ASM-ONE

Version 1.01

Assembler
Debugger
Monitor
Editor

for

Commodore Amiga

– English Version –

DMV-Verlag • Postfach 250 • 3440 Eschwege

Welcome to ASM-ONE, a professional assembler-package for the Commodore Amiga, which besides a fast assembler provides you with a professional editor, a screen for disassembling as well as screen-dumps and a debugger with step-, watch and jump-functions.

The creation of software, texts and pictures has been carried out with the utmost care. Errors, however, cannot fully be excluded.

We would like to point out that neither the publisher nor the authors can assume any legal responsibility or liability for damages or losses of data, which are the result of the usage of the program.

We would be grateful for suggestions for improvement and hints on software faults via our hotline.

The information provided in the product on hand are published without regard of a possible patent right. Trade names are used without guarantee of free usability.

All rights reserved, including photomechanical reproduction and storage in electronic media.

(C) 1991 DMV - Daten- und Medien-Verlag
Widuch GmbH & Co KG
Postfach 250, 3440 Eschwege

Program and manual: Rune Gram-Madsen

DTP-Setting: Andreas und Dagmar Ballhaus

Amiga, AmigaDOS, Kickstart, Intuition and Workbench are registered trade-marks of the Commodore Inc.

Printed in Germany

ASM-ONE

Table of Contents

Acknowledgements

I would like to thank my parents who trusted me enough to let me work at home on this program all summer.

Also thanks to the following people who with their hints and suggestions helped me to make this product as good as I believe it has become: Jesper Steen Møller from Danish Softech Aps, Sune Trudslev, Mads Henriksen and Michael Nielsen.

Thanks to Colin Fox (of Pyramyd Designs) and Bruce Dawson (of CygnusSoft Software) for making the requester library and letting other software developpers use it.

Also thanks to all my other friends for being so understanding and letting me work in silence. Some hellos to the following people for showing incredible interest in my new program: Florian Schroecker, Henrik Brink, Dan Jensen and Michael Holm.

Also a thank to my 2 1/2 feet tall long-haired brown-eyed dog Jason for keeping me awake when I was working late at night, and a thank to him for barking in the morning so that I couldn't sleep.

Thanks to all.

Rune Gram-Madsen

1 Introduction

Welcome to ASM-ONE! This program is an integrated environment including assembler, editor, debugger and monitor. Including all necessary functions in one tool makes program development both faster and more user friendly. You can write any type and size of program with this assembler, the only restriction is the available memory.

Short overview of the ASM-ONE macro assembler:

Integrated source level debugger: With ability to single step the program while watching the contents of memory locations, setting breakpoints, full register watch, register editing.

Compatibility with ALink and BLink: This enables high level language programmers to use machine language routines in their programs.

Fast assembling rate of 50000 - 70000 lines each minute.

Compatibility with all previous assemblers. Porting sources from earlier assemblers should not be any problem.

Many extra directives allowing more user-friendly source generation.

Integrated superfast editor with block, search, replace, mark and macro functions.

Real time full screen monitor with disassemble, hex dump, ASCII dump, address mark and jump.

User-friendly with **menus including shortcuts** on all items.

Optional **absolute memory allocation** makes this assembler ideal as game or demo developing tool.

Binary file support, with INCBIN and >EXTERN.

1.1 Intentions of this program

Programming in assembler is an activity which requires more attention to minute details than any other programming language. On the other hand it is possible to write a program that takes advantage of each of the processor instructions, this results in faster programs. Because programming is such an intensive activity, it is important that the programming tools are as fast and as flexible as possible. This is why **ASM-ONE** is integrated with **assembler**, **editor**, **monitor** and **debugger**. In this way you save time. All functions can be accessed from both menus and from the keyboard, the keyboard is recommended when you get used to this program.

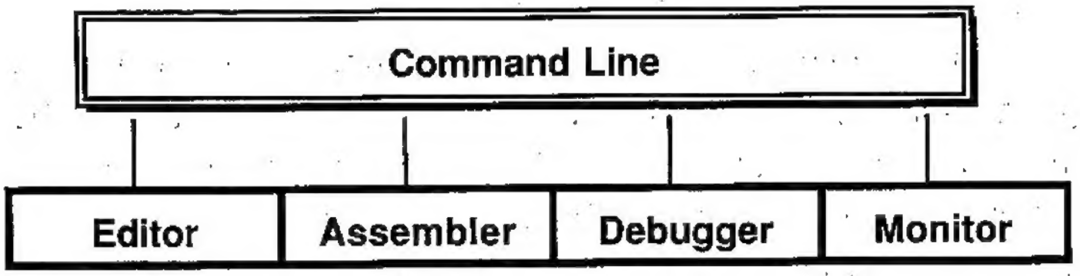
ASM-ONE is meant as both a system- and as a game or demo programming tool. This means that it is macro assembler compatible with include files, sections and so on, and it also supports direct memory placement, monitor functions etc. for non-system programmers. Both ways are supported without excluding any part of the other.

1.2 The ASM-ONE Macro Assembler

To use this program it is necessary to understand the basic functions of it.

The base of the program is the command line. You enter this after having choosen the type and size of your workspace. From here you can select any command or mode known to ASM-ONE. This is shown schematically below.

ASM-ONE structure:



When creating a program the process is as follows:

1. **Write** or **load** your **program text file** into the editor.
2. **Assemble** it and if errors occur, return to the editor (1).
3. **Try to run it.** If you discover any bugs correct these in the source in the editor (1). Or if you cannot find the errors directly enter the debugger.

Note: Before running any part of a newly written program, you should make a safety backup, because activating an assembly program gives complete control to that program.

2. Installing the assembler

The disk supplied in this package can auto-boot. In this way it is possible for you to see your new program fast. But after having made the first tests you might want to start it from workbench, use a printer or install the program on your hard-disk.

For future program execution you will have to choose one of the following 2 possibilities:

1. **Auto-booting as first time**
2. **Booting from workbench**

2.1 Auto-booting as first time

If you want to use your printer after auto-booting from the ASM-ONE disk, you will have to copy the following files from the Workbench disk onto the ASM-ONE boot disk:

Copying these files programs on a computer or from a disk carrying a virus might damage the ASM-ONE boot disk. If you are not sure that your computer and workbench disk are virus free go to 2.2.

```
devs/serial.device            to    devs/
devs/parallel.device          to    devs/
devs/printer.device           to    devs/
devs/system-configuration     to    devs/
devs/printers/<ihr Drucker>   to    devs/printers/
l/Port-Handler                to    l/
```

Be sure that »**your printer**« is the one choosen in the system configuration. You can do that by running the program called preferences. Consult your Amiga reference manual on how to check this.

The easier thing to do is just copying the entire »devs/« and »l/« directories onto the ASM-ONE boot disk.

This is the hardest way because you will have to copy most files. But maybe the most convenient afterwards because everything is on one disk. But if you are not totally sure of how to do this go to 2.2.

2.2 Booting from workbench

This is much easier to do than the above method. Because you just have to copy the following file from the ASM-ONE boot disk to the workbench disk:

```
libs/req.library    to    libs/
```

Now you can boot your workbench as usual, insert the ASM-ONE boot disk and click the ASM-ONE load symbol.

3. Getting started

After having loaded the program (see chapter 2) you are met with the following prompt:

```
ALLOCATE Fast/Chip/Abs>
```

If you got fast memory write »F« otherwise »C« and press »Return«. Now you are asked how much memory you want. (This is the memory for both your source code and the assembled program):

```
WORKSPACE (max.???) Kb>
```

Type the amount and press return. (100 kb is sufficient for the examples on the disk). You have now entered the command line and are met with the prompt:

```
>
```

This means that the choice is yours. From here you start doing what you want. To get started you can load from the »Examples« directory »**GettingStarted.S**«. Choose the read file item from the »Project« menu. A big file requester will be opened. Click on the »Examples« directory and double click on the »GettingStarted.S« file.

The file is now loaded, if you would like to watch it before running it select »Edit« in the »Assembler« menu. Now you can use all the options in the »Edit Funct.« menu. To exit the editor, press »Esc« or select »Exit« in the »Edit Funct.« menu.

You are back in the command line. Assemble the program by typing:

```
>a
```

Press »Return«. The response will be:

```
Pass1..
Pass2..
No Errors
>
```

To start the program type:

```
>J
```

Press »Return«. Now the program is running, press »Space« to exit the program.

You can try to assemble the rest of the programs in the examples directory. Try to look through the different menus. To activate not activated menus, select »Editor«, »Debugger« or »Monitor« in the »Assembler« menu.

If you already know something about machine code, skip the beginners chapter.

Good luck with your new programming tool!

4. Beginners chapter

If you already know something about machine code (you have tried to code before) you can skip this chapter, it is meant as an basic introduction to the language and also an introduction to different systems of arithmetical notation.

4.1 Why machine code?

The main reason is that machine language is the only language the MC68000 processor understands. You can create programs in Basic and in C, but both languages have to be translated into machine code before the computer can run them. In Basic the program is translated one line at a time while it is running. Of course this takes time. In C the entire program is translated at once. But because each command is translated into several machine code commands it is not possible to put the commands together in the most efficient way. This can only be done in machine code.

Another explanation can also be given. If you want to understand how the machine works (what different parts of the hardware do), machine code is a very good way to do this. With machine code you can access any part of the machine in exactly the way you want. That is why most games are written directly in machine code. Many user programs are also written directly in machine code (like this assembler). People who write programs in other languages than machine code often have to write small parts of their programs in this code to solve otherwise unsolvable problems or to increase speed.

I personally have always choosen machine language above all other languages because I like to have complete control over the machine and I enjoy speedy programs.

4.2 The machine

This Amiga (and most other computers) is based on the same concept. It has a central processing unit (MC68000), a main storage (Random Access Memory, RAM), some special purpose chips (the Amiga has become famous because it was born with unusually many help chips: **Agnus** (can create animations etc.), **Denise** (graphics in 4096 colors), **Paula** (4 channel sound and control of external devices). The Amiga also got an operating system placed in ROM (Read Only Memory). The **operating system** is actually just a program, which takes care of nearly anything that takes place in the computer. It loads your program, controls the mouse, and place the arrow at the right place on the screen. The screen and all other graphics on it are also drawn by the operating system.

The operating system takes up 256 kilo bytes (256 kb). Actually 256 kb is not exactly 256 000 bytes but 262144 bytes. This is because that the machine uses wires carrying 5 volt or 0 volt (hi or low). To access 1000 bytes it uses 10 wires. With 10 wires it is possible to address

2^10 = 2*2*2*2*2*2*2*2*2*2 = 1024 bytes

instead of just 1000 bytes as you might have expected. The **main memory** of your computer (if you have 512 kb) is 512*1024 = 524288 bytes long. Each byte consists of 8 bits. A bit can contain only two values, 1 or 0 (high or low). This means that a byte can contain 2^8 = 256 different values, normally interpreted as 0 to 255 but also as -128 to 127.

4.3 Systems of arithmetical notation

Maybe without you noticing it, you have already been introduced to a new arithmetical notation. I have mentioned bits carrying values 0 and 1, in the normal decimal notation each position in a number can be from 0 to 9. The **binary notation** is different in the way that it has only two possibilities on each position (0 and 1). Let's take the following number:

n = %100111 (% means binary)

To translate this value into decimal, try to use the build in calculator in the assembler. Just type:

```
>?%100111
```

The result will be:

```
   Hex       Decimal  ASCII                 Binary
---------------------------------------------------------------------
$00000027      39     "...'"  %00000000.00000000.00000000.00100111
```

As you see »%100111« is equal to the decimal value »39«.

Another more theoretical way to do this is as follows:

A normal decimal number is put together by ones, tens, hundreds etc. A binary number is put together by 1, 2, 4 etc. So to translate the binary value into a decimal value, do like this:

% 1 0 0 1 1 1 = 1*32 + 0*16 + 0*8 + 1*4 + 1*2 + 1*1 = 39

To convert the other way is also possible:

```
39 =  39/32   = 1*32   + 7
      7/16    = 0*16   + 7
      7/8     = 0*8    + 7
      7/4     = 1*4    + 3
      3/2     = 1*2    + 1
      1/1     = 1*1    + 0
   = %100111
```

As you see, it is a little more difficult to translate the other way around. One bad thing about binary numbers is that they are much longer and therefore harder to read than decimal numbers. We have to use them because that this is the way numbers are stored in a computer.

To make the binary numbers easier to read we normally use **hexadecimal notation.** In hex each number can be from 0 to 15, the numbers 10 to 15 are a problem so we call them »A« to »F« like this:

Decimal Hexadecimal	Decimal Hexadecimal	Decimal Hexadecimal
0 = 0	6 = 6	12 = c
1 = 1	7 = 7	13 = d
2 = 2	8 = 8	14 = e
3 = 3	9 = 9	15 = f
4 = 4	10 = a	
5 = 5	11 = b	

Let´s take the binary number from before:

n = %00100111

The binary number can easily be converted into hex:

0000 = 0	0100 = 4	1000 = 8	1100 = c
0001 = 1	0101 = 5	1001 = 9	1101 = d
0010 = 2	0110 = 6	1010 = a	1110 = e
0011 = 3	0111 = 7	1011 = b	1111 = f

So »%00100111« is »$27«.

It is possible to create all sorts of different arithmetical notation, but the three systems you have learned now are the most used.

4.4 Memory map.

Each byte has its own address. You could access address 300000. This would be in the chip memory (the memory where you can show graphics and play sound). Here follows a simple map of your memory:

$000000-$07ffff	Chip memory
$080000-$1fffff	Expansion area for chip memory
$200000-$9fffff	Expansion area for fast memory
$a00000-$beffff	Reserved
$bfd000-$bfdf00	PIA B even addresses
$bfe001-$bfef01	PIA A odd addresses
$c00000-$c7ffff	Expansion RAM area Amiga 500
$c80000-$dfefff	Reserved
$dff000-$dfffff	Hardware registers for special chips
$e00000-$e7ffff	Reserved
$e80000-$efffff	External area
$f00000-$f7ffff	Expansion ROM
$f80000-$fbffff	Future kickstart ROM
$fc0000-$ffffff	Kickstart ROM

As you can see it is possible to expand your main memory up to 10 mega bytes. But to get the extra 1.5 mega bytes of chip memory you must buy the new Super Fat Agnus.

4.5 The program

To store your program you use the main memory. The program is stored as a long list of numbers (a serie of zeros and ones), each number has its own meaning to the processor. These numbers are instructions. Each instruction can consist of 2 to 10 bytes.

One of the shortest instructions is the RTS (Return From Subroutine), it has the value %0100111001110101 in binary, $4e75 in hex and 20085 in decimal. Most instructions are much more complicated, because they can have various operands.

It is impossible to remember all these numbers so that is why you use an **assembler**. The assembler understands the name of all the possible commands. So if you write »RTS«, the assembler will automatically translate this into the number $4e75 as mentioned before.

There are of course many other commands, but it is possible to make a little program without knowing them all.

Two new commands:

```
MOVE      <source>,<destination>
ADD       <source>,<destination>
```

The move command will move the source value to the destination address. The add command will add the source value to the content of the destination address.

With these two commands you can now make your first program. To do this you will have to enter the editor. But first delete everything in the editor.

To delete everything choose the »Zap Source« command in the project menu.

Now enter the editor by pressing »Esc«.

A simple program made with the three above commands could be:

```
START   MOVE.L   #0,$000000      ; Zero the address
                                 ; (32 bit)
        ADD.L    #$1234,$000000      ; Add $1234
                                 ; to the address
        RTS
```

After typing the program into the editor leave it by pressing »Esc« again, and try to assemble by typing:

```
>A
```

The result should be:

```
Pass 1..
Pass 2..
No Errors
>
```

If you get any error message, enter the editor and find and correct the bug. After having assembled your program successfully, run it by typing:

```
>J
```

To see the result in the memory type:

```
>H.L 0
```

The first line of your output will look something like this:

```
00000000 00001234 00000676 00.....
```

The first column you see is the address you are looking at in the memory. The next value is »00001234«. This is the content of address »zero«.

This is correct. If you look at the program again you will see that the value zero is written into address »00000000«. And next the value »$1234« is added to to content of address »00000000«. What has happend is that the value »0+$1234« is put into the address »00000000«.

4.6 The registers

When you are making a program you will often use a value in a calculation or a counter. You could use a memory address to store to number, but if you use this value a lot you could use a register instead, because this is faster. The registers are like memory addresses but they are kept inside the processor. On the MC68000 we have 15 different registers that are available to the user.

These registers are of two main classes:

D0-D7	**Data registers**	**32 Bit**
A0-A6	**Address registers**	**32 Bit**

The data registers are the most flexible ones. They can be accessed as both bytes (8 bit), words (16 bit) or longwords (32 bits). You can make all sort of logical or arithmetical operations on these registers. The address registers can only be accessed as words and longwords, but can be used as pointers to data in memory.

We could take the addition from before, but this time just do the calculation in the data registers instead of the memory.

```
START   MOVE.L  #0,D0       ; Zero D0 (32 bit)
        ADD.L   #$1234,D0  ; Add $1234 to D0
        RTS
```

Assemble and run this little example like before. Now watch the register »D0« when the routine has been executed.

You see something like this:

```
D0:00001234 00000000 00000000 00000000 00000000 00000000 00000000
A0:00000000 00000000 00000000 00000000 00000000 00000000 00012345
```

The number just after »D0:« is the contents of register »D0« the next value is »D1« etc. Just watch »D0«, you see.??

We could try to make an more advanced example where we use the address registers together with some new commands.

The new commands are:

LEA	**load effective address**
TST	**test if operand is zero**
BEQ	**branch if result is zero**
BRA	**branch always (like GOTO in Basic)**

So here is the program; what does it do?

```
START   LEA.L   BUFFER,A0   ; Put buffer
                            ; adress into A0

LOOP    TST.B   (A0)        ; Test if A0 points
                            ; to value 0
        BEQ     END         ; if 0 then end
        ADD.B   #1,(A0)     ; add 1 to the adress
                            ; A0 points to
        ADD.L   #1,A0       ; add 1 to the
                            ; pointer A0
        BRA     LOOP        ; Branch to loop.
                            ; Do it again.

END     RTS               ; Return from subroutine

BUFFER  DC.B '  G KKN',0    ; The buffer with
                            ; the secret text
```

Try to assemble and run it with »J«. Try to type »H.BUFFER«, look at the char dump to the right. If the program is correct there should be a little message for you.

