address of a line, double-click on it. A requestor appears and you can specify the <ea> and determine whether you want the contents or address printed, and whether it is printed in byte, word, or longword format.

Address or other errors are indicated by the appearance of a "§" instead of the "=" or "^", followed by the error number, such as 3 for an address error. It should be mentioned that symbols can also be used in the effective addresses.

In any event (even when *Display register* is not checked), the addresses of the Segment-List and the Task control block are displayed if you have loaded a program. From the segment list you can find out where the individual segments of your program are in memory and how long they are (for more information see the AmigaDOS documentation).

The Debugger menus

The Debugger menu

Disassembler

Calls the disassembler. For more information, see Chapter 5.

Reassembler

Calls the Reassembler. For more information, see Chapter 5.

Load

This loads a program from disk, which is then started the same way AssemPro was started. This means that if AssemPro was started from the CLI, then this program is started as if it were being started from the CLI. The same applies when starting from the Workbench.

No parameters are passed to the program, so your program must test whether parameters were passed to it or not. The stack size is set to 4000 bytes, unless the program has an ".info" file, in which case the stack size is taken from it. You can then set the stack size from Workbench with **Info**.

After the program is loaded and a new process is initialized, control is passed back to the debugger so that you can set breakpoints, run the program in single-step mode, etc.

Erase

Ends the program and erases it from memory. Moreover, all breakpoints that were in the program are removed. You should note that only the program is actually erased. If the program had opened a window, for example, which was still open when you erased the program, the window would stay on the screen until you reset the computer.

Relocate variables

If you have previously assembled a program, you can calculate the labels relative to the starting address of the program loaded by the debugger with this option. Then all of the defined labels will appear in the symbolic output and you can also access these labels when entering addresses.

You can relocate the symbols only once. If you erase the program and load it in again you must first re-assemble it before you can calculate the symbols.

Quit

Exit AssemPro. If a program is still in the debugger, it is erased unless it is currently running, in which case it will stay in memory until it ends of its own accord.

The Commands menu

Start

With **Start** you can start the program at the current PC.

Start breakable

In start-breakable mode, your program first executes one instruction, after which AssemPro prints the register and memory contents. The next instruction in your program is then executed. You can see how your program is running without having to call **Single Step** each time. Naturally the speed is greatly reduced because of the screen output.

You can stop the process by pressing <Esc> or the right <Amiga> key and <A>. If you try to use this method to observe operating system routines, you should be aware that some of these prevent other programs from performing screen output, causing AssemPro to hang up.

Next command

Executes the next instruction. This works just like **Single Step** except that BSR, JSR, and TRAP instructions are completely executed, including subroutine calls. This allows you to execute subroutine calls which you don't want to see in detail, without having to set breakpoints.

Single Step

When you click **Single Step**, the instruction to which the program counter points is executed.

68020-Single Step

When the program to be tested consists of long passages in which nothing of interest happens, you can use a **68020-Single Step** emulation in which the debugger is called only before one of the following instructions:

Bcc, BSR,
DBcc,
JMP, JSR,
TRAP,
RTE, RTR, and RTS (and RTD on the 68010).

Since this is only emulated, the execution speed of the program naturally decreases. In contrast to **Start breakable**, the screen output is not updated while the program is running in this mode. However, the advantage of this emulation is that the program is stopped only before branches.

Breakpoint

The best way to get out of a started program is with a breakpoint. This also makes it possible to return to the debugger only at certain locations. By the way, a breakpoint is internally an ILLEGAL instruction.

You can set a breakpoint by clicking on an address, in the disassembled or symbolic output, where you want to set the breakpoint. This address is then displayed in reverse video. Now you just have to call **set** from the **Breakpoint** submenu or press the right <Amiga> key and <B>. In the output field you will then see the instruction BREAKPOINT in the corresponding line.

Every time the program comes to this location it will stop execution and the register contents are displayed. If you want to continue with the program, you first have to remove the breakpoint.

You can remove breakpoints just like you set them. Click on the address of the line in which the breakpoint is located. Then the **erase** option in the **Breakpoint** submenu is activated and **set** will not. Call **erase** or press <Amiga>+<K> and the breakpoint is removed.

If you want to display the breakpoints in order, call **display** from the **Breakpoint** submenu. If a breakpoint is defined, it is displayed in the top line of the window. To display the next breakpoint, select **display** again.