There was another new statement in this program (the DC.B). This is not a machine code command, but it tells the assembler to put the constant value that follows this statement into memory. So you could write:

```
DC.W 1000
```

This would put the constant value »1000« into memory. As you see in the above example it is also possible to put text into memory. This is because each character is converted into a number. These numbers are called **ASCII codes**. This is a standard way to translate characters into numbers and vice versa. To list the entire ASCII table would take up too much space, but some examples can be given:

0 = $30	A = $41	a = $61	. = $2e
1 = $31	B = $42	b = $62	; = $3b
2 = $32	C = $43	c = $63	? = $3f
3 = $33	D = $44	d = $64	

If you want to know the numbers of other ASCII numbers the build in calculator is capable of translating characters into numbers:

```
>?'A'
```

This will give you:

```
Hex        Decimal  ASCII    Binary
$00000041  65       "...A"  %00000000.00000000.00000000.01000001
```

So far, some of the principles of machine language have been explained to you. If you want to know more about the processor, read the chapter »The Motorola MC68000«.

5 The Editor

The buildt-in editor is fast, it has a text output of more than 30000 characters per second. It also supports a lot of helpful block operations and cursor manipulations.

To enter the editor select »**Edit**« in the »Assembler« menu, or press »**Amiga-Shift-E**« or just »**Esc**«. To open a half sized editor window press »**Ctrl-Esc**« instead of just »**Esc**«.

From the editor mode you can use all the functions listed in the »**Edit Funct**« menu or use a **shortcut** (»Amiga«, »Ctrl«, »Shift«, »Alt« + a key) to activate the command. To enable you to activate most of the key commands with one hand, the »**Ctrl**« **key** can be used as well as the normal »**Amiga**« **key**. A **big letter** in the menu specifies that you will also have to press »**Shift**« to get this function.

5.1 Summary of editor commands

Block commands:

»Amiga« or »Ctrl« +

	b	- Mark block
	c	- Copy block
	x	- Cut block
	i	- Insert block (same as »Fill block«)
	f	- Fill block (same as »Insert block«)
	u	- Unmark block
l	**l**	- Lowercase block
L	~~**k**~~	- Uppercase block
	y	- Rotate block
	k	- Registers used
	w	- Write block (for printing specify »PRT:«)

Search, Replace:

»Amiga« or »Ctrl« +

S	- Search start
s	- Search forward
R	- Replace start
r	- Replace forward

Jump commands:

»Amiga« or »Ctrl« +

!	- Mark 1
@	- Mark 2
#	- Mark 3
1	- Jump 1
2	- Jump 2
3	- Jump 3
J	- Jump to mark »;;«
j	- Jump to line

Move commands:

»Shift« +

up	- Page up	
down	- Page down	
left	- Begin of line,	BOLN
right	- End of line,	EOLN

»Amiga« or »CTRL« +

a	- 100 lines up
z	- 100 lines down
t	- Top of file
T	- Bottom of file

Alt +

left	- Word left,	LWORD
right	- Word right,	RWORD

Fast delete:

»Ctrl« +

Del	- Delete to end of line
Back	- Delete to beginning of line

»Amiga« or »Ctrl« +

d	- Delete line (remember)

Others:

»Amiga« or »Ctrl« +

m	- Do macro
M	- Start macro creation (end macro creation)
g	- Grab word to diary buffer, (press cursor up in input line to get the word)
DEL	- Delete one char
BACKSPC	- Back delete one char

If »NumLock« is selected you can use the numeric pad to move the cursor.

Special help to some of the editor commands:

»J« - Jump to mark »;;«

Place a »;;« mark on one or more lines in your source file. Pressing »amiga-shift-j« will **jump** to the nearest following »;;« **mark**. You can use this to mark each new part of your program.

»!«,»@«,»#« - Mark 1,2,3

You can **mark** a **special location** in your source, it is not a physical mark, just a location stored within the assembler. So if you want to be able to jump fast to a specific routine, just mark the first char of this routine, and jump to the mark when you want.

»M« - Start macro creation

The macro definition works as a tape recorder, it records all key pressions. If you have to change the same thing in many lines, just record the key pressions for changing one line, and press »Amiga-m« to perform the same actions on all the others.

6 The Command Line

The command line is the main area in this assembler. You can recognize the command line by the prompt:

```
>
```

Command line instructions:

Project:

ZS	- Zap source
O	- Old source
R	- Read
RR	- Read binary
RO	- Read object
W	- Write
WB	- Write binary
WO	- Write object
WL	- Write link file
I	- Insert
U	- Update file
ZF	- Zap file
ZI	- Zap include memory
WP	- Write preferences
=M	- Add workspace memory
!	- Quit assembler

Editor:

T	[line]	- Top of file/jump to line
B		- Bottom of file
L	[text]	- Search for text
ZL	[lines]	- Zap number of lines from cursor
P	[lines]	- Print number of lines from cursor

Memory:

M	[.size][addr]	- Edit memory
D	[addr]	- Disassemble
H	[.size][addr]	- Hex dump
N	[addr]	- ASCII dump
@D	[addr]	- Disassemble line
@A	[addr]	- Assemble line
@H	[.size][addr]	- Hex dump line
@N	[addr]	- ASCII dump line
S	[.size]	- Search
F	[.size]	- Fill
C	[.size]	- Copy
Q		- Compare

Insert:

ID		- Insert disassemble
IH	[.size]	- Insert hex dump
IN		- Insert ASCII

Assemble:

A	- Assemble source in editor
@A [addr]	- Assemble to memory
AO	- Assemble optimize
AD	- Assemble debug
=S	- Symbol table print
=	(show source stats)

Monitor:

J	[addr]	- Jump to address
G	[addr]	- Go to address
K	[steps]	- Single step n steps
X	[register]	- View/edit register values
ZB		- Zap breakpoint buffer

Disk:

RS	[drive]	- Read sector
RT	[drive]	- Read track
WS	[drive]	- Write sector
WT	[drive]	- Write track
CC	[drive]	- Calculate boot checksum

Others:

E		- Load extern files
V	[path]	- View directory
>		- Direct output (to PRT: or DFn: ...)
?	[expr]	- Calculate value

Special help to the above commands:

Project:

ZS - Zap source

Zaps the source, the copybuffer and the code. The source can be obtained again by typing »O« (Old), but only if you haven't entered the editor after zapping the source.

O - Old source

This command enables you to undo the »ZS« command (Zap Source). It can also sometimes be used to rescue the source after a crash, especially if you always allocate your workspace at the same absolute address.

R - Read

Reads a file into the editor. The file can be any kind of text file (you can even write an essay in the editor). Normally an extension ».S« (Source) is appended to the name, if you do not want this extension just delete it, or if you never want it disable the »Source.S« flag in the preference menu. This read file command will delete the file in the editor. Use »I« (Insert) to add a new piece of source to the already existing source.

RB - Read binary

This reads a binary file into a memory location. After having typed the file name you will be asked the following questions.

```
BEG>
END>
```

BEGin is the first location in memory where you want the file. END is the first location after the file: So

```
BEG>$70000
END>$71000
```

will read the first »$1000 = 4096« bytes of the binary file into the memory starting at »$70000«.

If you want the entire file loaded just ignore the »END>« question like this:

```
BEG>$70000
END>
```

The binary file will then be loaded in its full length.

RO - Read object

Reads executable file (files like DIR, LIST, ASMONE etc.). The file will be put into a newly allocated memory location. The program will be relocated to begin at an absolute address. This address will be returned to you. It is now in most cases possible to execute the program. (Note: commands like »dir«, »list« etc. are depending on information sent to it from the CLI, so you will have to simulate this information if you want to activate them, this can be a little hard.)

Because the file takes up memory, the memory used by the last loaded object will be freed when loading a new object. Trying to load a object file called "" (no name) will free the memory without loading any new object.

W - Write

Writes the file from the editor. This will save the file edited in the editor as a normal ASCII file. All normal editors and word processors will be able to load this file so it is possible for you to edit other files than sources in this editor.

Normally an extension ».S« (Source) is appended to the name input line, if you do not want this extension just delete it, or if you never want it disable the »Source.S« flag in the preference menu.

WB - Write binary

Writes a raw file from any specified memory location to disk as a binary file. After having typed the file name you will be asked the following questions.

```
BEG>
END>
```

»BEG« is the first location in memory where you want the file. »END« is the first location after the file. So

```
BEG>$70000
END>$71000
```

will write the first $1000 = 4096 bytes from the memory starting at $70000 into the binary file.

WO - Write object

If a source code is assembled, it is possible to save the code to the disk as a file that can be started from the CLI (also called a load file). If you want to write a linkable file use »WL« (Write link file) instead.

I - Insert

This command enables you to insert a new piece of source to the already existing source. The name of the already existing source will not be afflicted. See »R« (Read) for more detail.

U - Update file

Writes the file currently in the editor to the disk using the same name as when loaded. Works generally like »W« (Write).

ZF - Zap file

Zaps a file on the disk.

ZI - Zap include memory

When you use include files they are loaded into memory to be assembled (read info under assembler directives). To speed things up a little these files are kept in memory instead of being loaded everytime you assemble. To free this memory when not using the files type »ZI«.

WP - Write preferences

This function will create a file on the disk called »ASM-One.Pref«, this file contains the status of your flags in the preference menu. You can load this file into the editor, and it will look something like this.

```
-RS-L7+NL+AA+RL+WB
```

You can change the preference file to do more than just setting the flags. You can change it to:

```
-RS-L7+NL+AA+RL+WB\F\200\
```

This will automatically allocate 200 kb of fast memory. The »\« is »Return«.

=M - Add workspace memory

Let's say you have allocated 200 kb of memory but this is not sufficient. You can add any amount of memory to your workspace. But if it is not possible for the assembler to allocate the new area next to the existing area, you will have to restart the assembler to allocate the new memory. See »!« (Quit) for this action.

! - Quit assembler

When selecting this command, you can do two things. Quiting the assembler or restarting it. When restarting it, any memory allocated will be freed, and your code will be lost. The assembler is like newly loaded. If you choose to quit the assembler any memory allocated by the assembler will be freed.

Editor:

T - Top of file/jump to line

Jump to the line number specified after »T«, if no number is given, then just jump to the top. Like this:

```
T100    ; Go to line 100
T       ; Go to top
T-1     ; Go to last line
```

B - Bottom of file

Go to bottom of the file in the editor.

L - Look for text

Search in the source for the text specified after »L«. Example:

```
LMOVE
```

To continue searching for the same item:

```
L
```

ZL - Zap number of lines from cursor

Zaps a number of lines specified after »ZL« from the last cursor position. Example:

```
ZL100
```

will zap 100 lines from the cursor position.

```
ZL-1
```

will zap all lines from the cursor position to the end of the source.

P - Print number of lines from cursor

Prints a number of lines specified after »P« from the last cursor position. To print these lines on the printer, select »PrinterDump« in the preference submenu (or press »ctrl + p«). Information on how to install the printer can be found in Appendix A. Example:

```
P100
```

prints 100 lines from cursor position.

```
P-1
```

prints all lines from cursor position.

Memory:

M - Edit memory

Inserts text or hex values at a specified memory location.

D - Disassemble

Enters the monitor disassemble function. This will give you an editor like view of the source where you can scroll up, down, jump etc. If you want to change a command just type it on the line where you want it and press »Return«. If you regret press »Esc« (see the »Mon Funct.« menu).

H - Hex dump

Enters the monitor hex dump function. This will give you an editor like view of the memory with both hex dump and ASCII dump. You can edit directly in the memory and scroll up and down with the cursor keys. If you want to change a longword like a vector use the »M.L« command instead. Because this will set the entire longword at once instead of one nibble at a time (see the »Mon Funct.« menu).

N - ASCII dump

Enters the monitor ASCII dump function. This will give you an editor like view of the memory with 64 chars ASCII dump. You will be able to enter a new text simple by placing the cursor at the desired location and type your text as normal (see the »Mon Funct.« menu).

@D - Disassemble

Disassembles 12 lines of code from the address specified, if no address is given, then just continue from the last address. To use disassembled lines in your source see »Insert DisAssem«.

@A - Assemble memory

Assembles commands to memory like from a good old c64'er monitor.

@N - ASCII dump

To use an ASCII dump in your source see »Insert Ascii«.

@H - Hex dump

To use a hex dump in your source see »Insert hex dump«.

S - Search

The input sequence is as follows:

```
>S
BEG>$10000
END>$20000
```

After having specified the begin/end addresses you have to specify what data to search for, you can type:

```
DATA>123 4321.L "HELLO" $5432.W %100101.B
```

Byte size is default.

F - Fill

Be careful, filling a wrong area will wipe out the entire Amiga system.

C - Copy

Copies any memory area to another location. Be careful like with the fill command.

Q - Compare

Compares two areas. If areas are not equal, the first non-equal memory location will be displayed.

Insert:

ID - Insert disassemble

This is a very powerful command, that enables you to disassemble the memory, and insert the disassembled area in your source code. The new code will be as far as it was possible supplied with labels. This enables you to assemble the area like any other source code.

IH - Insert hex dump

This command lets you insert a memory area into your source code, as »DC« statements.

IN - Insert ASCII

This command lets you insert a text area into your source as »DC.B« string statements. If a non-ASCII value occures its alternate hex value will be inserted instead.

Assemble:

A - Assemble source in editor

This is the normal assembler option, the same as on »Amiga-Shift-A«.

@A - Assemble to memory

See the »Memory« section.

AO - Assemble optimize

Assembles your source as normal, but optimize all branches to ».S« (short form). If the branch is longer than -128 to 127 it will be forced to ».L« (long jump).

AD - Assemble debug

Assembles with the debug option on.. The source will also be assembled in this way if you activate the debugger.

=S - Symbol table print

After the source is assembled you might wish to view the global labels made. These can be outputted to printer by activating the »PrinterDump« option. A symbol table will also be created when you choose the list file option.

Monitor:

J - Jump to address

This command jumps to the specified address like a subroutine jump (JSR). If no address is specified a jump will be made to the first address in your program.

To jump to the label in your program called »START« type:

```
>JSTART
```

G - Go to address

Like jump, except that only a breakpoint or an illegal command can stop this (like JMP).

K - Single step n steps

Steps n steps from the current position of the program counter. See the debugger description for an easier way to single step.

X - View/edit register values

This can be used in one of two ways:

```
>X       ; Show all registers
>XD2     ; Edit D2
```

If you just type »X« all registers are shown, including USP, SSP, SR and PC. The status register flags are also displayed with letters. All changes since last time the registers were viewed will be highlighted (underlined when output to printer).

Disk:

RS - Read sector

Reads a sector into memory. A disk contains 80 tracks on both sides, each containing 11 sectors (80*2*11 = 1760 sectors). Each sector consists of 512 bytes. The boot block is two sectors long (1024 bytes), and starts at sector 0. The disk name is placed at sector 880 (on normal disks it is).

If you want to read the 2 sectors long boot block from »df1:« into »$70000« type:

```
>RS1
RAM PTR>$70000
DISK PTR>0
LENGTH>2
```

Warning: Data from direct disk access can only be loaded into chip memory.

RT - Read track

See »RS« for explanation.

WS - Write sector

See »RS« for explanation.

WT - Write track

See »RS« for explanation.

CC - Calculate boot checksum

As mentioned under »RS«, the boot block is 1024 bytes long. The boot block is the first part of the disk the system loads into memory when you insert a disk in your drive. It tells the system what kind of disk it is (DOS or KICKstart). The boot can also contain a program. To make sure it is ok a checksum is applied. With »CC« you can recalculate the checksum on a disk. This is used if you have written your own boot block to the disk.

To calculate the checksum on »df1:« type:

```
>CC1
```

Others:

E - Load extern files

Files marked in the source with an extern directive will be loaded when typing an »E«. To load only specific numbered extern files type

```
E[number]
```

For further information see the »>EXTERN« directive.

V - View directory

View directory from the path written after »V«. This was necessary to apply because if you have disabled the requester library this is the only way you can know what is present on the disk. You type like this:

```
VDF0:
```

If you don't want to view the directory, but just want to change the parent directory, type:

```
V DF0:
```

If you want to view the last directory, and want to watch it sorted just type »V«.

> - Direct output

You use this command to direct the output to the printer (PRT:) or to a file. If send to a file, the file will carry the extension ».TXT«. This file is a standard text file and can be loaded into the editor. To end this special output type, enter an empty path.

If you want to output the text written on the screen to the file »DIARY« type:

```
>>
FILENAME>DIARY
```

If you want to end the output just enter a blank file name.

? - Calculate value

You can use all the common operators, and all the labels defined. The result will be written in hex, decimal, ASCII and binary.

To calculate the sum to »123« and »321« type:

```
>?123+321
```

If you want to calculate the square of D0 type

```
>?D0*D0
```

7 The Assembler

The assembler part is one of the fastest on the market. It assembles with a rate of 50000-70000 lines/minute on a standard AMIGA. It is Macro Assembler compatible, this insures that old source codes can be fitted on this newer assembler, without big troubles.

When a source code is ready in the editor, choose Assemble in the Assembler menu or press Amiga-Shift-A or type from the command line:

```
>a
```

to assemble the source code. If an error occurs, the line number plus the content of the line will be printed and an error message will guide you to correct the error.

If you want to break assembling, press »Ctrl-C«.

You can choose the way of how to assemble the code. The following options can be choosen:

1 - List File
2 - Pageing
3 - Halt Page
4 - AllErrors
5 - Debug
6 - Label:
7 - UCase = LCase

How these flags work:

1. If you set this flag, a list file will be printed. Flag »2« and »3« selects the way of how to output this list file.
2. See flag 1
3. See flag 1
4. If set, the assembler will assemble the entire source code without stopping after each error. It will print the line and the error text, but will continue. (You can output this to the printer by choosing the »PrinterDump« option (»Ctrl+p«).
5. This flag selects the debug assemble option. This option takes up extra memory, but if you use the debugger often, you can enter the debugger without reassembling the source.
6. If set, all labels must end with a colon »:«. The advantage is that commands can be placed on the first column. Some old assemblers use this system, so if you used an assembler like that before ASM-ONE, the flag is to set.
7. Normally set. But if cleared two names spelled in the same way, is not the same name if they use different caseing.

When the program is finished and assembled, an executable file can be created with the »WO - Write Object« command from the Command Line or menu. (If you have used »XREF« or »XDEF« and want to link the program together with other programs, just save the file using »WL - Write linkfile«.

7.1 Programs

Each line in your source code can be one of the following types:

1 - **Blank line or comment**

2 - **Statement**

3 - **Assembler directive**

Some examples of comments:

(1). A line beginning with »*« is a comment line

```
* This is my program
```

(2). A »;« placed anywhere on a line tells that the rest of the line is a comment.

```
        subq.l   #1,d0       ; rest = rest - 1
        cmp.l    d1,d0       ; equal ?
```

(3). A text string following any complete instruction or directive is a comment.

```
        moveq    #0,d0       ; zero counter
```

(4). A blank line

In general a line (statement) has the following layout in BNF notation:

```
[<label>] <opcode> [<operand>[,<operand>]...] [<comment>]
```

As you can see, a statement is seperated in fields.

label	opcode	operands	comment

The fields must be separated with one or more spaces/tabs.