Search

If you want to search for something, first call **for what?** from the **Search** item. A requestor is displayed in which you can enter the byte, word, or longword sequence you want to find. The syntax is identical to that of the DC instruction:

```
expression{,expression}*
```

whereby the **expression** is an expression. If you are searching for a byte sequence, you can also search for a character string, which is enclosed in apostrophes or quotation marks. Search sequences are, for example:

Byte:	`'Amiga'`
Word:	`'a','aA',65535,%101001`
Longword:	`12,'abcd',LABEL-2,$6512,%100100101001010`

As you can see, it is also legal to use labels and constants, as long as they are still in memory from the last assembly.

Now you have to specify a range of addresses in which the search takes place. You can do this with the two lowest submenu options (**from Address** and **to Address**). Then select **Search**. AssemPro will give you an appropriate message if nothing was found. If something was found and you have the Hex/ASCII mode enabled, the cursor is placed at the start of the search string.

After a successful search, AssemPro saves the address at which the search string was found. When you call **Search** again, it will always search from the last occurrence. If you want to search for something else, you have to select **from Address** and re-enter the start address. The end address remains unchanged.

Calculate

If you want to do a little calculation but you don't want to track down your pocket calculator for it, select **Calculate**. A requestor is displayed, you can enter the desired expressions in the input line. When you press <Return> or click on **OK**, the result is displayed in the **Calc. Result** = line in either hex or decimal, depending on what you selected in the **Parameter** menu.

The requestor stays on the screen so you can perform calculations until you select **Cancel**.

Change Register

When you call this menu option, a requestor appears with an input line in which you can change the contents of a register according to the following syntax:

register=value

in which no spaces are allowed. **value** is again an arithmetic expression, and the following are allowed as **register**:

D0..D7	Data registers
A0..A6	Address registers
USP, SSP	Stack pointer
PC	Program counter
*	Address of the top line in the output field

If you don't enter anything (the input field is completely empty), nothing is changed. After you exit with <Return>, the register field is redisplayed.

The Parameter menu

Display Register

Here you can specify whether the register contents are displayed in the debugger window or not.

Output Numbers

You have a choice between hexadecimal and decimal output of the numbers.

Display

With the top three submenu options you can select whether the memory contents are displayed as a Hex/ASCII dump, a disassembly, or symbolically. With **from Address** you can set the address at which the display starts.

Mark

Here you can mark a label which is then displayed by the debugger as a symbolic address. To do this, proceed as for breakpoints.

Click on the address at which you want the label and call **Mark** or press <Amiga>+<M>. In the requestor which appears enter the name of the label and confirm your input with <Return>. This label will then appear in the output field at the desired address.

Range

These two submenu options allow you to set the range which the scroll bar at the right represents.

5. Disassembler and Reassembler

The disassembler and reassembler are called from the debugger menu. When selected, a requestor appears where you can select the desired features. This requestor is similar to that for output since both functions have the same purpose, to output a program.

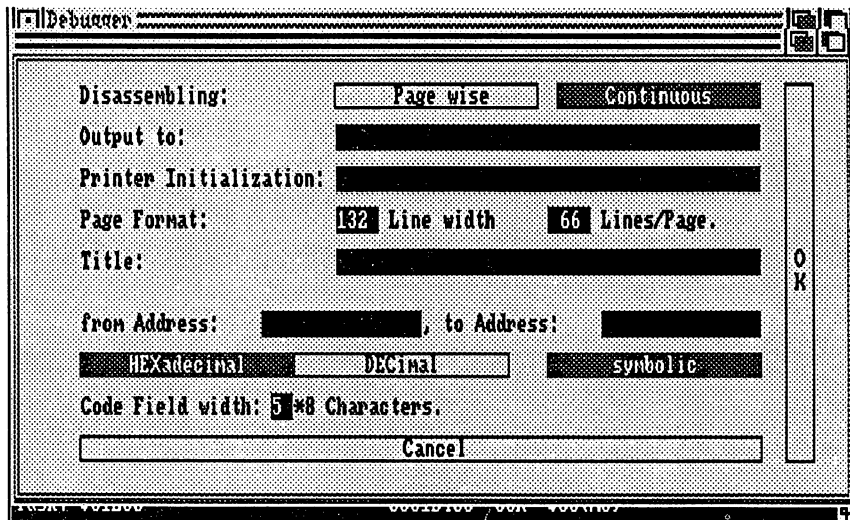


Figure 15 - The Disassembler requestor

The code is disassembled using standard Motorola mnemonics. Unimplemented instructions are translated as DC.W.

The disassembler

The requestor for the disassembler is shown in Figure 15. This just translates the code between the start and end addresses, without paying attention to whether it is a program or data.

You can use arithmetic expressions to specify the start and end addresses. If the end address lies before the start address, nothing is output. You should make sure that no area outside RAM or ROM is in the range between the start and end addresses or unforeseeable effects can occur.

For the output you can, in addition to files, specify any output device which the Amiga supports, such as the printer PRT: or the serial interface SER:.

You can also set the total width of the lines and the number of lines per page, the latter is important only if you have selected page wise output.

If you are producing continuous output, the header appears only once, at the beginning of the output, whereas if you are doing page wise output, the header appears at the top of each page. The printer initialization is sent only once, at the beginning.

Since the code is also printed in the disassembly, you can use **Code Field width** to set the number of places this will occupy on the line. The address of the code is printed in hexadecimal in front of the code.

In addition, you can set whether the output is continuous or page wise, whether numbers are printed in hexadecimal or decimal, and whether the symbols in the assembler memory are used or not, that is, whether the output is symbolic.

When you confirm your input with OK, the disassembler begins to disassemble. You can tell this because the appearance of the mouse pointer changes. When it is done and the mouse pointer appears normal again, you can start working with AssemPro again.

The reassembler

The requestor for the reassembler is identical to that for the disassembler. Since the primary purpose of the reassembler is to generate a source text which can be assembled again by the assembler, the following settings are ignored:

- number of lines per page
- header
- printer initialization
- width of the code field
- page wise/continuous
- symbolic

Reassembly is always done with continuous output and without header or printer initialization, since this would only upset the assembler if you tried to reassemble the program again. Since no code field is output, the pertinent settings are also dispensed with. All of the remaining settings have exactly the same meaning as for disassembly, so we won't mention them again.

The reassembly is always symbolic. When a instruction accesses an address, a label is defined which points to this address. Since the reassembler isn't so clever at naming things as you are (probably), the label names have the form "Lxxxxxx" where xxxxxx is a hexadecimal number which comes from the address of the label.

The created source code is terminated with END. In addition, the reassembler attempts to divide the program into instructions and data.

For this purpose the program is translated twice. The first time, the reassembler calculates the start address of all the subroutines accessed through direct addressing and the end points of the subroutines, which are indicated by RETURNs or unconditional jumps. Branches are also recognized.

The division between program and data is then made based on this information. Since this algorithm does not discover indirect jumps, the reassembler asks you to enter additional start addresses after you exit the requestor. Here you can enter the addresses of subroutines which are called through indirect addressing. These start addresses are also assigned a label, which has an 8-digit hexadecimal number so that you can tell it from the other labels.

Naturally, to do this you first have to figure out these addresses, so it is recommended that you first disassemble the program, analyze the disassembly, and then try to run it through the reassembler. You must confirm the start addresses with **OK**. If you don't want to enter any more start addresses, click **Cancel**.

Important: The last start address must also be confirmed with **OK**, not with **Cancel**.

In conclusion, it is worth mentioning that the reassembler only creates labels which point to an even address. If the program accesses odd addresses, the <ea> is given as "Lxxxxxx+1".

6. The Tables

Effective addressing modes

If the 68000 processor is new to you, you will probably have trouble at first remembering what effective addressing modes are allowed for each instruction. Even if you already know the processor inside and out, it'll still be a while before you can quote the libraries from memory.

Operating system calls

To make your work easier, you can get information about the instructions (including the assembler instructions) and the operating system calls via the help table window.

The information about the instructions is always available. For each 68000 instruction you get an overview of the addressing modes allowed. For the assembler instructions the syntax and, if necessary, a brief explanation is supplied. The allowed operand sizes are also indicated. The default is shown in square brackets. If the default is [\], then the operand size cannot be omitted, such as with EXT or DC.

The information about the operating system calls is divided into libraries. You must use the **Load Tables** item to select which library you want to access. All tables are in the "Tables" directory. Parameters and CPU registers are specified in the explanations.

If a parameter contains a return value after the call, this is indicated with a ">" in front of the CPU register (usually D0). In addition, a distinction is made between pointers to parameters and the parameters themselves, as follows:

Rn = parameter
Rn ^ pointer to parameter

The information about the operating system calls is not contained in AssemPro but in the file AssemPro.TAB. If this is not in the same folder as AssemPro, the information about the operating system calls is not loaded, as you can see by the fact that only <ea> is selectable in the menu. This allows you to save some memory. It's up to you to decide whether you want more information or more memory available.