7.1.1 Label field

A label is a user defined symbol, and is characterized by:

1. **Starting at the first column**
2. **Starting at any column and ended with a »:«**

Labels are assigned to the value and type of the program counter, the address of the first byte in the following instruction. Labels with the same name may not occur twice. The assembler will report the error message:

```
** Double Symbol
```

Local labels:

A local label is like a normal label, but only known in one part of the program. In that way you can use the same name more than once in a program. You define a local label like a normal label but add a period as the first letter like this:

```
.label1
```

A local label may only be referenced between two normal global labels. An examble can show how:

label	opcode	operands	comment
clear	moveq	#0,d0	
	moveq	#100-1,d0	
	lea	memory,a0	; mem to clear
.loop	move.l	d0,(a0)+	
	dbf	d1,.loop	
end	rts		

In this simple example ».loop« is only known between »clear« and »end«. We can now use the label ».loop« in other places in the source.

7.1.2 Opcode Field

The opcode may not be placed in the first column (it is only possible to place the opcode in the first column if the flag called »Label:« is set). It can only follow a space/tab or a label followed by space/tab. A opcode can be a pre-defined or user-defined symbol. Specified with:

1) **A Motorola MC68000 opcode,** mentioned later.
2) **Assembler directive**
3) **Macro statement**

A size specifier may follow an opcode. A size specifier is one of the following:

.B	**- Byte size**
.W	**- Word size**
.W	**- branch Word size (also ».L« branch long)**
.B	**- branch Byte size (also ».S« branch short)**

The size must match with the current opcode specification.

7.1.3 Operand Field

A operand can consist of one or more operands. Each operand must be separated from the opcode with one or more spaces/tabs, and each operand is separated by a »,« followed by one or more spaces/tabs. The optional spaces after the comma are an extension the motorola standard included, because it often is convenient with the operands placed in columns.

7.1.4 Comments

After a correctly ended command followed by at least one space or tab, anything else is treated as a comment.

7.1.5 Expressions

An expression is a combination of algebraic operators, operands, constants and parentheses. It may have a constant-, relative- or a truth-value. »True« is assigned the value »-1«, »false« the value »0«.

Operators:

ASM-ONE has the following operators:

1. Monadic minus, logical not	**-, ~**
2. Left/Right shift, Power	**<<, >>, ^**
3. Logical AND, OR, EOR	**&, !, ~**
4. Multiply, Divide	***, /**
5. Add, Sub	**+, -**
6. Compare	**>, >=, =, <=, <, <>**

When evaluating an expression, the operators are evaluated according to their priority. In the above list the lowest number is evaluated first.

To change this precedence, parentheses can be included. Both square and normal brackets can be used.

```
[1+2]*3 = 9     or     (1+2)*3 = 9]
```

but

```
1+2*3 = 7
```

Operands (Symbols):

A symbol is a string of up to 100 characters, each character can be one of the four given below but the first character can only be one of the first three

1. **Alphabetic character** ['A'..'Z'] or ['a'..'z']. Distinction between lower- and UPPERcase is optional.
2. **Underscore** '_'
3. **Period** '.'
4. **Alphanumeric character** ['0'..'9']

Notice that all symbols starting with a period are treated as a local symbol (see »local labels«).

A symbol can be defined in one of the following ways:

Absolute value:

EQU	- The symbol was EQUated to an absolute value
SET	- The symbol was SET to an absolute value
RS	- Define a structure of offsets
OFFSET	- Define labels as offsets
<label>	- The symbol was used as a label in an absolute program (a program using »ORG« is absolute).

Relativ value:

EQU	- The symbol was EQUated to a relativ value
SET	- The symbol was SET to a relativ value
<label>	- The symbol was used as a label

Register:

EQUR	- The symbol was EQUated to a Register.
REG	- The symbol was equated to a REGister list.

Numbers:

A number can be used in a expression, like any other constant symbol. There are five different number types:

1. Decimal	**1234 , -234**
2. Hex	**$fab, -$fc**
3. Octal	**@176**
4. Binary	**%1001001**
5. ASCII	**'abcd', "abcd", \`abcd\`**

With ASCII-strings containing less than four characters, the character values are justified to the right. To use quotes in a ASCII string, do like this:

```
Quote   '   =  ''''   or   "
Quote   "   =  """"   or   ''''
Quote   `   =  "`"    or   ``
```

7.2 Directives

The directives are not code-producing statements, but instructions to the assembler on how to create the code. Exceptions are »DC« and »DCB«.

The directives are:

Assembly control:

SECTION	Program section
RORG	Relocatable origin
ORG	Origin (absolute)
LOAD	Load address (absolute)
OFFSET	Define offsets
ENDOFF	End offset
END	End program
BASEREG	Set base register

Data definition:

DC	Define constant
DCB	Define constant block
DS	Define storage
BLK	Block (see »DCB«)
DR	Define relativ value

Symbol definition:

EQU	Assign permanent value
SET	Assign temporary value
EQUR	Assign register name
REG	Assign register list
RS	Assign relativ value
RSRESET	Zero relativ value
RSSET	Set relativ value

Macro directives:

MACRO	Start macro definition
NARG	Special symbol
ENDM	End of macro definition
MEXIT	Macro exit
CMEXIT	Conditional macro exit
REPT	Repeat code start
ENDR	End repeat area

Conditional assembly:

CNOP	Conditional »NOP« for alignment
EVEN	Force address »EVEN«
ODD	Force address »ODD«
IFEQ	Assemble if expression = zero
IFNE	Assemble if expression <> zero
IFGT	Assemble if expression > zero
IFGE	Assemble if expression >= zero
IFLT	Assemble if expression < zero
IFLE	Assemble if expression <= zero
IF	Assemble if expression true
IFC	Assemble if strings are identical
IFNC	Assemble if strings are not identical
IFD	Assemble if defined
IFND	Assemble if not defined
IFB	Assemble if blank
IFNB	Assemble if not blank
IF1	Assemble if Pass1..
IF2	Assemble if Pass2..
ELSE	Else do this if last was false
ENDC	End conditional assembly

Listing control:

PAGE	New page in list file
NOPAGE	Turn off pageing
LIST	Turn on listing
NOLIST (NOL)	Turn off listing
LLEN	Set line length
PLEN	Set page length
SPC	Skip n blank lines
TTL	Set program title
FAIL	Generate an assembly error
MASK2	No action
PRINTT	monitor output string
PRINTV	monitor output value

External Symbols:

XDEF	External definition
XREF	External reference
ENTRY	See XDEF
EXTRN	See XREF
GLOBAL	See XDEF

General Directives:

JUMPPTR	Jump start address pointer
INCBIN	Include binary file
IMAGE	See INCBIN
INCLUDE	Include a source file
INCDIR	Set include directory path
>EXTERN	Load a data file
IDNT	Name program unit
AUTO	Automatic line command

7.2.1 Assembly Control Directives

SECTION Program section

Syntax: [<label>] SECTION <name>[,<type>]

This directive creates a **new section** called <name>, with the program counter starting at zero. Or, if <name> is already defined, restores the program counter to the last address in this section.

<name>	is a string optional enclosed in double qoutes.
<type>	if included is one of the following:

CODE	create a section containing relocatable code
DATA	create a section containing initialized data
BSS	create a section containing uninitialized data

To force the code to be placed either in fast, chip or public memory, the above types can be extended with the following:

DATA_P	place data in public memory (same as »DATA«)
DATA_C	place data in chip memory
DATA_F	place data in fast memory

The same extensions can be used on »CODE« and »BSS«. Specifying »fastmem« can be a dangerous thing to do, use this statement only if it is absolutely neccessary for your program to run in »fastmem«.

Areas carrying the **extension** »_**F**« or »_**C**« will be »Auto-Allocated« when assembling. If you don´t want the area explicitly put in chipmemory while assembling, deselect the »AutoAlloc« option in the preference menu.

You can create up to 255 sections. The assembler starts with a code section carrying the name »TEXT«.

<label> will be assigned with the address of this new section.

RORG Set Relativ Origin

Syntax: [<label>] RORG <absexp>

This directive **changes** the **program counter** to be <absexp> bytes from the start of the current section. <absexp> can not be smaller than the current program counter.

The <label> will be assigned with the value of the new program counter.

ORG Set absolute Origin

Syntax: [<label>] ORG <absexp>

This directive **changes** the **program counter** to the absolute value <absexp>.

Warning: The Amiga is a multitasking system, where everything except »address $4« theoretically can be changed and put at any address, so using this directive must be done very carefully, and on own risk.

The advantage is that you know exactly on wich address your code and data is placed. Many hardware orientated game or demo programmers use this directive.

LOAD Set absolute memory load address

Syntax: [<label>] LOAD <absexp>

This directive **changes** the **output address** of the assembled code to the address <absexp>. This enables you to assemble a program to one absolute address, and store it into another.

For further details see »ORG«.

OFFSET Define offsets

Syntax: [<label>] OFFSET <absexp>

The »OFFSET« directive **starts** an **offset definition block** that has the following layout:

```
start     OFFSET     100
dat0      ds.b       1
dat1      ds.b       9
          ENDOFF
```

The <label> »start« will be assigned to the start address of this offset definition: »D0 = <absexp> = 100« and »d1 = d0 + 1«. In offsets you can create both »code« and »data«. The data area is allocated in the current section. In your source code you can use the above offset like this:

```
        LEA       start,a4
        MOVE.B    dat0(a4),d0
        LEA       dat1(a4),a0
.loop   CLR.B     (a0)+
        DBF       D0,.loop
```

Making data-addressing is faster and shorter and takes up two less bytes in the code, and four less bytes in the relocation table.

To terminate an offset definition use one of the following directives:

```
        ENDOFF
        SECTION
        OFFSET
        END
```

ENDOFF End offset definition

Syntax: [<label>] ENDOFF

See »OFFSET«

END End program

Syntax: [<label>] END

Marks the **end of** your **source** code. If nothing follows the »END« directive it is not needed, the assembler normally assembles all lines.

BASEREG Set Base Register

Syntax: [<label>] BASEREG <label>,An

Defines a register as a »**base register**«. This affects the following two addressing modes:

nn(An) and **nn(An,Rn.s)**

Normally »nn« has to be a written as a positive or negative integer. But if »An« was defined as a base register, the offset »nn« would be treated as an offset from this address:

```
                LEA         DataArea,A4
                MOVE.W      D0,DataWord-DataArea(A4)
                ---
DataArea:                   dcb.b       100
DataWord:                   dc.w        0
```

This is the normal procedure if you want to access the location »DataWord«. Now this can be written in the following way:

```
           BASEREG    DataArea,A4
           LEA        DataArea,A4
           MOVE.W     D0,DataWord(A4)
           ---
DataArea:  dcb.b      100
DataWord:  dc.w       0
```

In this way it is much easier to use an **address register** as an global pointer to a data area. All address registers can be used to point at different areas, but they can not be redefined to point to more than one base in each source.

Normally you would use a **»BSS« hunk** to store your data, and this way gives you a shorter and faster possibility to do this (»A4« is normally used by the C-Compiler as a global data area pointer).

7.2.2 Data definition

DC Define constant

Syntax: [<label>] DC[.size] <exp>[,<exp>[...]]

This directive **starts** a **constant definition**, it can be followed by one or more numbers, and these numbers will be put into memory.

Valid sizes are:

.B	**- Byte**
.W	**- Word** (word size is default)
.L	**- Long**

When the byte-size is specified, strings can be directly written without terminating commas, like this:

```
        DC.B      'hello how are you'
```

With any other size, strings are right-aligned to the chosen size.

Using »Word« or »Long« on an odd boundary, will cause a warning, but the address will be word-aligned, and the <label> will get the new value. To avoid the warning, use the »EVEN« directive to word align the address.

DCB Define constant block

Syntax: [<label>] DCB[.size] <absexp>,<exp>

The »DCB« directive lets you **define** a **block** that is <absexp> elements long. One element has the specified size and contains the value <exp>.

Valid sizes are:

.B	**- Byte**
.W	**- Word** (word size is default)
.L	**- Long**

Using »Word« or »Long« on an odd boundary will cause a warning, but the address will be word-aligned, and the <label> will get the new value. To avoid the warning, use the »EVEN« directive to word-align the address.

DS Define storage

Syntax: [<label>] DS[.size] <absexp>

The »DS« directive lets you **define** an **uninitialized block** that is <absexp> elements long. One element has the specified size. Valid sizes are:

.B	**- Byte**
.W	**- Word** (word size is default)
.L	**- Long**

Using »Word« or »Long« on an odd boundary will cause a warning, but the address will be word-aligned, and the <label> will get the new value. To avoid the warning, use the »EVEN« directive to word align the address.

BLK Block

Syntax: [<label>] BLK[.size] <absexp>,<exp>

This directive has been included to keep up **compatibility** with a few non-standard assemblers. If you are writing new programs you should use »DCB« instead.

DR Define relativ value

Syntax: [<label>] DR[.size] <exp>

The »DR« directive lets you **define a relative value.** The value put into memory will be:

```
<exp>-*              (* = current address)
```

The **size** of this value will be checked as a signed size. An example of how it can be used is shown below:

```
JUMP    LEA       DATA(PC),A0
        ADD.W     D0,D0
        ADD.W     D0,A0
        ADD.W     (A0),A0
        JMP       (A0)
DATA    DR.W      ROUTINE_PRINT         ; d0 = 0
        DR.W      ROUTINE_CLEAR         ; d0 = 1
```

The jump routine will jump to routine number put in »d0«. Valid sizes are:

.B	- **Byte**
.W	- **Word** (word size is default)
.L	- **Long**

Using »Word« or »Long« on an odd boundary will cause a warning, but the address will be word-aligned, and the <label> will get the new value. To avoid the warning, use the EVEN directive to word-align the address.

7.2.3 Symbol definition

EQU Assign permanent value

Syntax: <label> EQU <exp>

This directive **assigns** the **value** <exp> to **<label>**. <label> can now be used as a constant value in other expressions.

This statement is the same as the »=« statement.

SET Assign temporary value

Syntax: <label> SET <exp>

This directive **assigns** the **value** <exp> to **<label>**. <label> can now be used as a value in other expressions or be assigned to a new temporary value.

Note: You should never use forward references to a »SET« symbol.

EQUR Assign register name

Syntax: <label> EQUR <Rn>

This directive can be used to attach a register to a <label>. You can use this if you are coding critical areas of your code where you are using many registers.

```
Eg.  BitPlane1  EQUR      A3
                MOVE.L    D0,(BitPlane1)+
```

The name »BitPlane1« can not be redefined, but it is possible to attach many names to one register.

NOTE: Only data and address register names can be used.

REG Assign register list

Syntax: <label> REG <List>

This directive is used like the »EQUR« directive, but to this register you can attach a **list of registers** known from the »MOVEM« command.

```
Eg.     AllRegs REG       D0-A6
                MOVEM.L   AllRegs,-(A7)
```

The name »AllRegs« can not be redefined, but it is possible to attach many names to the same register list.

NOTE: Only data and address register names can be used.

RS Assign relativ value

Syntax: [<label>] RS.[size] <absexp>

The »RS« directive **assigns** the <label> to the **current value** of »**RS**«. After the value <absexp> in bytes is added to »RS«. To zero »RS« use the »RSRESET« directive.

This directive gives a fast and easy way to define data offsets.

How some library offsets could be defined with this directive:

```
        RSRESET
        RS.B     -30         ; Start value
Open    RS.B     -6
Close   RS.B     -6
Read    RS.B     -6
Write   RS.B     -6
Input   RS.B     -6
Output  RS.B     -6
```

RSRESET Zero relativ value

Syntax: [<label>] RSRESET

See the »RS« directive for explanation.

RSSET Set relativ value

Syntax: [<label>] RSSET <absexp>

Instead of clearing, the »**RS**« value is set to a **start value**. This is equivalent to writing:

```
RSRESET
RS.B    <AbsWert>
```

See the »RS« directive for explanation.

7.2.4 Macro directives

MACRO Start macro definition

Syntax: <label> MACRO

Start a **macro definition.** The <label> will become the name of this macro. To end a macro definition use the »ENDM« directive.

A macro is used to assign a name to one or more lines of code. Each time the name is used as a operator the code will be filled in. It is possible to specify operands with each macro. To refer to each operand in the macro definition you use the backslash »\« followed by a number from 1-9. A backslash followed by »@« produces the following symbol:

```
\@ = _nnnn
```

where »nnnn« is the number of times a macro is used.

An example of a simple macro:

```
CoordsXYZ  MACRO
           DC.W    \1,\2,\3
           ENDM
```

In your program you can write:

```
CoordsXYZ  10,10,10
CoordsXYZ  10,-10,10
---
---
CoordsXYZ  -10,10,10
```

If suddenly the coordinates have to be scaled by a factor of 16 you just rewrite the macro to:

```
CoordsXYZ  MACRO
           DC.W    \1*16,\2*16,\3*16
           ENDM
```

You can not define a macro in a macro, but you can refer to other macros within a macro, you can even refer to it self. To terminate macros like this use »MEXIT« or »CMEXIT«. The limit to the level of nesting macros is currently set to 25.

NARG Special symbol

Syntax: NARG

This special symbol is assigned to the number of arguments passed to the current macro. Outside a macro »NARG« carries the value »0«.

ENDM End macro definition

Syntax: ENDM

End a macro definition. See also »MACRO«, »MEXIT« and »CMEXIT«.

MEXIT Macro exit

Syntax: MEXIT

Exit a macro definition. When the assembler reaches this statement, the macro is terminated, even if the end wasn't reached.

CMEXIT Conditional macro exit

Syntax: CMEXIT <absexp>

Exit a macro definition if current level of the macro is <absexp>. If this level is reached the macro is terminated, even if the end wasn't reached.

REPT Repeat code start

Syntax: REPT <absexp>

Start repetitive block assemble. You use this to repeat one or more commands a given number of times. The block is ended with the »ENDR« directive.

Let´s say you want to move 10 bytes very fast, you could do like this:

```
REPT 10
MOVE.B    (A0)+,(A1)+
ENDR
```

ENDR End repeat area

Syntax: ENDR

End the **repetitive block assemble.** See »REPT« for the use of repeative blocks.

7.2.5 Conditional assembly

CNOP Conditional NOP for alignment

Syntax: [<label>] CNOP <absexp1>,<absexp2>

This directive **aligns** the **address** to <absexp2> and adds <absexp1> to this new address.

```
CNOP 0,4
```

Aligns to a longword boundary.

```
CNOP 2,4
```

Aligns to a longword boundary plus 2.

The <label> will be assigned to the new value of the program counter after »CNOP«.

NOTE: »CNOP« does no initializing.

EVEN Force address to even

Syntax: [<label>] EVEN

This statement has the same effect as:

```
CNOP 0,2
```

The <label> will be assigned to the new value of the program counter after »EVEN«.

NOTE: »EVEN« does no initializing.

ODD Force address to odd

Syntax: [<label>] ODD

Force address to »**odd**«. This can be replaced by:

```
CNOP 1,2
```

but »ODD« will NOT change an already odd program counter.

The <label> will be assigned to the new value of the programcounter after »ODD«.

NOTE: »ODD« does no initialisation.

IFEQ	Assemble if expression	=	zero
IFNE	Assemble if expression	<>	zero
IFGT	Assemble if expression	>	zero
IFGE	Assemble if expression	>=	zero
IFLT	Assemble if expression	<	zero
IFLE	Assemble if expression	<=	zero

The above commands can be replaced by the more general

IF Assemble if expression true

Syntax: IF <boolean>

A »boolean« consists of two expressions divided by one of the following:

=, >, <, >=, <=, <>

If the »IFxx« statement was false, the code will be skipped until an »ENDC« or an »ELSE« statement is reached.

»IFxx« statements can be nested up to a level of 25.

IFC Assemble if strings are identical

Syntax: IFC <string>,<string>

See the general »IF« command for information on how »IF« statements are treated.

IFNC Assemble if strings are NOT identical

Syntax: IFNC <string>,<string>

See the general »IF« command for information on how »IF« statements are treated.

IFD Assemble if defined

Syntax: IFD <exp>

See the general »IF« command for information on how »IF« statements are treated.

IFND Assemble if NOT defined

Syntax: IFND <exp>

See the general »IF« command for information on how »IF« statements are treated.

IFB Assemble if blank

Syntax: IFB <symbol>

This instructions tests the <symbol>, if any symbol supplied, wether the symbol is not blank. You use this to check if a macro statement is supplied or not.

IFNB Assemble if not blank

Syntax: IFNB <symbol>

This instructions tests the <symbol>, if any symbol supplied, wether the symbol is not blank. You use this to check if a macro statement is supplied or not.

IF1 Assemble if Pass1..

Syntax: IF1

If Pass 1 the following block is assembled.

IF2 Assemble if Pass2..

Syntax: IF2

If Pass 2 the following block is assembled.

See the general »IF« command for information on how »IF« statements are treated.

ELSE Else do this if last was false

Syntax: ELSE

See the general »IF« command for information on how »IF« statements are treated.

ENDC End conditional assembly

Syntax: ENDC

See the general »IF« command for information on how »IF« statements are treated.

7.2.6 Listing control

PAGE New page in list file

Syntax: PAGE

Insert a new page in the list file and begin paging.

NOPAGE Turn off paging

Syntax: NOPAGE

LIST Turn on listing

Syntax: LIST

Start creation of the list file.

NOLIST Turn off listing

Syntax: NOLIST (or NOL)

End creation of the list file.

LLEN Set line length

Syntax: LLEN <absexp>

Set line length between 60 and 132

PLEN Set page length

Syntax: PLEN <absexp>

Set page length between 20 and 100

SPC Skip n blank lines

Syntax: SPC <absexp>

Send n blank lines to the list file.

TTL Set program title

Syntax: TTL <string>

Set the program title. The title is used as the list file title.

To give a name to a program unit see »IDNT«.

FAIL Generate an assembly error

Syntax: FAIL

Generate an assembly error. This error will appear in the list file.

PRINTT Print a text string to the screen

Syntax: PRINTT <string>[,<string>,..]

Print a text string to the screen while assembling. You can use this directive to print needed information or instructions to others when they assemble your source, or use it as a reminder for yourself, like this:

```
PRINTT "This line was assembled"
```

PRINTV Print a value to the screen

Syntax: PRINTV <expr>[,<expr>,..]

Print a value to the screen while assembling. You can use this directive to print useful information, to check that everything is done correctly while assembling. This directive is especially useful if you use conditional assembling or »REPT« statements.

```
PRINTV ScreenAddress,ScreenWidth
```

7.2.7 External symbols

XDEF External definition

Syntax: XDEF <label>[,<label>..]

Defines a **name** as **external**. Each <label> may be referenced from other modules even from high language modules.

See »XREF« for an example of how to use this directive.

NOTE: Code using »XDEF« or »XREF« can not directly be executed.

XREF External reference

Syntax: XREF <label>[,<label>..]

Reference to other external definitions in other modules. Each <label> may now be handled as a normal relative symbol.

An example of how to use this directive is given below.
Assemble and save Program 1:

```
        XDEF    ClearScreen
ClearScreen
        CLR.L   Screen
        RTS
Screen  DC.L    0
```

Now assemble and save Program 2:

```
        XREF    ClearScreen
GoClear         JMP         ClearScreen
```

You now got to files. Use a linker to link these two files into one executable file.

NOTE: Code using »XDEF« or »XREF« cannot directly be executed.

7.2.8 General directives

JUMPPTR Jump Start Address Pointer

Syntax: J UMPPTR <label>

This directive **sets** the **start address** for you program when you want to debug it. Normally the start address is set to the point at the first instruction in the first nonempty section.

This is address is also the address the assembler will jump to when you issue the command line instruction »J« or »G« without an address.

INCBIN Include binary file

.Syntax: INCBIN <string>[,<absexp>]

Include binary data in your program. This data will be loaded every time you assemble.

An example can be given:

```
START:
          INCBIN   "datafile"
END:
```

If »datafile« is a binary file of the length 1000 bytes, the label »END« will carry a value 1000 greater than »START«, and the »datafile« will be loaded at the address »START«.

INCLUDE Include a source file

Syntax: INCLUDE <string>

Include a **source file** in your own source. Include files normally contain constant definitions like library offsets and so on but they can actually contain any type of code.

Create a normal source code containing all your definitions (constants, macros etc.).

Include it at top of your program and now you can use all the definitions in the code as if you had actually written them at the top of your source.

Here is how you write:

```
INCDIR     DF0:INCLUDE\     ; Where to get
                            ; the files
INCLUDE    exec/exec.i      ; What file
                            ; to include
```

You can have **multiple** includes following each others. Include files may also include other files (Up to a maximum level of 5).

To speed things up a little, loaded include files are kept in memory. This prevents that they have to be loaded every time you assemble.

To delete unused include files issue the command »ZI« - Zap Include Memory.

INCDIR Set include directory path

Syntax: INCDIR <string>

Set the **path** of where to find the include files. Default is

```
INCDIR "Df0:Includes/"
```

>EXTERN Load a data file

Syntax: >EXTERN [<num>,]<string>,<loadpos>[,<length>]

Define a reference to an extern file with the name <string>. The file will be loaded at the address <loadpos>, if <length> is applied this will be the length of the file loaded.

The »EXTERN« files is loaded when the line command 'E' is issued. If <num> is supplied, you choose only to load the extern files carrying the same number. This is done by issuing the line command E<num>.

IDNT Name Program unit

Syntax: IDNT <string>

A program **unit** containing multiple sections must have a **name**. Normally this name is set to zero, but with IDNT you can assign a name to the program unit.

AUTO Automatic line command

Syntax: AUTO <cmd>[\<cmd>..]

Auto lets you issue one or more command line commands.

```
AUTO E\
```

if you want all the extern files to be loaded automatically, every time you assemble.

Here backslash means <return>.

8 The Debugger

This debugger is both a source-level debugger and a normal debugger. This means that you can run your own source directly, as if you where running the text file directly. You see your own source exactly as in the editor, with comments, line numbers etc. But at the same time you can watch all registers and memory locations etc. If you are debugging a program where you haven't got the source you can single step in a normal disassembly window. All the commands from the source level debugger are still available. You can even shift directly between the two viewing modes while debugging the same program.

In the debugger you can use all the functions listed in the »Debug Funct«-menu. The functions are as listed:

Step One

One **single step forward** using a breakpoint. This is used to trace the main routine. Subroutines will not be traced, they will be skipped in one step. If you wish to enter a subroutine called from the main routine see the command <enter>.

Conditional branched »Bcc«, »DBcc« and return from subroutine »RTS« are single stepped using trace mode. This means that you dont have to enter these to follow where they go. »BRA« is also handled as a conditional branch.

A »JMP« command is handled specially, it is also taken in one step, but the return address is copied from the stack and used as a breakpoint. This can cause a problem, because if you have pushed data onto the stack before issuing the »JMP« command (here I mean data used by the »JMP« routine), the last longword on the stack is not the return address. To trace this set a breakpoint yourself or use the command <enter>.

Enter

This command is used to **enter** a **subroutine** that would otherwise be taken in one big step. Here we talk about one of the following commands: »BSR«, »JSR« or »JMP«. If you are running in the »ShowSource« mode (this is the default way, where you see the source text while stepping) it is not advisable to enter a library call, because the address would not be found in your source.

If this happens anyway, you can set a breakpoint and run the rest of the library routine in one bit or you can disable the »ShowSource« mode and single step the rest of the library routine by watching the disassembled version of it. Please notice that it is impossible to set a breakpoint in a ROM (Read Only Memory), so you will have to use the <enter> command all the time.

Step n

This enables you to run »**n**« **commands** in one step. This can be a little chaotic because it can be hard to see where you will end in 143 steps. The advisable way is to set a breakpoint.

Edit regs

You can while you are tracing your program **edit** or change the contents of any **register**. This will of course change the result of your routine, but in some cases it is convenient just to change a wrong value instead of correcting the bug at once, being forced to reassemble the program.

AddWatch

Watch a **memory location**. Both direct addresses, registers and location names can be used. So the following are legal watch locations.

```
BUFFER A0+D0+10
BUFFER+2+D1      etc.
```

The location can point to one of the following types of data:

ASCII, String, Hex, Decimal, Binary, Pointer.

If the pointer type is specified, it can be a pointer of one of the following types:

```
dc.l    ; Absolute long (32 bits)
dc.w    ; Absolute word (16 bits)
dr.l    ; PC relative long (32 bits)
dr.w    ; PC relative word (16 bits)
```

This pointer can point to all these types except a new pointer.

DelWatch

Each time you »addwatch« on a **new element**, this element name will be stored in the »DelWatch« menu. It is possible to remove one element from the watch list by selecting the desired element in the menu.

ZapWatch

This works like »DelWatch«, but it is just more efficient because it deletes all watched locations from the watch list.

JumpAddr

Jump to a **absolute address**, that you are asked to type at the menubar. The absolute address can be a label, a register value etc. You can even jump to A0+4, A0+D0+2 etc.

JumpMark

Your **current line** will be **highlighted**. Now move this line up or down with the cursor keys. Press <return> if you want to jump to the highlighted line, or press <esc> if you regret.

B.P. Addr

Set a **breakpoint** at an absolute address, that you will be asked to type in the menubar.

B.P. Mark

Your **current line** will be **highlighted**. Now move this line up or down with the cursor keys. Press <return> if you want to set a breakpoint at the location of the highlighted line, or press <esc> if you regred.

Zap B.P.

It is possible to set 16 breakpoints. Each time the program is stopped, the breakpoint(s) at the break location will be removed. If you want to **remove** (Zap) **all breakpoints** use this command.

Additional flags:

DisAssem	- If selected, a disassembled version of the line you are placed at will be shown. (only of interest when debugging a source file).
ShowSource	- If set, the source will be shown.

To see a little example of how the debugger works, you can load the program »BinaryConv.S« from the »examples« directory.

9 The Monitor

The monitor is not used to run programs but to examine the contents of different memory locations. It functions nearly like an editor. You can scroll up - down and **change** the contents of the **memory** as desired.

Memory can be viewed in the following ways:

1. **DisAssemble**
2. **Hex Dump**
3. **ASCII Dump**

Each mode has the same basic functions, but also dedicated functions:

DisAssemble - Special keys:

Any alpha-numeric keys:

Pressing any of the normal keys will activate »line assembly« mode. This mode is like a normal input line with all its special keys. Type the desired command and press <return>. If the command is invalid, no change will be made.

Non alpha-numeric keys: (Eg. <cursor right>)

Pressing any non alpha-numeric key will activate »line assembly« mode. But instead of starting on a blank line as normal, the current assembly line is put in to the line buffer. This makes alteration of existing code easier.

General keys:

<CTRL-Shift-B>:	Change hex output to byte size
<CTRL-Shift-W>:	Change hex output to word size
<CTRL-Shift-L>:	Change hex output to longword size
<DEL>	Pressing <Del> will put a »NOP« instruction on the current address. and move the pointer to the next address.

Hex/Ascii Dump - Special keys:

<Shift left/right>:	Jump to start/end of line.
<Alt left/right>:	Scroll screen left/right

The following is an explanation of the **global commands**, the commands that can be used in all three modes (these are also the commands you will find in the »monitor« menu):

DisAssem

Change to »**DisAssembly**« mode. It is possible to change from one mode to another without leaving the monitor. So if you while you disassemble a program encounter something that does not look like machine language, just change to »ASCII« or »Hexdump« to analyze it more closely.

HexDump

Change to »**HexDump**« mode.

AsciiDump

Change to »**AsciiDump**« mode.

Jump Addr

Jump to **address.** Type the desired address in the menu input line. If the address is invalid no action will be taken. The old address will be stored in the last buffer (see »Last Addr«), so that it is possible to return to the address you jumped from.

Last Addr

Return to the **last address** you jumped from. The »last« buffer can remember the 16 last addresses.

Mark 1..3

Mark a **location** that you want to remember. Use the marks to change between two locations.

Jump 1..3

Jump to one of the **marked locations.** Another use of the mark is if you are looking at a dynamically changing data area. Each time you press »Jump« mark the screen will be updated, even if the address was not changed.

QuickJump

Normaly when jumping you would have to type in the address. But this function works as follows:

In »Disassem« mode:

Get the rightmost longword accessed on the »current line«. If no longword is accessed, no jump is taken.

Warning: It is NOT checked wether the memory area pointed to by the longword address is legal. If the address is illegal it might cause a crash.

In »hex« or »ASCII« mode:

The longword taken from the nearst even address pointed to by the cursor is copied to the »Jump Addr« prompt. To accept the jump press <return>.

esc

Quit monitor, and return to the command line.

Additional flags:

»OnlyAscii«	- If set only ASCII codes will be showed in the »ASCII dump«. Normaly alternate codes are shown as well.

WARNING: With the monitor you can watch all memory locations, and edit in these memory locations even if they are not allocated to the system. Accessing non existing memory or I/O areas can cause a crash.

Changing memory locations used by the system can also cause a crash, or make the system malfunction. The risk is that you will loose the current source in the editor. So make a safety backup of your source before using the monitor.

10 The Motorola MC68000

The MC68000 is a 16/32 bit processor. This means that it communicates with the rest of the machine through a 16 bit data bus, but internal it communicates with full 32 bit.

To specify where to address the main memory it has a 24 bit address bus. This gives a maximum workspace of 2 ^{24} addresses = 16 MegaByte.

The processor has the following registers:

Data registers	D0 bis D7	8, 16, 32 Bit
Address registers	A0 bis A7	16, 32 Bit
User Stack Pointer	USP	32 Bit
Supervisor Stack	SSP	32 Bit
Program Counter	PC	24 Bit
Status Register	SR	16 Bit
Condition Code Register	CR	8 Bit

Data registers: (d0-d7)

These registers can be used as both counters and constants. They can also be used in calculations, both logical and arithmetical.

Address registers: (a0-a6)

These registers can be used as pointers and constants. They can be used in arithmetical calculations (not logical).

Stack pointers: A7, USP, SSP

There are two different stack pointers, »User« and »Supervisor«. The stack is used to store the return address from a subroutine. And can also be used to store registers.

A bit in the status register selects what stack is used. Normally the user stack is used, but in interrupts, and other special routines the supervisor stack is used (see »SR«). The current stack pointer is stored in »A7«, so it is dangerous to use this register in your routines.

An little example that returns the return address from a subroutine in »d0«:

```
START     BSR       SUBROUT
CONT      RTS
SUBROUT   MOVE.L    (A7),D0
          RTS
```

Assemble this example and start it with »J«. Watch »D0« after the call. Try to print the label »CONT« with »?CONT«. Is it the same address?

You can also store registers on the stack if you may not delete them:

```
START   MOVEM.L  D0-A6,-(A7)
        NOP                  ; Put your routine here
        MOVEM.L  (A7)+,D0-A6 RTS
```

Status and Conditional Code Register: SR, CCR

The »CCR« register is just the low 8 bits of the »SR« register so here follows a complete list of the meaning of each bit in the Status register:

SR and CCR registers:

Bit	Name	Meaning
0	C, Carry	Is set in calculations if result is to big.
1	V, Overflow	Like carry but just on signed numbers.
2	Z, Zero	Set if the result of an operation is zero.
3	N, Negative	Is set if the result is negative, highest bit.
4	X, Extended	Is set in arithmetic calculations like carry; is also affected by shift and rotate instructions.
5 - 7	Unused	

The highest byte of SR:

Bit	Name	Meaning
8 - 10	I0 - I2	Interrupt level I2,I1,I0 defines a value from 0 to 7, normal programs run at level 0.
11 - 12	Unused	
13	S, Supervisor	Supervisor mode. If set, all privileged commands can be executed and the »SSP« stack is used
14	Unused	
15	T, Trace	If set the trace mode is active. After each instruction the trace exception is activated. So that only one instruction is executed.

10.1 The Instruction Set

First a little list of how to read the instructions:

Label	A label or address
Rn	Address or data register
An	Address register
Dn	Data register
Source	Source operand
Dest	Destination operand
<ea>	Effective address
#n	Integer value
<list>	A list of Rn. (eg. D0/D1-D4/A0-A3 This refers to registers D0 D1 D2 D3 D4 A0 A1 A2 A3, used in the »MOVEM« instruction)

Where ever you see the »ea« operand you can use all normal addressing modes. The possible addressing modes are:

Addressing Mode	Assembler-Syntax
Data register direct	`Dn`
Address register direct	`An`
Register indirect	`(An)`
Register indirect with postincrement	`(An)+`
Register indirect with predecrement	`-(An)`
Register indirect with displacement	`d16(An)`
Register indirect with index	`d8(An,Rn)`
Absolute Short	`xxxx.W`
Absolute Long	`xxxxxxxx`
PC relative with displacement	`d16(PC)`
PC relative with index	`d8(PC,Rn)`
Immediate	`#xxxx`

In some instructions (Bcc, DBcc, Scc) we read »cc«. This means that instead of these two characters you write the conditional code:

cc	Meaning	Bits
T	True	
F	False	
HI	Higher	C'* Z'
LS	Lower or Same	C + Z
CC,HS	Carry Clear, Higher or Same	C'
CS,LO	Carry Set, Lower than	C
NE	Not Equal	Z'
EQ	Equal	Z
VC	oVerflow Clear	V'
VS	oVerflow Set	V
PL	PLus	N'
MI	MInus	N
GE	Greater than or Equal	N*V+N'*V'
LT	Less Than	N*V'+N'*V
GT	Greater Than	N*V*Z'+N'*V'*Z'
LE	Less than or Equal	Z + N*V'+N'*V

* = Logical AND,
+ = Logical OR,
' = Logical NOT

Following here is a list of instructions of the MC68000 processor, for more information consult the original documentation from Motorola Inc.