The operation of all the requestors you encounter through the table window is the same. Either you enter the name of the desired instruction in the orange-bordered input field, or you can search for the instruction by clicking on one of the two arrows. The instructions are sorted in alphabetical order.

It's even simpler if you call the tables from the editor. Simply place the text cursor on the start of the instruction for which you want information and then press the <Help> key. The table window is then enlarged to maximum size and brought to the front. If the instruction is found in the current table, information for it is displayed.

When you have seen enough, click on the close field of the table window and it disappears.

If for some reason you don't like the information presented here, you can change it according to your own ideas.

Construction of the table files

If you're not satisfied with the information in the operating system tables, you can change it relatively easily.

The source files for the tables are in the "Tables" directory. The data, which will later appear on the screen, are stored in the files which end with ".tab".

At the start of each of these files is the actual list of the instructions. If you don't add any additional instructions, you shouldn't change this.

If you do want to add new instructions, you can insert them anywhere. You must follow the format, however:

```
·ENTRY "name",Ptr_to_Text
```

name cannot be more than 17 characters long. **Ptr_to_Text** is the label you put in front of the text which is later printed for this instruction.

Example:

```
·ENTRY "OpenLibrary",OpenLibrary
...
OpenLibrary:
DC.B "A1 ^ name of the library",0
DC.B "D0 = Version number",0
DC.B "> D0 ^ library",0,0
```

You cannot change the first and last element of the table, by the way, because these serve as start and end markers. Take a look at one of the existing files.

After the list comes the text which is printed on the screen. The strings can theoretically be of any length, terminated with a null byte. Two zero bytes follow the last line to show that this is indeed the last line.

The lines should not be longer than 75 characters so that they are completely visible in the window. Only as many lines as will fit on the screen are printed. Note that "DC.B" must appear before each line and that the strings are enclosed in quotation marks.

After you have made all of the changes, save the file again under the same name.

Now load the TABLE.Q file and assemble it as an absolute program. Store the created code on the disk as AssemPro.TAB. Voila!

Now activate the table window. If the old tables are still in memory, first click on Clear tables. Then click on Load tables, whereupon you will get a requestor which you can load the table file AssemPro.TAB.

If you want to delete entire tables, load TABLE.Q. There you delete the two lines for each table, the one that starts with ·HEADER and the one that starts with ·INCLUDE. Example:

```
·HEADER "POTGO",POTGO_TAB
...
·INCLUDE POTGO.TAB
```

Then proceed as described above, assemble, save, and load AssemPro.TAB. If you didn't make any mistakes, you will have your very own AssemPro.

7. The Libraries

Operating system calls

In the "Includes" directory are files which contain all of the libraries which are necessary for the Amiga. You can bind these into your program with "INCLUDE :Includes/name". Also in this directory are the files containing the offsets for the operating system calls. These are the files whose names end with ".Offsets".

The overhead of these fields is fairly high, since you will rarely need all of the constants in a particular file, and it is also not the fastest method.

It is therefore advisable for you to combine all of the constants you use most often and save them with SLABEL. When you later load this with ILABEL it will go much faster than when you use INCLUDE.

Alternatively you can bind in a small library with "ILABEL :Includes/Amiga.L" which contains the minimum number of constants and macros which you need to write programs.

You should then start all of your programs with INIT_AMIGA and end them with EXIT_AMIGA. "Amiga.L" also gives you the macros CALLSYS and LINKSYS.

For those who are interested, here is the format for the variables stored with SLABEL:

+ 0.B	Type
+ 1.B	Length of the name
+ 2.B	First character of the name
...	
+ n-1.B	Last character of the name. If the name has an odd length, the last character is stored in n-2. The contents of byte n-1 is then undefined.
+ n.L	Value (always at an even address!)

The following possibilities are available for the type:

= $00	Address label
= $40	Replacement symbol
= $80	Macro
= $C0	Constant

Bit 5 is also set for a redefinable symbol, so that a redefinable constant would have a type byte of $E0.

For address labels and constants, "Value" contains the value of the symbol. For replacement symbols and macros "Value" contains the number of bytes which must still be read and which contain the necessary data. For a replacement symbol this is a word for the length of the string, followed by the string itself, followed by an undefined byte if it has an odd number of characters.

With this information it is possible to write a conversion program to convert such libraries to another format, so that assembly language programs written with AssemPro Amiga could be linked with high-level compiled programs, for example.

Appendix A
Error Messages

AssemPro distinguishes between two types of errors: those which can be corrected, and those which cannot. The criterion for correctability is only valid in connection with assembling the program. Whether or not an error is correctable can be seen from the requestor in which it appears. If you only have the option of stopping the current process, the error is not correctable.

The following is an overview of the errors which AssemPro recognizes. If the meaning is not intuitively clear, a short explanation is given. First a few historical errors retained only for compatibility and should not occur:

(-1)	Error: general error
(-2)	Drive not ready
(-3)	Unknown command
(-4)	CRC error
(-5)	Invalid command
(-6)	Track not found
(-7)	Invalid boot sector
(-8)	Sector not found
(-9)	No paper
(-10)	Write error (disk full?)
(-11)	Read error (end of file?)
(-12)	Error
(-14)	Disk was changed
(-16)	Errors in sectors
(-32)	Illegal function number
(-35)	Too many open files
(-36)	Access not allowed
(-37)	Illegal handle number
(-40)	Illegal memory block address
(-46)	Illegal drive designation
(-49)	No more files

Now for the errors that come from the operating system, and are not correctable. For more information, see the Amiga documentation, such as Appendix A-2 of your AmigaDOS manual.

(103)	Insufficient free storage
(104)	Process table full
(120)	Argument line invalid or too long
(121)	File is not an object module
(202)	Object in use
(203)	Object already exists
(204)	Directory not found
(205)	Object not found
(206)	Bad stream name
(209)	Packet request unknown
(210)	Stream name component invalid
(211)	Invalid object lock
(212)	Object not of required type
(213)	Disk not validated
(214)	Disk write protected
(215)	Rename or alias across devices attempted
(216)	Directory not empty
(218)	Device not mounted
(219)	Seek failure
(220)	Comment too big
(221)	Disk full
(222)	File is protected
(223)	File is write protected
(224)	File is read only
(225)	Not DOS diskette
(226)	Diskette not inserted
(232)	Requested access not permitted

Errors generated by AssemPro which are not correctable:

(-257)	End of file
(-258)	Buffer full: Your source contains a line longer than 127 characters.
(-259)	End of file (forgot ENDM)
(-260)	End of file (forgot ENDIF)
(-261)	nnnnn ENDIF missing: You forgot a total of nnnnn ENDIFs in your program.
(-262)	Relocation error: Usually occurs when you try to assemble processor commands in the BSS segment.
(-263)	Length of TEXT field changed: In your TEXT segment (Code segment) there is at least one command which generates a different code in pass 1 then in pass 2. This is usually due to improper access to address labels.
(-264)	Length of the DATA field changed

(-265)	Forgot UNTIL
(-266)	End of file (forgot ENDS)
(-267)	False error: Occurs only when AssemPro gets really confused, which is hopefully never.
(-268)	Only global names
(-269)	Name already exists
(-270)	Name not found
(-271)	Illegal memory access
(-272)	Length of the BSS field changed
(-273)	Filename missing!

Correctable errors:

(1)	Name reserved: Symbol names may not be the same as commands or CPU registers.
(2)	Label already defined
(3)	Illegal character: You used a character which may not appear at this location.
(4)	Nesting error: You probably forgot a few ")" or had too many.
(5)	Too long
(6)	Line incomplete
(7)	Label not defined
(8)	Arithmetic error
(9)	Arithmetic overflow
(10)	Too many levels of parentheses
(11)	Operand expected
(12)	Two many operators
(13)	Operator expected
(14)	Too many operands
(15)	Division by zero
(16)	Too large
(17)	Register number illegal: for example D8
(18)	Illegal register: something like BC or IX, if you're a little confused.
(19)	<ea> error: the effective address itself is wrong.
(20)	Illegal operand size
(21)	Distance too large
(22)	Command not implemented
(23)	<ea> not allowed: This effective address is not allowed for this command.
(24)	Size specification missing (.X)
(25)	.B not allowed because destination is An
(26)	Odd jump distance