Mnemonic	Description
ABCD	Add binary coded decimal with extend
ADD	Add
ADDQ	Add quick
ADDX	Add with extend
AND	Logical AND
ASL	Arithmetic Shift Left
ASR	Arithmetic Shift Right
Bcc	Branch
BCHG	Bit Test and Change
BCLR	Bit Test and Clear
BRA	Branch Allways
BSET	Bit Test and Set
BSR	Branch To Subroutine
BSET	Bit Test
CHK	Check Register Against Bounds
CLR	Clear Operand
CMP	Compare
CMPM	Compare memory
DBcc	Test Condition, Decrement and Branch
DIVS	Signed Divide
DIVU	Unsigned Divide
EOR	Exclusive OR
EXG	Exchange Registers
EXT	Sign Extend
JMP	Jump
JSR	Jump to Subroutine
LEA	Load Effective Address
LINK	Link Stack
LSL	Logical Shift Left
LSR	Logical Shift Right
MOVE	Move
MOVEM	Move Multiple Registers

Mnemonic	Description
MOVEP	Move Peripheral Data
MOVEQ	Move Quick
MULS	Signed Multiply
MULU	Unsigned Multiply
NBCD	Negate Binary Coded Decimal with extend
NEG	Negate
NEGX	Negate with extend
NOP	No Operation
NOT	One's Complement
OR	Logical OR
PEA	Push Effective Address
RESET	Reset External Devices
ROL	Rotate Left Without Extend
ROR	Rotate Right Without Extend
ROXL	Rotate Left With Extend
ROXR	Rotate Right With Extend
RTE	Return From Exception
RTR	Return and Restore
RTS	Return from Subroutine
SBCD	Subtract binary coded decimal with extend
Scc	Set Conditional
STOP	Stop
SUB	Subtract
SUBQ	Subtract quick
SUBX	Subtract with extend
SWAP	Swap data register halves
TAS	Test and Set Operation
TRAP	Trap
TRAPV	Trap on overflow
TST	Test
UNLK	Unlink

The above commands are only the 56 general types of commands that are available on the 68000.

In the above list some command types are written in different variations like »ADD«, »ADDQ« and »ADDX«. Actually there are more variations than these, but they are ignored by the assembler because it is easier for you not to think about what type of add you are using when they do exactly the same.

An example of commands that are treated the same by the assembler:

Add	**normal**
Add	**to address register**
Add	**Integer**

All the above commands are just treated like a normal add. And the normal ADD can also do all the addressing modes of the ADDA and ADDI instruction. This is most convenient for the programmer and it makes no problem at all. So you have:

```
ADDA.W A0,A1
```

is exactly the same as:

```
ADD.W A0,A1
```

You can write both, but it would be silly to force the programmer first to tell that this is an add to a address register and then write the names of two address registers as operands.

The only thing to remember when you add to an address register is that it is only possible to add word and long size to it and that the word value is sign-extended to long before added, so

```
ADD.W #1000,A0
ADD.L #1000,A0
```

performs the same action, the word instruction is just shorter and faster.

10.1.1 Data Movement Instructions

The following instructions are used to transfer data between memory, data- or address-registers.

The fundamental instruction in this category is the »**move**« instruction. It can transfer byte, word or long data size. This can be done between two memory locations, register-memory or register-register.

Some special move instructions allow you to set the »Condition Code« register (move <ea>,CCR) and also let you read the »SR« register. (move SR,<ea>).

WARNING: This action is privileged on the MC68010 and later chips.

Mnemonic	Syntax		Size
EXG	EXG	Rx,Rx	L
LEA	LEA	<ea>,An	L
MOVE	MOVE	<ea>,<ea>	B W L
	MOVE	<ea>,CCR	W
	MOVE	<ea>,SR	W
	MOVE	SR,<ea>	W
	MOVE	USP,An	L
	MOVE	An,USP	L
MOVEM	MOVEM	<list>,<ea>	W L
MOVEM	MOVEM	<ea>,<list>	W L

10.1.2 Integer Arithmetic Instructions

The MC68000 can add, subtract, multiply, divide and compare two operands. It can also clear, test, sign-extend and negate a single operand.

Some of these commands might demand a little explanation.

Let's say you have two 64 bit numbers that you like to add. To do this the »ADDX« comes in handy because the following can be done.

Let's say the first number is in »D0« and »D1«, and the last number is in »D2« and »D3«. We want the result in »D2« and »D3«. The lowest number is placed in »D1«. We do like this:

```
ADD.L D1,D3
ADDX.L D0,D2
```

If a carry was set in the first addition, this will be added in the next addition.

Also the »DIVS« and »DIVU« commands should be explained. They divide a 32 bit constant with a 16 bit constant. The result is placed in the low 16 bit of the destination operand. In the upper 16 bit the remainder is stored.

So the »MOD« function from Pascal can be made with only a few instructions. Normally we would have:

```
Result = 10 MOD 7
```

We know this is the value »3« but the computer can also calculate this for us:

```
START   MOVEQ   #10,D0
        MOVEQ   #7,D1
        DIVU    D1,D0
        CLR.W   D0
        SWAP    D0
        RTS     ; The remainder »3« is now
                ; placed in the longword »D0«.
```

Mnemonic	Syntax		Size
ADD	ADD	<ea>,Dn	B W L
	ADD	Dn,<ea>	B W L
ADDA	ADDA	<ea>,An	W L
ADDI	ADDI	#d,<ea>	B W L
ADDQ	ADDQ	#d,<ea>	B W L
ADDX	ADDX	Dy,Dx	B W L
	ADDX	-(Ay),-(Ax)	B W L
CLR	CLR	<ea>	B W L
CMP	CMP	<ea>,Dn	B W L
CMPA	CMPA	<ea>,An	W L
CMPI	CMPI	#d,<ea>	B W L
CMPM	CMPM	(Ay)+,(Ax)+	B W L
DIVS	DIVS	<ea>,Dn	W
DIVU	DIVU	<ea>,Dn	W
EXT	EXT	Dn	W L
MULS	MULS	<ea>,Dn	W
MULU	MULU	<ea>,Dn	W
NEG	NEG	<ea>	B W L
NEGX	NEGX	<ea>	B W L
SUB	SUB	<ea>,Dn	B W L
	SUB	Dn,<ea>	B W L
SUBA	SUBA	<ea>,An	W L
SUBI	SUBI	#d,<ea>	B W L
SUBQ	SUBQ	#d,<ea>	B W L
SUBX	SUBX	Dy,Dx	B W L
	SUBX	-(Ay),-(Ax)	B W L
TAS	TAS	<ea>	B
TST	TST	<ea>	B W L

10.1.3 Logical Instructions

One common thing about all logical instructions is that they cannot use address-registers as operands. And they cannot do memory-to-memory operations. But they are still very powerful and useful. If we think back at the »MOD emulation program« from last chapter:

```
Result = 10 MOD 7 = 3
```

If the question had been:

```
Result = 10 MOD 8 = ?
```

this could have been done with logical operations, here we go:

```
START   MOVEQ    #10,D0      ; number 1
        MOVEQ    #8,D1       ; number 2
        SUBQ.L   #1,D1       ; minus 1
        AND.L    D1,D0
        RTS
```

As long as we have a value of 2,4,8,16... we can use logical operations because these values are exactly the maximum combinations in 1,2,3,4.. bits.

Mnemonic	Syntax	Size
AND	AND <ea>,Dn	B W L
	AND Dn,<ea>	B W L
ANDI	ANDI #d,<ea>	B W L
	ANDI #d,SR	B W
EOR	EOR Dn,<ea>	B W L
EORI	EORI #d,<ea>	B W L
	EORI #d,SR	B W
NOT	NOT <ea>	B W L
OR	OR <ea>,Dn	B W L
	OR Dn,<ea>	B W L
ORI	ORI #d,<ea>	B W L
	ORI #d,SR	B W

10.1.4 Shift and Rotate Instructions

The shift and rotate instructions are also very powerful. With these instructions it is possible to get one bit from a 32 bit number at the time. You can also divide a number by 2,4,8,16 etc., because every time a number is shifted one to the right it is divided by 2.

Division:

```
DIVU    #8,D0    =    LSR.L    #3,D0
DIVS    #8,D0    =    ASR.L    #3,D0
```

And where »DIVU« only is 16 bit »LSR.L« is 32 bit.

The shift commands just shifts all bits in one direction and the spare bit is put into both the »C« and »X« flag:

```
     abcdefgh
=    nabcdefg      C = h X = h
```

In »LSR«, »LSL« and »ASL« n = 0, in »ASR« n = a.

The rotate commands shifts like the shift commands, but the spare bit is put into the the other end. So if you rotate a byte 8 bit left, you have the same byte again:

```
     abcdefgh
=    habcdefg      C = h
```

In »ROXL« and »ROXR« it is a little different, because here the spare bit is put into the »X« flag in the old content of the »X« flag is shifted in. So you would have to »ROXR« a byte 9 times to get the same byte again:

```
     abcdefgh
=    Xabcdefg      C = h X = h
```

After next rotation:

```
          Xabcdefg
     =    hXabcdef      C = g X = g
```

It is possible with rotate instructions to make a general 32 bit division routine:

I expect you to put the two operands into »d0« and »d1«, this will result in d1 = d1/d0.

```
DIVIDE32:
         MOVEQ     #32,D3
         MOVEQ     #0,D2
.LOOP    SUB.L     D0,D2
         BCC.B     .OK
         ADD.L     D0,D2
.OK      ROXL.L    #1,D1
         ROXL.L    #1,D2

         DBF       D3,.LOOP
         NOT.L     D1
         RTS
```

So you see, it can become very short.

Mnemonic	Syntax	Size
ASL	ASL Dx,Dy	B W L
	ASL #d,Dn	B W L
	ASL <ea>	W
ASR	ASR Dx,Dy	B W L
	ASR #d,Dn	B W L
	ASR <ea>	W
LSL	LSL Dx,Dy	B W L
	LSL #d,Dn	B W L
	LSL <ea>	W

Mnemonic	Syntax	Size
LSR	LSR Dx,Dy	B W L
	LSR #d,Dn	B W L
	LSR <ea>	W
ROL	ROL Dx,Dy	B W L
	ROL #d,Dn	B W L
	ROL <ea>	W
ROR	ROR Dx,Dy	B W L
	ROR #d,Dn	B W L
	ROR <ea>	W
ROXL	ROXL Dx,Dy	B W L
	ROXL #d,Dn	B W L
	ROXL <ea>	W
ROXR	ROXR Dx,Dy	B W L
	ROXR #d,Dn	B W L
	ROXR <ea>	W

If you shift 1 or 2 right it is faster to make an addition instead:
You want: D0 = D0*2

Normally:

```
LSL.W #1,D0       ; takes 8 cycles

ADD.W D0,D0       ; takes 4 cycles
```

You want: D0 = D0*4

Normally:

```
LSL.W #2,D0       ; takes 10 cycles

ADD.W D0,D0
ADD.W D0,D0       ; takes 8 cycles
```

But if you use longword only D0 = D0*2 is faster with addition.

10.1.5 Bit Manipulation Instructions

The bit manipulation instructions are used to set, clear or change a specified bit in memory or a data register.

In your programs you could perhaps use »D7« as a global status register that tells what things are on and what are off.

```
CursorAn = 0
        BSET    #CursorAn,D7  ; The Cursor is on
```

One smart thing about the bit instructions is that they test the state of the bit before altering it.

So the following is possible:

```
        BCLR    #CursorOn,D7    ; Clear cursor
        BNE     TheCursorWasSet
```

Mnemonic	Syntax	Size
BTST	BTST Dn,<ea>	B L
	BTST #d,<ea>	B L
BSET	BSET Dn,<ea>	B L
	BSET #d,<ea>	B L
BCLR	BCLR Dn,<ea>	B L
	BCLR #d,<ea>	B L
BCHG	BCHG Dn,<ea>	B L
	BCHG #d,<ea>	B L

The long format is to registers only.

10.1.6 Binary Coded Decimal Instructions

The »binary coded decimal« is a special format that makes it easier for you to convert a number in memory to a decimal number on screen.

Normally a byte can carry the value 0 - 255, but in binary coded decimal it can only carry 0 - 99. If we look at the two nibbles (4 bits) in a byte we only use the first 10 combinations in each nibble.

Binary	Binary coded decimal
0000 0000	0
0000 0001	1
0000 0010	2
0000 0011	3
0000 0100	4
0000 0101	5
0000 0110	6
0000 0111	7
0000 1000	8
0000 1001	9
0001 0000	10
0001 0001	11

So if you want to convert a binary coded decimal byte to a string you can use the following program:

```
BCDToStr:
         MOVEQ    #0,D0        ; Zero D0
         MOVE.B   BinByte,D0     ; Put byte into
                                 ; D0 = $00xy
         LSL.W    #4,D0        ; Shift 4 left
                               ; D0 = $0xy0
         LSR.B    #4,D0         ; Shift 4 right
                                ; D0 = $0x0y
         ADD.W    #'00',D0      ; Add ASCII '00'
                                ; D0 = $3x3y
         MOVE.W   D0,String     ; Store in memory
         RTS
BinByte  DC.B     4<<4+6
         EVEN
String   DC.W     0
```

If you have e.g. two 8 digit BCD numbers in memory and you wanted to add them together, the following program would do the trick.

```
AddBCD:
        LEA.L     BCD1+3(PC),A0   ; Get address of
                                  ; Number 1
        LEA.L     BCD2+3(PC),A1   ; Get address of
                                  ; Number 2
        MOVE.W    #4,CCR          ; Clear X-Flag
        ABCD      -(A0),-(A1)
        ABCD      -(A0),-(A1)
        ABCD      -(A0),-(A1)
        ABCD      -(A0),-(A1)
        RTS
BCD1    DC.L      $12345678
BCD2    DC.L      $87654321
```

Mnemonic	Syntax	Size
ABCD	ABCD Dx,Dy	B
	ABCD -(Ay),-(Ax)	B
SBCD	SBCD Dx,Dy	B
	SBCD -(Ay),-(Ax)	B
NBCD	NBCD <ea>	B

10.1.7 Program Control Instructions

Program control instructions are very helpful, and some of them have already been used in earlier examples.

The instructions most used are the conditional instructions. These are the instructions ending with »cc«. Instead of »cc« you write a conditional name: EQual, NEqual etc. You can see all these names at the beginning of this chapter. When you use these you do like this:

```
SUBQ.W   #1,D0
BNE      NotFinished

CMP.W    D0,D1
BHI      Higher     ; Where D1 > D0 ?
```

Conditional Instructions:

Mnemonic	Syntax	Größe
Bcc	Bcc <Label>	B W (S L)
DBcc	DBcc Dn,<Label>	W
Scc	Scc <ea>	B

Unconditional Instructions:

Mnemonic	Syntax	Größe
BRA	BRA <Label>	B W (S L)
BSR	BSR <Label>	B W (S L)
JMP	JMP < ea>	L
JSR	JSR <ea>	L

Return Instructions:

Mnemonic	Syntax	Aktion
RTR	RTR MOVE.W (SP)+,CCR	MOVE.L (SP)+,PC
RTS	RTS MOVE.L (SP)+,PC	

10.1.8 Link and Unlink Instructions

These instructions are most used in high level language compilers link C and Pascal. They are used by subroutines to allocate space on the stack.

It is also possible to use them in your program:

```
START   MOVEM.L D0/D1,-(sp)    ; Parameter 1 + 2
        BSR.L   DIVIDE
        RTS
DIVIDE  MOVEM.L D2/D3/A0,-(sp)
        LINK    A0,#16
        MOVEM.L (sp)+,D0/D1
        BSR.L   DIVIDE32     ; An earlier made
                             ; routine
        UNLK    A0
        MOVEM.L (sp)+,D2/D3/A0
        RTS
```

This example might look a little messy. But in this way it is possible to put parameters to a subroutine on the stack.

Mnemonic	Syntax	Aktion
LINK	LINK An,#d	MOVE.L An,-(SP)
		MOVEA.L SP,An
		ADD.W #d,SP
UNLK	UNLK An	MOVE.L An,SP
		MOVE.L (sp)+,An

10.1.9 System Control Instructions

One of the instructions in this category that deserves most explanation is the »STOP« command. It is privileged because it stores an immediate value into the status register. After this it stops all execution and fetching of instructions. The execution is first started when a sufficiently high interrupt occurs. This instruction can be used to save valuable bus cycles.

An other very important thing to keep in mind is that the »MOVE SR,<ea>« instruction is a privileged instruction on all later chips. So if you are making a program that you want to work on these later chips, don´t use this instruction in user mode.

Privileged Instructions:

Mnemonic	Syntax	Aktion
`RESET`	`RESET`	

`RTE`	`RTE`	`MOVE.W (SP)+,SR`
		`MOVE.L (SP)+,PC`

`STOP`	`STOP #d`	`MOVE.W #d,SR`
		`und Stop`

`ANDI`	`ANDI #d,SR`	
`EORI`	`EORI #d,SR`	
`ORI`	`ORI #d,SR`	
`MOVE`	`MOVE <ea>,SR`	
	`MOVE USP,An`	
	`MOVE An,USP`	

Trap-Generating Instructions:

Mnemonic	Syntax	Aktion
`TRAP`	`TRAP #<vector>`	
`TRAPV`	`TRAPV`	`If V = 1 then trap`
`CHK`	`CHK <ea>,Dn`	

Status Register Instructions:

Mnemonic	Syntax	Aktion
`ANDI`	`ANDI.B #d,CCR`	
`EORI`	`EORI.B #d,CCR`	
`ORI`	`ORI.B #d,CCR`	
`MOVE`	`MOVE <ea>,SR`	
	`MOVE SR,<ea>`	`Privileged on MC68010 and higher`

10.2 Exceptions

Normally when we want some special routine to be executed, we just execute it as a subroutine. If something special happens, the processor can also execute some special routines, these are called exceptions. An exception routine is ended with a »RTE« instead of a »RTS« in a subroutine.