(27)	Odd address
(28)	Label expected
(29)	File not found
(30)	Access denied: File is being used by another program.
(31)	Directory not found
(32)	End of string expected: A string always ends with " or ', depending on what character it started with.
(33)	Distance = 0: Occurs mainly as a result of other errors.
(34)	Symbol expected
(35)	ORG not allowed in PC-relative
(36)	PC changed since pass 1: This error is really the result of other errors. If you already got other error messages, ignore this one. After you correct these other errors, this will normally not appear, but if it does, it means that a command in the line before the on in which this error occurred creates a different code in pass 1 than it does in pass 2.
(37)	Constant expected
(38)	No macro definition allowed: You may not define a macro inside another one.
(39)	Only global names
(40)	No macro definition: A solitary ENDM was found without a corresponding MACRO.
(41)	Too many parameters: You may not pass more parameters to a macro than you have defined.
(42)	No IF defined: A solitary ENDIF or ELSE appeared without an IF.
(43)	No additional ELSE (forgot ENDIF)
(44)	Not relocatable: Only the calculation operators + and - are allowed between labels. All other operators lead to this error message.
(45)	ORG not allowed with relocatable: ORG is allowed only for absolute programs.
(46)	Only longwords are relocatable: You may write only "DC.L Label" not "DC.W Label" or "DC.B Label", because addresses must always be longwords.

(47) REPEAT is on a different level: REPEAT and UNTIL must be in the same source file, not in an INCLUDEd file or a macro.

(48) No REPEAT defined: A lone UNTIL was encountered without a corresponding REPEAT.

(49) No SLABEL defined: A solitary SLABEL was encountered without an ILABEL.

(50) Label from illegal segment: If you want to add or subtract address labels, the two labels must be in the same segment.

(51) No label allowed

(52) Illegal name

(53) BSS not allowed: The BSS segment is allowed only in normal programs, not in PC-relative or absolute.

(54) Block moved into itself!

(55) Block copied into itself!

(56) 68010 command!: Only legal if the 68010 code option is checked in the output menu.

(57) Too many tables: AssemPro.TAB contains too many tables, so that the menu would be longer than the screen. Delete unimportant tables or switch to interlace mode.

(58) All breakpoints set!: You just tried to set the 17th breakpoint.

(59) No breakpoints in ROM!: It won't stop.

Appendix B
Directives & Pseudo-ops

All assembler directives and pseudo-ops are listed and their syntaxes given.

constant=expression
ALIGN.y
BLK.x number[,fill value]
BSS
CODE
DATA
DC.x expression{,expression}*
DEFB expression{,expression}*
DEFL expression{,expression}*
DEFM expression{,expression}*
DEFS number[,fill value]
DEFB expression{,expression}*
DS.x number[,fill value]
ELSE => ENDIF
END
ENDIF
ENDM
ENDS
constant EQU expression
replacement symbol EQUR string
IBYTES filename[,number]
IFcc expression,expression => [ELSE =>] ENDIF
ILABEL filename
INCLUDE filename
INPUT [message,]variable
LIST
name:MACRO [parameter(,parameter}*] => ENDM
MERGE filename
NOLIST
ORG expression
PAGE
REPEAT => UNTILcc
SLABEL filename => ENDS
TEXT
UNTILcc expression,expression

Meaning:

y::=(.W|.L)
x::=(.B|.W|.L)
expression, number, fill value::=arithmetic expression
constant, symbol, variable, name, parameter::=symbol name
string::=arbitrary sequence of characters
filename::=arbitrary filename; optionally enclosed in ' or "
cc::=condition as per the Bcc command, but without T, F, or RA, and with D or ND for IF
message::=string enclosed in apostrophes or quotation marks

Appendix C
Editor Short-cuts

Block commands

BA	mark start
BC	copy
BE	mark end
BH	unmark
BK	delete
BM	move
BP	insert
BV	display
BX	cut

Cursor control

CL(|S|B|W)	left
CO(|S|B|F|S|V)	up
CR(|E|B|W)	right
CU(|E|B|F|S|V)	down
C(x,y)	move cursor to position (x,y)
CN	cursor to start of next line
CV	cursor to start of previous line

Meaning: A=start, B=block, E=end, F=error, S=search, V=variable, W=word

Replace

E{(A|V|D|?}*)(L|N) string1 string2

Meaning: A=all, V=variables, D=distinguish between upper and lowercase, ?=ask before replace

Delete

LA	delete to start of line
LE	delete to end of line
LL	delete left; <Backspace>
LR	delete right; <Delete>
LZ	Delete line

Search

S{(V|D)}*(L|N) string

Meaning: V=variables, D=distinguish between upper and lowercase

Tab commands

TS	set
TL	remove

Other commands

Z string	insert string
ZH string	insert line after current line
ZV string	insert line before current line

Index

68010 Code 69
68020-Single Step 78

Absolute Code 69
ALIGN 52
Arithmetic expressions 43
Assemble... <Amiga><A> 63

BA 30
Back-up 17, 62
Backup Copy 18, 62
BC 30
BE 30
BH 30
BK 30
Block commands 30
Block menu 23
Blocks 23
BM 30
BP 30
Breakpoint 78
BSS 48
bubble-sort algorithm 1
BV 30
BX 30

Calculate 80
Change Register 80
CODE 48
Compare 17, 62
Control Keys 19
Copy 24
Cursor control 29
Cut 25

DATA 48
DC (Define Constant) 49
Delete 25
Delete commands 31
Directory 17, 62
Disassembled output 73
Disassembler 76, 83
Display 81
Display register 74, 80
DS (Define Space) 49

Edit menu 18
Editor menus 15
Editor window 14
END 51
EQU 50
EQUR 50
Erase 17, 61, 76
Error file 22, 68
error-search 22

File menu 16
Filename output 63
Flag undef. variables 64
Function keys 20

Hex/ASCII dump 72
Hide <Amiga><H> 25

IBYTES 53
Icon 17, 62
IF (conditional assembly) 57
ILABEL 55
INCLUDE 52
Info line 15
INPUT 59
Insert 25

Keypad 19

LA 31
Labels 37
Last error 23
LE 31
LIST 60
List 18, 24
Listing 65
LL 31
Load 25, 76
LR 31
LZ 31

MACRO....55
Mark....81
Mark end....24
Mark start....24
Memory....20, 65
Miscellaneous commands....32
Move....24

New....16, 61
Next command....78
Next error....23
NOLIST....60
Normal Code....68
Number formats....43
Numeric Output....74

Open....16, 61
Operators....44
Optimize Backward BCC's....64
ORG....51
Original Line....64
Output Numbers....80
output display....72

PAGE....60
Paper color....20
PC-relative Code....68

Quit....18, 62, 77

Range....81
Reassembler....76, 84
Relocate variables....77
Rename....17, 61
REPEAT (repeated assembly)....58
Replace....21, 30

Save....16, 25, 61
Save as....16, 61
Save pre-setting....17, 62
Search....79
Search backward....21
Search commands....31
Search for?....21
Search forward....21
Search menu....21
Set tab....18
Single Step....78

SLABEL....54
Source....63
Start....77
Start breakable....77

Tab commands....32
Text color....20
The Assemble menu....63
The Commands menu....77
The Debugger menu....76
The debugger window....72
The File menu....61
The Output menu....65
The Parameter menu....80
TL....32
TS....32

U....31
Undo....19

V....31
Variables....37

Write mode....19

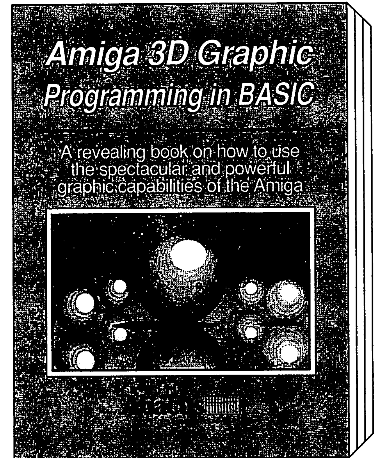
Z 'string'....32
ZN 'string'....32
ZV 'string'....32

Amiga 3D Graphic Programming in BASIC Vol.#3

Amiga 3D Graphic Programming in BASIC- shows you how to use the powerful graphics capabilities of the Amiga. Details the techniques and algorithm for writing three dimensional graphics programs: ray tracing in all resolutions, light sources and shading, saving graphics in IFF format and more.

Topics include:

- Basics of ray tracing
- Using an object editor to enter three-dimensional objects
- Material editor for creating parameters of color, shading and mirroring of objects
- Automatic computation in different resolutions
- Using any Amiga resolution (low-res, high-res, interlace, HAM)
- Different light sources and any active pixel
- Save graphics in IFF format for later recall into any IFF compatible drawing program
- Mathematical basics for the non-mathematician



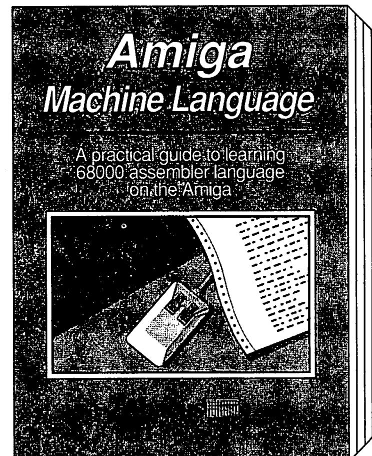
ISBN 1-55755-044-1. Suggested retail price: $19.95