The possible exceptions are:

Vector	Adress	Assignment
0	$000	RESET: Initial SSP
-	$004	RESET: Initial PC
2	$008	Bus Error
3	$00c	Address Error
4	$010	Illegal Instruction
5	$014	Zero Divide
6	$018	CHK Instruction
7	$01c	TRAPV Instruction
8	$020	Privilege Violation'
9	$024	Trace
10	$028	LINE_A Emulator
11	$02c	LINE_F Emulator
12 - 14	$030 - $03b	Reserved
15	$03c	Uninitialized Interrupt
16 - 23	$040 - $05f	Reserved
24	$060	Spurious Interrupt
25	$064	Level 1 Interrupt Autovector
26	$068	Level 2 Interrupt Autovector
27	$06c	Level 3 Interrupt Autovector
28	$070	Level 4 Interrupt Autovector
29	$074	Level 5 Interrupt Autovector
30	$078	Level 6 Interrupt Autovector
31	$07c	Level 7 Interrupt Autovector
32 - 47	$080 - $0bf	TRAP instruction vectors
48 - 63	$0c0 - $0ff	Reserved
64 - 255	$100 - $3ff	User Interrupt

Given below is a short explanation to each individual exception.

RESET: Initial SSP

When a reset is made the initial stack pointer is fetched from here. If you look in address $0000 and $0004 you will not find the »SSP« and »PC«. This is because that the Amiga has a special boot ROM that is put down here when a reset is made.

RESET: Initial PC

When a reset is made the initial program pointer is fetched from here. If you look in address $0000 and $0004 you will not find the »SSP« and »PC«. This is because that the Amiga has a special boot ROM that is put down here when a reset is made.

Bus Error

This exception is evoked if a reserved or not existing memory area is accessed.

Address Error

If an odd address is accessed this error is evoked.

Illegal Instruction

An illegal instruction is a 16-bit binary pattern that does not represent one of the legal opcodes in the 68000 instruction set. One opcode is reserved as illegal ($4afc). In assembly language this is called »ILLEGAL«.

Zero Divide

If a division by zero occurs (eg. DIVS #0,d0) this exception is evoked.

CHK Instruction

The »CHK« instruction can cause an exception through this vector. The »CHK« instruction checks data register boundaries. On overflow the exception is generated.

TRAPV Instruction

The »TRAPV« instruction can cause an exception through this vector. The »TRAPV« instruction will cause this exception if the »V« (overflow) bit is set in the »CCR« registers.

Privilege Violation

If you activate a privileged instruction in user mode this exception is executed. Some privileged instructions are »RESET«, »MOVE.W #0«, »SR«.

Trace

If the trace mode is enabled, an exception will be generated through this vector each time a command has been executed.

LINE_A Emulator

Not all possible opcodes are used on the MC68000 processor, some are reserved for later use. Especially all commands starting with the bit combination »%1010« or »%1111«. All commands starting with »%1010 = $A« are called »LINE_A« commands. You can use these commands to make your own commands or to emulate floating point commands.

LINE_F Emulator

Not all possible opcodes are used on the MC68000 processor, some are reserved for later use. Especially all commands starting with the bit combination »%1010« or »%1111«. All commands starting with »%1111 = $F« are called »LINE_F« commands. You can use these commands to make your own commands or to emulate floating point commands.

11 Programming By Examples

In this chapter we will look at different programming technics, and at different ways to solve several programming problems.

On the disk are some examples in the »Examples«-directory. These are as follows:

GettingStarted.S

This is a source based on the SystemStartup routine. It makes a fractal picture just to show off.

SystemStartUp.S

If you are making a system program there are some basic routines you will always need. Most of these are placed in this source.

BinaryConv.S

Convert a register value into a binary ASCII string. A small example recommended as a first test of the debugger.

BootBlock.S

Create your own bootblock, with your own text.

NonSystemStartUp.S

Whenever you are making a hardware based program or routine, this general »startup routine« will come in handy.

ScrollExample.S

This is a small scroll text routine based on the »Non-System startup« routine.

LineDraw.S

Like the »ScrollExample« this small routine is based on the »Non-System startup« routine.

WindowExample.S

This is an example using include files. It opens a window, reads a text string from the keyboard and prints it again.

Directory.S

This is based on include files like the window example. It opens a window and prints the directory into this window.

The above examples are self-explaining and are put on the disk because they were to large to print. You can use these sources in your own programs, but I think that you should only try to understand them and use what you have learned to make your own programs.

This is also the intention of the programs printed in this manual. It is of course possible to optimize many of these programs, but I think that it was more important to make several different programs solving different problems, than making a few highly optimized ones.

The following programming examples can be divided into three different categories:

General examples: Programs that can be used on any MC68000 computer

System programming: Programs using the Amiga operating system

Hardware programming: Programs using the Amiga hardware directly

11.1 General Examples

In the following examples we will discuss different ways of solving a few simple problems. The execution time of each routine will also be calculated so that it is possible to estimate wich routine is the fastest. If you are a total newcomer to machine code, skip this sub-chapter and return to it later when you have acheived the necessary skills.

11.1.1 Longword multiply

The MC68000 is only capable of multiplying two 16bit numbers, but in some applications it is necessary to multiply 32bit numbers. So here is first some theory:

We look at two 2 digit numbers, each digit is shown with a letter:

```
xy * mn
```

If we want to multiply these two numbers but only know how to multiply one digit numbers we can do as follows:

```
    | x | y |                 |  y*n  |
  * | m | n |   =         |  x*n  |       * 10
                          |  y*m  |       * 10
                      |  x*m  |           * 100
                        ---------------
                = | a | b | c | d |
```

So we see that we get a four digit number. Instead of looking at digits we look at 16bit numbers. The following programs multiply 32bit numbers with the help of four 16bit multiplications:

We have:

	HiWord	LoWord
D0 =	x	y
D1 =	m	n

The result will be placed in D0 (high 32bit) and D1 (low 32bit).

```
MULU32  MOVEM.L   D2/D3/D4,-(A7)       ; Save used
                                       ; registers
        MOVE.W    D1,D2       ; d2 = n
        MOVE.L    D1,D3       ; d1 = n
        SWAP      D3          ; d3 = m
        MOVE.W    D3,D4       ; d4 = m

        MULU      D0,D1       ; d1 = y*n
        MULU      D0,D3       ; d3 = y*m
        SWAP      D0
        MULU      D0,D2       ; d2 = x*n
        MULU      D4,D0       ; d0 = x*m

        SWAP D1
        ADD.W     D2,D1       ; d1 = d1 + d2<<16
        CLR.W     D2
        SWAP      D2
        ADDX.L    D2,D0       ; d0 = d0 + d2>>16
                              ; + Carry
        ADD.W     D3,D1
        CLR.W     D3
        SWAP      D3
        ADDX.L    D3,D0       ; d0 = d0 + d2>>16
                              ; + Carry
        SWAP      D1          ; D1 in correct order
                              ; again

        MOVEM.L   (A7)+,D2/D3/D4
        RTS
```

This was the 32bit unsigned multiply using only 5 registers. You can also make a signed multiply but I will leave this as an exercise. For fun I have calculated the maximum execution time, it was 432 cycles. But if you use it as a macro (without the »RTS«) and allowing »d2«, »d3« and »d4« to be altered, it takes a maximum of 348 cycles. It is a maximum because »MULU« takes a maximum of 70 cycles.

It is also possible with this principle to write a program that multiplies bigger numbers than 32bit, but numbers with that precision are very rarely used. Normally you would use floating point instead.

11.1.2 Longword Division

Now when we are talking about 32bit arithmetics I will repeat the division routine from the chapter about the Motorola instruction set. But this time just in a slightly optimized version.

The calculation is as follows: D1 = D1/D0

```
DIVIDE32b:
        MOVEQ     #0,D2

        REPT      32
         SUB.L    D0,D2
         BCC.B    *+2            ; Jump to current
                                 ; adress +2
         ADD.L    D0,D2
         ADDX.L   D1,D1
         ADDX.L   D2,D2
        ENDR
        NOT.L     D1
        RTS
```

This routine will take a maximum of 1288 cycles (a minimum of 1096).

11.1.3 Delete, Copy and Compare

After implementing these simple mathematical routines, a different area of exercise would be nice. Another important field is deleting, copying and comparing data lists. First we look at delete.

Before you choose the method of deleting you must think carefully about the problem. What part of the problem is always constant? Let´s say you want to delete the screen memory. You could use the blitter, but what if the blitter was busy working with something else?

The first try might be to make a loop like this:

```
CLEAR1  LEA.L   SCREEN,A0
        MOVE.W  #80*200-1,D0        ; Bytes - 1
.LOOP   CLR.B   (A0)+
        DBF     D0,.LOOP
```

This could be improved dramatically if you have deleted an entire longword at the time with »MOVE.L D1,(A0)+«. Do not use »CLR.L (A0)+« because the »clr« instruction performs a read cycle before each write.

You could perhaps repeat the instruction many times so that you could save the clock periods of the »DBF« instruction like this:

```
CLEAR2  MOVEQ    #0,D0
        LEA.L    SCREEN,A0
        REPT     80*200/4
         MOVE.L  D0,(A0)+
        ENDR
        RTS
```

This would allow you to delete 2387000 bytes/second. But this program would use 12800 bytes!

It can be done faster if you use the »MOVEM« command and move all blank registers with this command:

```
CLEAR3  MOVEM.L    BLANK,D0-A1
        LEA.L      SCREENEND,A2
        REPT       2*200
         MOVEM.L   D0-A1,-(A2)
        ENDR
        RTS
```

This routine fills 3240000 bytes/sec. and uses 1600 bytes.

This example was about a data field that you knew everything about. Let´s take a string that you know nothing about. Your routine is called with a position and a length, nothing more. Here we could again use the »CLEAR1« routine, but this routine can still become faster. The first approach to this could be to repeat the clear instruction several times. You call this routine with a pointer in »A0« and a length in »D0«:

```
CLEAR1.2
         MOVEQ    #0,D1       ; Fill value= 0
         SUBQ.L   #8,D0       ; Dekrement counter
                              ; by 8
         BMI.B    .NOMORE     ; No more 8 ?

.LOOP    REPT     8
         MOVE.B   D1,(A0)+    ; Clear 8 bytes
         ENDR
         SUBQ.L   #8,D0       ; take 8 more
         BPL.B    .LOOP       ; finished

.NOMORE  ADDQ.L   #7,D0       ; How many left?
         BMI.B    .FINI       ; no one?
.LOOP2   MOVE.B   D1,(A0)+    ; clear the few left
         DBF      D0,.LOOP2     ; Loop
.FINI    RTS
```

This routine is quite fast, but it it can become several times faster if the address is first aligned so that you can use longword instructions instead:

A schematic program is made here below. The program is not coded, this is left as an exercise to you. Note that it is better if you longword-align the address rather that just word aligning it, because on sophisticated processors like MC68020/30/40 the »MOVE.L« is faster accessing a longword-aligned address.

This is because these processors are 32bit, and a longword write to a non-longword-aligned address would take 2 memory write cycles.

```
CLEAR1.3
        Longword align address by clearing 0 to 3 bytes.
        Clear max number of longwords.
        Clear the last 0 to 3 bytes.
        RTS
```

This routine is faster if the area is large, i.e. more than 10-20 bytes. This depends on the way you have implemented it. If implemented right, this routine will be 3 times faster than the above routine and 7 times faster than »CLEAR1«.

The above algorithms can also be used to make memory move and compare. I will only show you how to implement a useful copy routine with the principle used in »CLEAR1.2«.

You call this routine with a pointer in »A0« (source) and in »A1« (destination) and a length in »D0«:

```
COPY1.2
          SUBQ.L   #8,D0
          BMI.B    .NOMORE

.LOOP     REPT     8
          MOVE.B   (A0)+,(A1)+
          ENDR
          SUBQ.L   #8,D0
          BPL.B    .LOOP

.NOMORE   ADDQ.L   #7,D0
          BMI.B    .FINI
.LOOP2    MOVE.B   (A0)+,(A1)+
          DBF      D0,.LOOP2
.FINI     RTS
```

11.1.4 Sorting

Sorting is a very important subject and there exist many different algorithms to do this. One of the very easy and very well known is the algorithm »Bubble Sort«. This is as follows:

We look at the following five numbers:

3 4 2 5 1

Bubble sort looks at two numbers and swaps them if the first was bigger. After the look at the first it looks at the next pair etc. If after looking at all pairs no one was swapped, the numbers are sorted. Like this:

3 2 4 1 5	Changes: 4-2 5-1
2 3 1 4 5	Changes: 3-2 4-1
2 1 3 4 5	Changes: 3-1
1 2 3 4 5	Changes: 2-1
1 2 3 4 5	Changes: none

You see that it took 5 runs to sort 5 numbers this is »5*5=25« compares. We say that the time complexity for this routine is »O(n*n)«. This means that if there are »n« elements it will take n*n processes to sort the »n« numbers. So if one process takes 72 cycles (100,000 processes/second) it will take 10 seconds to sort 1000 elements.

The bubble sort routine can be implemented in another way that is more efficient.

The first we see about the routine is that the following two lists with only one wrong placed number are sorted in totally different times:

5 1 2 3 4	Sorted in 2 runs
2 3 4 5 1	Sorted in 5 runs

A little simple change can be made to bubble sort to improve this fact. If we first run »left-right« and second »right-left« and repeat this sequence until no elements are swapped the execution time will be shorter:

5 1 2 3 4	Sorted in 2 runs
2 3 4 5 1	Sorted in 3 runs

We can try this new method with the first data row »3 4 2 5 1«

3 2 4 1 5	Changes: 4-2 5-1
1 3 2 4 5	Changes: 4-1 2-1 3-1
1 2 3 4 5	Changes: 3-2
1 2 3 4 5	Changes: NONE

Now we only used 4 runs instead of 5 before.

If we wanted to be more advanced we could remember the first position where we had to change the first element, and with next run we will only have to start at the position »one« before this place. This would prevent us from comparing elements that are already sorted.

We could check even more things, but we have to think about, that if the routine has to check a lot of things, it is not sure that it becomes faster just because it only has to check half the elements. The Bubble Sort implementation I usually use is the following:

Check the two elements. If they are swapped went one step back if not one step forward. The list is sorted when the end is reached. (Simple !) My results with this routine are very good: Usually 4-5 times faster than the implementation I first told you about. Coded in 68000 code it looks like this:

```
BUBBLE_A_LA_RUNE:
         LEA      ListStart,A0  ; A0 = Start ptr
         LEA      ListEnd,A1    ; A1 = End ptr
         MOVEQ    #ElemLen,D1   ; D1 = Element
                                ; length
         MOVE.L   A0,A2       ; A2 = Actual element
                              ; pointer
         MOVE.L   A0,A3
         ADD.L    D1,A3       ; A3 = Next element
                              ; pointer
.LOOP    MOVE.B   (A3),D0
         CMP.B    (A2),D0     ; First =< Next
         BLO.B    .CHANGE     ; No change
         ADD.L    D1,A2
         ADD.L    D1,A3
         CMP.L    A3,A1       ; End reached?
         BNE.B    .LOOP       ; No loop
         RTS

.CHANGE  MOVE.L   D1,D2       ; Swap elements
.LOOP2   MOVE.B   (A2),D0
         MOVE.B   (A3),(A2)+
         MOVE.B   D0,(A3)+
         SUBQ.L   #1,D2
         BNE.B    .LOOP2
```

```
        SUB.L   D1,A2
        SUB.L   D1,A3
        CMP.L   A2,A0       ; At first element?
        BEQ.B   .LOOP       ; Yes
        SUB.L   D1,A2       ; One element forward
        SUB.L   D1,A3       ; One element back
        BRA.B   .LOOP       ; Go again
```

This implementation is very general. If you only want to sort elements of a fixed length and these elements have an even lenght you can make a fixed swap routine using longwords. But the above routine will work of course. Bubble sort is best when you have small data areas or bigger areas if they are almost sorted.

11.2 System Programming

The Amiga system has a lot of possibilities, and it is far to komplex to describe it in detail. This chapter will just give a few simple examples of what you can do with the system. For more details about the system consult the »ROM kernel manuals« from Commodore.

The system is based on different libraries. Most of them are placed in the ROM, but some of them are also placed in the »libs« directory on the disk. Before you can use a library you have to load it. The most common libraries are as follows:

exec.library

This is the basic library. With this library you can open other libraries, allocate memory, create tasks etc.

dos.library

Contains basic input/output routines like disk and screen communication.

intuition.library

Handles all creation of screens, windows, menus etc.

graphics.library

Contains routines to draw lines, circles, to fill scroll, delete etc.

layers.library

Used to handle modification of the screen memory.

diskfont.library

This library is kept on the disk. It contains routines to handle new fonts for use in your programs.

icon.library

A library which can create and modify icons on the workbench screen.

translator.library

Is used together with the speech synthesizer. It translate the written word into phonetic symbols.

mathffp.library

Basic floating point calculation routines.

mathieedoubbas.library

Basic integer calculation routines.

mathtrans.library

Advanced floating point calculation routines, like SIN, COS, LOG, EXP etc.

There are several other libraries. But the above are the most common ones. Normally, when you use these libraries you use the include files placed on the disk. The include files are like normal source codes, but they contain only constant definitions and macro difinitions.

Before you ever use any include file, you will have to set the path of where to get the include files from. This is done with the following command:

```
INCDIR df0:include/
```

This is where you normally will get the include files from. Now we can include the files that we are going to use in the next examples:

```
include   exec/exec_lib.i
include   libraries/dos_lib.i
include   libraries/dos.i
include   intuition/intuition_lib.i
include   intuition/intuition.i
```

To open a library you issue the following commands. We want to open »dos.library«:

```
DOS_LIBOPEN
         LEA.L     _DOSNAME(PC),A1   ; A1 =>
                               ; dos.library
         MOVEQ     #0,D0             ; Version 0
         CALLEXEC  OpenLibrary ; Open library
         MOVE.L    D0,_DOSBASE ; _dosbase =>
                               ; library
         BEQ.L     ERROR_ERROR ; ?? No pointer
                               ; returned!
         RTS
```

```
_DOSNAME DOSNAME        ; Dosname is a makro
                        ; containing the string
                        ; 'dos.library',0
_DOSBASE DC.L 0         ; Pointer to the library
```

After calling this program, a pointer to the library is returned. If the library was not found, a »zero« will be returned. If »dos.library« is not found, something is really wrong.

If you had not used include files the routine would look like this:

```
OpenLibrary = -552

DOS_LIBOPEN_NO_INCLUDE:
         LEA.L   _DOSNAME(PC),A1    ; A1 =>
                              ; dos.library
         MOVEQ   #0,D0        ; Version 0
         MOVE.L  $4.W,A6      ; Get pointer to
                              ; exec.library
         JSR     OpenLibrary(A6)    ; Open the
                                    ; library
         MOVE.L  D0,_DOSBASE        ; _dosbase=>
                                    ; library
         BEQ.L   ERROR_ERROR ; ?? No pointer
                              ; returned !
         RTS

_DOSNAME DC.B    'dos.library'      ; Name with
                              ; lower case only!
_DOSBASE DC.L    0            ; Pointer to the
                              ; library
```

The advantage with include files is that you don´t have to remember the library offsets. In this case »OpenLibrary = -552« all the time. But the above program shows more about how the libraries work.

The only library we know the address of is the »exec.library«. This pointer is kept in address »$4«. »Exec« can do a lot of things, what can be found in the include file called: »exec_lib.i«. When even you want to call an exec routine you will have to call »Execbase + Offset«. All offsets are negative, so the address you call is lower than the pointer to the library. Each library offset is decremented with 6 starting from -30. The first few entries in »exec. library« are as follows:

```
Supervisor  = -30
ExitIntr    = -36
Schedule    = -42
```

And very long down in the list we got:

```
OpenLibrary = -552
```

If we looked at the address »ExecBase-552« we would see something like this:

```
$0000044E  JMP  $00005E3A     ; Dont count on
                         , this. Different versions
                         ; of the system changes this
```

So we see that we call just a »jump command« that calls the routine we want to execute.

In the rest of the examples I will use definitions from the include files. If you want to see exactly what is done just disassemble the routines. To see all the include definitions, assemble and type »=S«.

Now, after opening the library, we would have to make a routine that close it again when we dont want to use it any more.

```
DOS_LIBCLOSE
       MOVE.L   _DOSBASE(PC),A1    ; Pointer
                             ; to the library
       CALLEXEC CloseLibrary ; Close it!
       RTS
```

After having opened »dos library« we can easily open a window on the workbench screen.

```
WINDOW_OPEN
        MOVE.L    #_WINDOWNAME,D1      ; Window
                                       ; definitions
        MOVE.L    #MODE_OLDFILE,D2     ; Mode old
        CALLDOS   Open
        MOVE.L    D0,_WINDOWHANDLE     ;The returned
                                  ; Window handle
        BEQ.L     ERROR       ; 0, if the window
                              ; wasn´t opened
        RTS
_WINDOWNAME    DC.B    'CON:10/10/600/100/My window',0
_WINDOWHANDLE  DC.L    0
```

Here is a routine to close the window.

```
WINDOW_CLOSE
        MOVE.L    _WINDOWHANDLE(PC),D1
        CALLDOS   Close
        RTS
```

After you have opened the window please remember that the window will appear behind the ASM-ONE-screen. So use the mouse to drag it down. You can do a lot of things with windows, the routine below can print a text in the window:

```
TEXT_WRITE
        MOVE.L    #TEXT,D2         ; Pointer to text
        MOVE.L    #TEXT_END-TEXT,D3   ; Text lengh
        MOVE.L    _WINDOWHANDLE,D1 ; Wind. handle
        CALLDOS   Write            ; Write the text
        RTS
TEXT    DC.B      'Hello, this is a text'
TEXT_END
```

To see a program that does it all, try to run the example called »WindowExample.S« or »Directory.S«.

11.3 Hardware Programming

As mentioned in the first chapter, one of the reasons for programming in machine language is that you can easily get access to the computer´s custom chips. This chapter will only contain a small introduction on how to access some of the special purpose registers in the Amiga. Because of the limited space and the complex subject not all registers will be explained. For further information you will have to get yourself a copy of the »Amiga Hardware Manual«.

Normally when you code using direct access to the custom chips, you will have to close down the system. This is the only way to be sure that nothing suspicious will happen. Anything can be done using library calls, but some coders like to know exactly what the machine is doing. If you access the chips directly, you are able to make very specialized routines that might run a little faster than the system routines, but then you are not insured that the routine will work on future releases of the machine. On the other hand, even if you are going to code using only system routines, it is a big help to understand exactly how the machine is working.

On the disk is placed a file called »NonSystemStartUp.S«. This file contains routines that close down the system enabling, you to start your own routines in a totally clean environment. After pressing the left mouse button, it reopens the system into its old state.

Below is a list of all registers placed as offsets from the base address »$DFF000«. If you want to access »BplCon0« you would have to access address »$dff100«.

NAME	ADD	R/W	Function
BLTDDAT	$000	ER	Blitter destination early read
DMACONR	$002	R	DMA control read
VPOSR	$004	R	Read Vert Most sig. bit
VHPOSR	$006	R	Read vert and horiz pos of beam
DSKDATR	$008	ER	Disk data early read
JOY0DAT	$00A	R	Joy-Mouse 0 data
JOY1DAT	$00C	R	Joy-Mouse 1 data
CLXDAT	$00E	R	Collision data reg (read & clear)
ADKCONR	$010	R	Audio, disk control read
POT0DAT	$012	R	Pot counter pair 0 data
POT1DAT	$014	R	Pot coubter pair 1 data
POTINP	$016	R	Pot pin data read
SERDATR	$018	R	Serial port data and status read
DSKBYTR	$01A	R	Disk data byte and status read
INTENAR	$01C	R	Interrupt enable bits read
INTREQR	$01E	R	Interrupt request bits read
DSKPT	$020	W	Disk pointer (longword)
DSKLEN	$024	W	Disk length
DSKDAT	$026	W	Disk DMA data write
REFPTR	$028	W	Refresh pointer
VPOSW	$02A	W	Write vert most sig bit
VHPOSW	$02C	W	Write vert and horiz position
COPCON	$02E	W	Coprocessor control register
SERDAT	$030	W	Serial Port data and stop bits write
SERPER	$032	W	Serial port period and control
POTGO	$034	W	Pot count start
JOYTEST	$036	W	Write to all 4 mouse counters at once
STREQU	$038	S	Strobe for horiz sync (VB & equ)
STRVBL	$03A	S	Strobe for horiz sync (VB)
STRHOR	$03C	S	Strobe for horiz sync
STRLONG	$03E	S	Strobe for long horiz. line
BLTCON0	$040	W	Blitter control register 0
BLTCON1	$042	W	Blitter control register 1
BLTAFWM	$044	W	Blitter first word mask for source A
BLTALWM	$046	W	Blitter last word mask for source A
BLTCPT	$048	W	Blitter source C pointer (longword)

NAME	ADD	R/W	Function
BLTBPT	$04C	W	Blitter source B pointer (longword)
BLTAPT	$050	W	Blitter source A pointer (longword)
BLTDPT	$054	W	Blitter dest. D pointer (longword)
BLTSIZE	$058	W	Blitter start and size (width,higth)
BLTCMOD	$060	W	Blitter source C modulo
BLTBMOD	$062	W	Blitter source B modulo
BLTAMOD	$064	W	Blitter source A modulo
BLTDMOD	$066	W	Blitter dest. D modulo
BLTCDAT	$070	W	Blitter source C data
BLTBDAT	$072	W	Blitter source B data
BLTADAT	$074	W	Blitter source A data
DSKSYNC	$07E	W	Disk sync pattern register
COP1LC	$080	W	Copper pointer 1 (longword)
COP2LC	$084	W	Copper pointer 2 (longword)
COPJMP1	$088	W	Restart at pointer 1
COPJMP2	$08A	W	Restart at pointer 2
COPINS	$08C	W	Copper inst. fetch identify
DIWSTRT	$08E	W	Display window start (vert-hor)
DIWSTOP	$090	W	Display window start stop (vert-hor)
DDFSTRT	$092	W	Bit plane data fetch start
DDFSTOP	$094	W	Bit plane data fetch stop
DMACON	$096	W	DMA control write
CLXCON	$098	W	Collision control
INTENA	$09A	W	Interrupt enable bits
INTREQ	$09C	W	Interrupt request bits
ADKCON	$09E	W	Audio, disk, UART control
AUD0	$0A0		Audio channel 0 start address
AUD1	$0B0		Audio channel 1 start address
AUD2	$0C0		Audio channel 2 start address
AUD3	$0D0		Audio channel 3 start address
AUDxPTR	+$00	W	Audio channel x pointer
AUDxLEN	+$04	W	Audio channel x length
AUDxPER	+$06	W	Audio channel x period
AUDxVOL	+$08	W	Audio channel x volume
AUDxDAT	+$0A	W	Audio channel x data

NAME	ADD	R/W	Function
BPL1PT	$0E0	W	Bitplane 1 pointer
BPL2PT	$0E4	W	Bitplane 2 pointer
BPL3PT	$0E8	W	Bitplane 3 pointer0
1BPL4PT	$0EC	W	Bitplane 4 pointer
BPL5PT	$0F0	W	Bitplane 5 pointer
BPL6PT	$0F4	W	Bitplane 6 pointer
BPLCON0	$100	W	Bit plane ctrl reg (misc ctrl bits)
BPLCON1	$102	W	Bit plane ctrl reg (scroll value)
BPLCON2	$104	W	Bit plane ctrl reg (priority ctrl)
BPL1MOD	$108	W	Bit plane modulo (odd planes)
BPL2MOD	$10A	W	Bit plane modulo (even planes)
BPL1DAT	$110	W	Bit plane 1 data (parallel to serial)
BPL2DAT	$112	W	Bit plane 2 data (parallel to serial)
BPL3DAT	$114	W	Bit plane 3 data (parallel to serial)
BPL4DAT	$116	W	Bit plane 4 data (parallel to serial)
BPL5DAT	$118	W	Bit plane 5 data (parallel to serial)
BPL6DAT	$11A	W	Bit plane 6 data (parallel to serial)
SPR0PT	$120	W	Sprite 0 pointer
SPR1PT	$124	W	Sprite 1 pointer
SPR2PT	$128	W	Sprite 2 pointer
SPR3PT	$12C	W	Sprite 3 pointer
SPR4PT	$130	W	Sprite 4 pointer
SPR5PT	$134	W	Sprite 5 pointer
SPR6PT	$138	W	Sprite 6 pointer
SPR7PT	$13C	W	Sprite 7 pointer
SPR0	$140		Sprite 0 area
SPR1	$148		Sprite 1 area
SPR2	$150		Sprite 2 area
SPR3	$158		Sprite 3 area
SPR4	$160		Sprite 4 area
SPR5	$168		Sprite 5 area
SPR6	$170		Sprite 6 area
SPR7	$178		Sprite 7 area

NAME	ADD	R/W	Function
SPRxPOS	+$00	W	Sprite x Vert-Horiz position
SPRxCTL	+$02	W	Sprite x stop pos & control
SPRxDATAA	+$04	W	Sprite x image data a
SPRxDATAB	+$08	W	Sprite x image data b
COLOR0	$180	W	Color 0
COLOR1	$182	W	Color 1
bis			
COLOR31	$1BE	W	Color 31
SPECIAL	$1DC	W	NTSC control on Fatter Agnus

The next programming examples can only be understood if you, while reading, also consult a Hardware Reference Manual. See the appendix about recommended literature

Before we really start with non-system programs, I will show you some very small examples of how to get simple reactions from the hardware.

One of the first and most simple things we can do is to make routines what can check the fire buttons on the mouse and joystick.

To do this we have to look at a new register. This is placed at address »$bfe001«, it has the following layout:

Bit	Function
7	Game port 1 (fire button)
6	Game port 0 (fire button)
5	Disk ready
4	Disk track 00
3	Write protect
2	Disk change
1	Led light (0 = filter on)
0	Memory overlay

We see that bit 6 is the left mouse button and bit 7 the joystick button. If the bit is »zero«, the button is pressed:

```
LeftMouseButton:
        BTST    #6,$BFE001
        BEQ     LeftMousePressed
JoyButton:
        BTST    #7,$BFE001
        BEQ     JoyPressed
```

To test the right mouse button we have to check bit 10 in »POTINP« located at address »$dff016«:

```
RightButtonTest:
        BTST    #10,$DFF016
        BEQ     RightMousePressed
```

If you bit-set or test in other registers, remember that »BSET«, »BTST« etc only work on byte size. So »BTST #10,??« is equivalent with »BTST #10-8,??«.

When you are making non-system programs, programs that do not use the library and kickstart routines (most games/demos are programmed this way), I recommend you to use the source on the disk called »NonSystemStartUp.S«. This source provides you with an standard easy to use routine. An example of how to use this startup is shown in the source called »ScrollExample.S«

The startup routine is based on an interrupt. This interrupt (called »vertical blanking«) is activated each time the screen is redrawn on the monitor. This happens 50 times/second (In USA 60 times/second). We say that the computer shows 50 frames/second. (Read in the Hardware Manual about playfield hardware to understand this). With this interrupt you can activate one or more routines.

In the »scroll« example mentioned above, the text on the screen is moved one time each frame. In one second you will see the scroller positioned on 50 different positions each time moved a little to the left. Because the fact that the eye can only perceive about 25 different frames/second, the 50 frames will be recognized as one smooth motion.

This technic can be used with a lot of other things. You could draw a ball at one frame, delete the old ball and draw a new on a little different position on the next frame. In this way you can make a bounching ball or, if you change the shape of the ball with each frame, a little animation.

To work properly, you will of course insert these routines at the correct place in the »NonSystemStartUp.S« source.

First find these lines in the source:

```
INTER:
     MOVEM.L  D0-D7/A0-A6,-(A7)     ; Put
                              ; registers on stack
     LEA.L    $DFF000,A6
     MOVE.L   #SCREEN,$E0(A6)

;--- Place your interrupt routine here ---

     MOVE.W   #$4020,$9C(A6)        ; Clear
                              ; interrupt request
     MOVEM.L  (A7)+,D0-D7/A0-A6     ; Get regis-
                                 ; ters from stack
     RTE
```

Insert the routine after the text »;--- Place y... «.

Drawing objects on the screen takes time. Because the object has to be redrawn 50 times/second to form a smooth motion, the routine must of course not take more than 1/50 second. If a routine takes more time you will in most cases not activate this from the interrupt.

Let´s say you have made a routine that can calculate a picture, but this takes 30 seconds, you would place this routine between the following two lines in the »startup« routine:

```
;***** Your main routines *****
;***** Main loop Test mouse button *****
```

Making programs using the custom chips can only be done if you can both code and know how the hardware works. Because the Amiga hardware is very advanced it is not possible in this short chapter to give a close description of how it works. I advise you to read in one of the books written about this subject. Following here is a example that can draw one object on the screen.

11.3.1 Making an object

To make animations on the Amiga you will find the blitter very handy. It is fast and not so difficult to use. Before we can show you a programming example we must look at the registers to control the blitter.

Add the »ADD« value to the base address »$dff000«:

NAME	ADD	R/W	Function
BLTCON0	$040	W	Blitter control register 0
BLTCON1	$042	W	Blitter control register 1
BLTAFWM	$044	W	Blitter first word mask for source A
BLTALWM	$046	W	Blitter last word mask for source A
BLTCPT	$048	W	Blitter source C pointer (longword)
BLTBPT	$04C	W	Blitter source B pointer (longword)
BLTAPT	$050	W	Blitter source A pointer (longword)
BLTDPT	$054	W	Blitter dest. D pointer (longword)
BLTSIZE	$058	W	Blitter start and size (width,higth)
BLTCMOD	$060	W	Blitter source C modulo
BLTBMOD	$062	W	Blitter source B modulo
BLTAMOD	$064	W	Blitter source A modulo
BLTDMOD	$066	W	Blitter dest. D modulo
BLTCDAT	$070	W	Blitter source C data
BLTBDAT	$072	W	Blitter source B data
BLTADAT	$074	W	Blitter source A data

The bit-pattern for the two control registers is as follows:

Bit	BLTCON0	BLTCON1	Bit	BLTCON0	BLTCON1
15	ASH3	BSH3	07	LF7	x
14	ASH2	BSH2	06	LF6	x
13	ASH1	BSH1	05	LF5	x
12	ASH0	BSH0	04	LF4	EFE
11	USEA	x	03	LF3	IFE
10	USEB	x	02	LF2	FCI
09	USEC	x	01	LF1	DESC
08	USED	x	00	LF0	LINE

ASH0-3	- Source A shift value
BSH0-3	- Source B shift value
USEA-D	- Bit pattern to select source A - D.
LF0-7	- Logic function minterm select
EFE	- Exclusive fill enable
IFE	- Inclusive fill enable
FCI	- Fill carry input
DESC	- Descending bit
LINE	- Line mode (1=On)

The blitter has three sources (A, B, C) and one destination (D). !What the blitter generally does is to take data words from the three sources and join these three words together to one word. The way the three words are joined together is described by the »LF« byte (Bit 0-7 in »BLTCON0«).

Let´s say we have an object (Pattern) and a screen with the width of 40 bytes. We can use source »A« to point at the object data. We use destination »D« to point at the position on the screen where we want the object. If we just source »A« and destination »D« we would perform a block move, but we don't want the object to delete the old pattern on the screen. To prevent this we will have to use source »C«, this source must point at the same position on the screen as the destination. The following three lines will set these three pointers

```
MOVE.L   #OBJECT,$DFF050      ; Source A
MOVE.L   #SCREEN,$DFF048      ; Source C
MOVE.L   #SCREEN,$DFF054    ;Destination D
```

If the object is one word wide (2 bytes), we must set the modulo on source »C« and destination »D« to 40-2 bytes = 38 bytes. The modulo on source »A« must be set to zero. Here we go:

```
MOVE.W   #0,$DFF064    ; Modulo Source A
MOVE.W   #38,$DFF060   ; Modulo Source C
MOVE.W   #38,$DFF066   ; Modulo on Desti-
                       ; nation D
```

Now we need to set the control registers, we start with BLTCON0. The shift value is set to zero (%0000), we have used A, C and D so the next four bits are set to »%1011«. The logical function bytes are set to »%11111010«. When the blitter joins the three source words in to one word, it looks at one bit at the time (the first bit in all three words). Three bits give a maximum number of 8 combinations. (2^3=8 that is why you find 8 »LF« bits). The bits are set according to the list below:

Bit	A	B	C	Desired result in D
7	1	1	1	1
6	1	1	0	1
5	1	0	1	1
4	1	0	0	1
3	0	1	1	1
2	0	1	0	0
1	0	0	1	1
0	0	0	0	0

We don´t use source »B«, so we only have to concern about the four different combinations in A and C (11 = 1, 10 = 1, 01 = 1 and 00 = 0). If you try going, think about these four combinations by saying to yourself: If a bit is set in the object word and a bit is set in the background word, then we want the result to be a set bit in the destination word etc.