Companion Diskette available: *Contains every program listed in the book- complete, error free and readyto run! Saves you hours of typing in program listings. Available only from Abacus.* **$14.95**

Amiga Machine Language Vol.#4

Amiga Machine Language introduces you to 68000 machine language programming presented in clear, easy to understand terms. If you're a beginner, the introduction eases you into programming right away. If you're an advanced programmer, you'll discover the hidden powers of your Amiga. Learn how to access the hardware registers, use the Amiga libraries, create gadgets, work with Intuition and more.

- 68000 microprocessor architecture
- 68000 address modes and instruction set
- Accessing RAM, operating system and multitasking capabilities
- Details the powerful Amiga libraries for access to AmigaDOS
- Simple number base conversions
- Text input and output - Checking for special keys
- Opening CON: RAW: SER: and PRT: devices
- Menu programming explained
- Speech utility for remarkable human voice synthesis
- Complete Intuition demonstration program including Proportional, Boolean and String gadgets



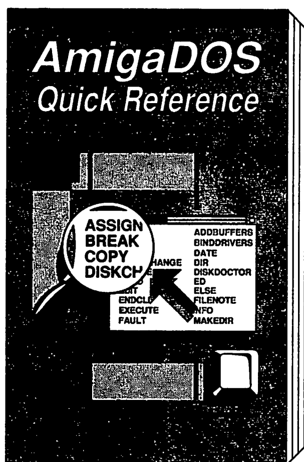
ISBN 1-55755-025-5. Suggested retail price: $19.95

Companion Diskette available: *Contains every program listed in the book- complete, error free and readyto run! Saves you hours of typing in program listings. Available only from Abacus.* **$14.95**

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

AmigaDOS Quick Reference

AmigaDOS Quick Reference is an easy-to-use reference tool for beginners and advanced programmers alike. You can quickly find commands for your Amiga by using the three handy indexes designed with the user in mind. All commands are in alphabetical order for easy reference. The most useful information you need fast can be found including:



- All AmigaDOS commands described with examples including Workbench 1.3
- Command syntax and arguments described with examples
- CLI shortcuts
- CTRL sequences
- ESCape sequences
- Amiga ASCII table
- Guru Meditation Codes
- Error messages with their corresponding numbers

Three indexes for instant information at your fingertips! The **AmigaDOS Quick Reference** is an indispensable tool you'll want to keep close to your Amiga.

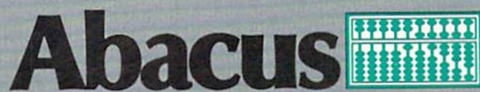
ISBN 1-55755-049-2. Suggested retail price: $9.95

Companion Diskette not available for this book.

Abacus Amiga Book Summary			
Vol.1	Amiga for Beginners	1-55755-021-2	$16.95
Vol.2	AmigaBASIC: Inside and Out	0-916439-87-9	$24.95
Vol.3	Amiga 3D Graphic Programming in BASIC	1-55755-044-1	$19.95
Vol.4	Amiga Machine Language	1-55755-025-5	$19.95
Vol.5	Amiga Tricks and Tips	0-916439-88-7	$19.95
Vol.6	Amiga System Programmers Guide	1-55755-034-4	$34.95
Vol.7	Advanced System Programmers Guide	1-55755-047-6	$34.95
Vol.8	AmigaDOS: Inside and Out	1-55755-041-7	$19.95
Vol.9	Amiga Disk Drives: Inside and Out	1-55755-042-5	$29.95
Vol.10	'C' for Beginners	1-55755-045-X	$19.95
Vol.11	'C' for Advanced Programmers	1-55755-046-8	$24.95
Vol.12	More Tricks & Tips for the Amiga	1-55755-051-4	$19.95
Vol.13	Amiga Graphics: Inside & Out	1-55755-052-2	$34.95
Vol.14	Amiga Desktop Video Guide	1-55755-057-3	$19.95
Vol. 15	Amiga Printers: Inside & Out	155755-087-5	$34.95
Vol. 16	Making Music on the Amiga	155755-094-8	$34.95
	AmigaDOS Quick Reference	1-55755-049-2	$9.95

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

The Complete Assembler Language Development Package for the Amiga



5370 52nd Street SE • Grand Rapids, MI 49512