The result of the above considerations is that the »BLTCON0« must be set to:

```
MOVE.W #%0000101111111010,$DFF040
```

Control register 1 must be set to zero.

```
MOVE.,W #%0000000000000000,$DFF042
```

To start the blitter you must write the size of the object into the register called »BLTSIZE«. The size is calculated as »64*Hight+Width/2«.

```
MOVE.W #64*8+2/2,$DFF058
```

The above lines can be joined into one little program:

```
MAKE_OBJECT:
        LEA.L   $DFF000,A6      ;Custom base to A6

WAIT:   BTST    #14-8,$2(A6)    ; Is the blitter
                                ; ready?
        BNE.B   WAIT            ;

        MOVE.W  #%0000101111111010,$40(a6)
        MOVE.W  #%0000000000000000,$42(a6)

        MOVE.L  #OBJECT,$50(a6)       ; Source A
        MOVE.L  #SCREEN,$48(a6)       ; Source C
        MOVE.L  #SCREEN,$54(a6)       ; Destina-
                                      ; tion D
        MOVE.W  #0,$64(a6)      ; Modulo source A
        MOVE.W  #38,$60(a6)     ; Modulo source C
        MOVE.W  #38,$66(a6)     ; Modulo destina-
                                ; tion D

        MOVE.L  #$FFFFFFFF,$44(a6)  ; Don´t mask
                                ; anything away
        MOVE.W  #64*8+2/2,$58(a6)   ; Start
                                    ; blitter
        RTS

OBJECT: DC.W    %0000000110000000   ;Objectdates
        DC.W    %0000001111000000
        DC.W    %0100011001100010
        DC.W    %0100111001110010
        DC.W    %0111111111111110
        DC.W    %0100011111100010
        DC.W    %0000001111000000
        DC.W    %0000011111100000
```

To see more examples of how to use the blitter, look in the files called »ScrollExample.S« and »LineExample.S«.

Appendix

A. Preferences

To insure that you get exactly the assembler you want, a lot of options have been supplied. You find these options in the »Project« and the »Assembler« menus.

The options are as follows:

Option	Presetting	Short Name
Rescue	Off	RS
Level 7	Off	L7
NumLock	On	NL
AutoAlloc	On	AA
ReqLibrary	Off	RL
PrinterDump	Off	PD
Interlace	Off	IL
1 Bitplane	Off	1B
Source .S	On	.S
LineNumbers	Off	LN
AutoIndent	On	AI
ShowSource	On	SS
ListFile	Off	LF
Pageing	On	PG
Halt Page	On	HP
All Errors	Off	AE
Debug	Off	DB
Label:	Off	L:
UCase=LCase	On	UL
Offset(A4)	Off	A4
DisAssemble	On	DA
OnlyAscii	Off	OA
Comment	Off	C
Close WB	Off	CW

A close discription af all these options will follow later.

It is possible to save a »preference file« called
»ASM-One.Pref«.
This file will consist of a (+/-) meaning »on/off« followed by the »Short Name« of the option. Like this:

```
-RS-L7+NL+AA+RL etc.
```

After this row of preferences it is possible for you to place some text. This is done in the following way:

```
-RS-L7+NL+AA+RL\c\200\
```

This will allocate 200 kb of workspace in the chipmemory every time the assembler is loaded. Any other command string can also be placed here. The sign »\« stands for <return>.The extension »!« allows to apply a suffix to the filename when saved. See preference »Source.S«.

Close description of the individual options:

Rescue

The »rescue« option will **rescue** any **program fault** or crash in your program even if the system was shut down (interrupts disabled, own copperlist etc). It will survive »division by zero« in a »3. level interrupt«.

WARNING: This action can be dangerous for the Amiga multi-tasking system. But on the other hand, if you make non-system programs and a crash occured you would have to reset the Amiga and restart, now at least you have a chance to save the source.

Level 7

This Level 7 option allows to **break** your **program** at **any** given **time**. Let´s take an infinite loop. These programming bugs can now be stopped and the source saved. To activate the level 7 exception you will have to get a soldering iron and a switch. If you haven´t made a little electronics before look for a person who knows about. If you do it yourself you can damage your computer!

If you got an **Amiga 500,** here is how you do it:

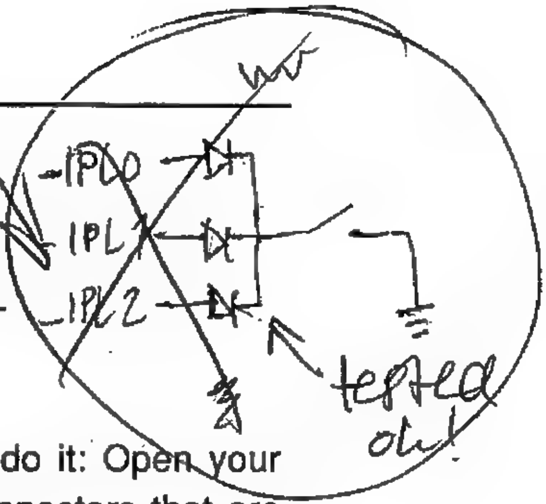
At the left side of the computer is a 86-pin connector behind a little shutter. Remove this and you will see the connector as shown below.

	GND	1	2		
		3	4		
		5	6		
U		7	8		L
P		9	10		O
P		11	12		W
E		13	14		E
R		15	16		R
S					S
I		39	40	_IPL0	I
D		41	42	_IPL1	D
E		43	44	_IPL2	E
		83	84		
		85	86		

You get yourself 3 normal semi-conductors (diodes) and a switch. Connect one diode to each IPL-line (positive end). Connect the 3 other legs to one another and connect these to a switch. Connect the other pin of the switch to GND. See the schematic diagram below.

correct!!

40	_IPL0	~~- Diode +~~		
42	_IPL1	~~- Diode +~~	- / - Switch	(1) GND
44	_IPL2	~~- Diode +~~		



If you got an **Amiga 2000**, here is how you do it: Open your computer. Find the five 100-pin expansion connectors that are placed next to one another. On the print board at one end of the connector is printed the number 1. (At the end that is turned to the machine). The connector will look somewhat like this:

	GND	1	2		
		3	4		
		5	6		
U		7	8		L
P		9	10		O
P		11	12		W
E		13	14		E
R		15	16		R
S					S
I		39	40	E-Int 7	I
D					D
E		97	98		E
		99	100		

If you connect pin 40 with a switch to pin 0 you will have your »level 7 switch«. We hope it is possible to follow these instructions, if you are not totally certain of how to do it, then don´t. Wrong connection can be fatal.

Note: No warranties are given that these connections are right. If they are wrong they might damage the machine or software. If you choose to make these connections the risk is yours alone, not ours.

NumLock

With this option set on the **numeric keyboard** is turned into its alternate functions:

7	Home	8	Up	9	Page up
4	Left			6	Right
1	End	2	Down	3	Page down

AutoAlloc

If you create a »**DATA_C**«-**section** the memory used for this section will **automatically** be **allocated** in chipmemory if »AutoAlloc« is on. Otherwise the normal workspace memory will be used, not concerning if it is chip- or fastmemory.

ReqLibrary

The »Requester library« is a help library that **creates** the **file requesters** and the other small requesters. They are easy to use, but takes up a little memory (14 kb). If you don't like this you can turn it off. If it is turned off, the first time you load the assembler it is not loaded into memory. Otherwise it will just stay unused in the memory.

PrinterDump

This option **activates output** to the **printer**. If set, all text from the command line will be outputted to the printer. This option can also be toggled with »Amiga-P«. If you use the printer for the first time, the printer device will be loaded.

Interlace

Shift **display** to »**interlace**«. If there is not enough memory available for this action, a »one bitplane mid-res window« will be opened instead.

1 Bitplane

Shift **display** to »**one bitplane**«. This will save some memory. But the cursor will become the same color as the text, and therefore can be hard to find.

Source .S

If set, a »**.S**« will be appended to the filename everytime you try to **save**. If you don´t want the ».S«, you can delete it from the **file name** string everytime you save, or disable this option.

The option »!« is availiable in the »preference file«. The syntax is as follows:

Syntax: !.<extension max.3 chars>

e.g.:

```
!.ASM
```

The extension is the text that is appended to the filename when saved. This text is however only appended when the »Source .S«-option is on. If the »Source .S«-option is off, no text is appended to the »SHOW«, »HIDE« gadgets in the requester library.

LineNumbers

If this option is set, each **line** will be shown with a **number** in the editor. Nothing else will be different.

AutoIndent

If on, »**leading spaces/tabs**« will be copied to the **next line** each time you press <return>. This is helpful if you use a »leading tab« on each line.

ShowSource

When not set, you are able to **trace programs** without having the source. Or if you are tracing a program with a lot of macros, it is nice to watch the entire translation of the macro.

ListFile

Each time you assemble, a **list file** will be created if this option is set. The list file will consist of line number, memory address, memory content, assembly line and the symbol table.

Paging

If you want the **list file** divided into **pages**, select this option. See the »ListFile« flag for more details.

Halt Page

If **paging** is turned on, you can choose that each page waits on a keyboard pression. See the »ListFile« flag for more details.

All Errors

Normally when a source is assembled, the assembling is broken each time an error occurs. But if the option is choosen, all **errors** will be **reported** each time you assemble. If you got a printer, our can output the errors to the printer.

Debug

If you are debugging your source frequently with the build-in-debugger, you can set this option on. It takes a little more memory but if on, the **debugger does not** have to **reassemble** the source.

Label:

ASM-ONE is macro assembler compatible. This means that labels don´t have to use colons. A command is distinguished from a label by the fact that a command cannot be placed at the first column. But some assemblers do not allow **labels without colons** but they allow commands to be placed at the first coloumn. If the »label:« option is set, sources like this can still be assembled.

UCase=LCase

Sometimes you might like to **use** the same **name twice**, if you clear this flag this is possible if you just use different capitals in the two names, like:

```
DOSBase   und   DOSBASE
```

They will normally be treated as the one and same name, but if the flag is cleared these two names are different.

DisAssemble

When watching the source in the debugger, it might sometimes be convenient to watch a **disassembled** version of the **next line** because this enables you to check that the source line is translated correct.

OnlyAscii

If you are watching an ASCII-dump and are searching for text strings, you are not interested in **alternate characters**. If »OnlyAscii« is selected these chars are **shown as** a **spot**.

;Comment

Normaly a **comment** is everything after a correctly ended instruction. A instruction is ended with one or more blanks. But if a space by accident was put into the operand field, the part right of this space was in some cases treated like a comment. Example:

```
        MOVE.W   d0,DataPtr +2
```

Here »+2« is treated like a comment. To avoid these nasty errors, turn on the flag »;Comment«. In the preference file use »+;C«.

Close WB

If you set this on, the **workbench screen** will be **closed**. It is however only possible to close the workbench, if no task window is opened on it (for example the CLI window). In the preference file this option is called »CW«.

B. Recommended literature /Books

This manual will not be enough if you really want to program the Amiga. The books that I would recommend are here listed with a short explanation. They can be divided into two categories: Special Amiga books and general MC68000 books:

Hardware Reference Manual (Commodore)

If you want to get a close description of how the hardware works on the Amiga, this is the book to get. It contains some direct code examples, but not so many. But if you can code some MC68000 code in advance, it is not difficult to translate what you read into code.

Libraries and Devices (Commodore)

If you want to use the system on the Amiga, this manual tells you a lot about how the different libraries work. Nearly no matter what you code, a good understanding about how the system works is necessary. You use this book to get a understanding about how to do different things about the system, but this book does not give that many assembler coding examples (most C examples), so you have to look a lot in the include files in the beginning.

Includes and Autodocs (Commodore)

If you have bought »Libraries and Devices«, this is the next book to get. It´s more specific and written in a schematic way. It also explains about the libraries, but gives a closer description of the different library calls. It also tells what arguments go into what registers, when you call the different library routines.

Amiga System Programmers Guide (Abacus/Data Becker)

This book is a mixture of both the »Hardware Manual« and the »Libraries and Devices« (the main subject is the hardware). It is different in the way that everything is explained with examples, but giving more examples of course cuts the number of subjects. It will be a good substitute for the hardware manual (with nearly no exception), but if you are a really serious library system coder, it cannot replace the books »Libraries and Devices« and »Includes and Autodocs«. If you only want to buy one big book, this is it.

Amiga Machine Language. (Abacus/Data Becker)

This is a little book that covers a lot of subjects about coding in machine language on the Amiga. It is based on a lot of small routines using the libraries. Nearly all small routines can be used together. It is easy to read and not so expensive. If you do not want to spend much money but want a little closer explanation about making windows, menus and gadgets, this book is good.but not a substitute for the Hardware Manual.

Programming the MC68000 (Sybex/Steve Williams)

This is a really good book about coding programs with the MC68000. First it gives a really close description about all commands and addressing modes available. It has also general program examples for the processor and discussions about how to make structured programs.

Of course there are other books, and maybe even better books. But the above are just the books I have found useful and think/hope that you can find useful too.

C. Error messages

When assembling a new program, you will probably get one or more error messages because of errors in your program. In order to correct these bugs you get a text that gives a very short explanation of what was found wrong.

Here is a closer description of each of the error messages.

1. Workspace Memory full

The assembler has to use additional memory for labels, reloc-tables, op-code etc. It takes this memory from the preallocated workspace memory. If not sufficient memory is available, this error message is issued.

2. Address Reg. Byte/Logic

An address register (An) has a restricted addressing ability. You can not access this register as a byte, and you cannot use this register in a logical operation (like »OR.W A1,D2«).

3. Address Reg. Expected

In some addressing modes you can only use an address register (e.g. (An), (An)+, -(An), nn(An,Rn)). In these modes where »An« is specified, you have to use an address register. Some commands like »LEA <ea>,An« etc. can also only use an address register as destination.

4. Data reg. expected

Like error 3 about address registers some instructions only accept data registers as operands.

5. Comma expected

When a command wants two operands or more you have to separate the arguments with commas. Some people like to put tabs between each argument. It is possible to put tabs/spaces after a comma, but not before.

6. Double Symbol

It is not possible to use the same name twice as a label or a constant. The only symbol that can be redefined is the »once« defined with the »SET« directive.

7. Unexpected end of file

A macro definition, a REPT-block or a IF-statement has to be ended with a »ENDM«, »ENDR« and »ENDC« statement. If the end of the program is reached before the matching end-statement, this error is issued.

8. User made FAIL

The assembler has a directive called »FAIL«, if this is encountered this is the error message you will get. You use this let´s say to check the size of statements for a macro, if the statements have an illegal size you are able to stop assembling and report this error.

9. Illegal Command

The command line understands a lot of different commands, but if you type an unknown command, this is illegal.

10. Illegal Address size

Specifying a direct address can be done in two ways. If the address is of word size, the following is possible: »MOVE.L $4.W,A6«. The two legal sizes are ».W« and ».L«.

11. Illegal Operand

A source line is divided into different fields. Label, Operator, Operands and Comment field. Each type requires a special syntax, see each type for a closer description. If an operand doesn't confirm to this description, it is illegal.

12. Illegal Operator

A source line is divided into different fields. Label, Operator, Operands and Comment field. A legal operator is an op-code, macro or a directive. If the operator isn't recognized as one of these types, it is an illegal operator.

13. Illegal Section type

There are three possible types of sections (»CODE«, »DATA« or »BSS«). These types are devided into subclasses:

CODE_P,	CODE_C,	CODE_F
DATA_P,	DATA_C,	DATA_F
BSS_P,	BSS_C,	BSS_F

See the section »directive« for a closer description. Any other types are illegal.

14. Illegal Operator in BSS area

It is possible to make different types of sections (»CODE«, »DATA« or »BSS«). A »BSS« section is non-initialized, this means that it cannot carry any constant values (it will be filled with zeros). An op-code or »DC«, »DCB« is code producing, and these are therefore illegal operators in a »BSS« area. »DS« (Define Storage) is non-code-producing and is therefore legal.

15. Illegal Order

The command »MOVEM« takes a register list as argument (e.g. »D0-D3/D4«). The registers must be written in ascending order with data registers first.

16. Illegal reg. size

The address mode »nn(An,Rn)« has an optional size specifier on the »Rn« register. The size specifier can be one of the following: ».W« or ».L« (e.g. »10(A3,D0.W)«).

17. Illegal Size

Some commands have a size specifier (».B«, ».W« or ».L«). Some commands can only use one or two of these sizes. If the size is independent of the operands, the size is illegal.

18. Illegal macro def.

Trying to define a macro inside a macro is not possible. But it is still possible to use a macro inside a macro.

19. Immediate operand ex.

Some commands, have a special addressing mode. If one of its operands is limited to an immediate expression (#nnn), and you write something else, this is the error you get (e.g. »MOVEQ D0,D1«).

20. Include Jam

This is a very hazardous massage, because it only occurs if you try to include a new file during »pass two«.

21. Macro overflow

You can nest macros (one macro uses another macro) up to a level of 25. This restriction is set because the machine has a limited stack space.

22. Conditional overflow

You can nest conditional statements up to a level of 25. This restriction is set because the machine has a limited stack space.

23. Section overflow

You can only use a limited number of sections (max. 255). This maximum is set to preserve memory.

24. Include overflow

You can nest include files up to a level of 5. The standard include files only reach a level of 3. This restriction is set because the machine has a limited stack space.

25. Repeat overflow

You can nest »REPT...ENDR« statements up to a level of 4. This restriction is set because of the limited stack space.

26. Double definition

Using the directive »BASEREG« in order to define the same address register as a base register twice is not possible.

27. Invalid Addressing Mode

This is the general addressing error. If you get this either the source and/or destination are wrong.

28. LOAD without ORG

You can assemble your data into an absolute memory location using the »ORG« statement. If you want the source to be assembled as if it was placed at the »ORG« address, but loaded into memory at another address, you use the »LOAD« statement. The assembler starts in a relative section. So using a »LOAD« statement without first having told the assembler to put the source at a absolute address, has no effect.

29. Missing Quote

When you use text strings starting with a quote (' " `) this text string also has to end with the same quote. To use a quote in the text, write two times in a row ('' "" ``).

30. NO operand space allowed

It is not possible to place tabs/spaces inside one operand. It is only possible to put tabs/spaces between two operands. So the following operand is not allowed: 10(a0 , d0).

31. NOT a constant/label

In calculations it is of course only possible to use constants or labels. With this I mean that it is impossible to use macro names in a calculation.

32. Not in Repeat area

Trying to end a repeat area using an »ENDR« is of course an impossible action if you are not in a repeat area.

33. Not in macro

To use a »ENDM«, »MEXIT« or »CMEXIT« directive trying to end a macro is not possible if you are not in a macro definition.

34. Out of Range 0 bit

The only way you can get this odd error message is from the command »shift« in memory. If you try to roll more than one bit at the time, you recieve this error message.

35. Out of Range 3 bit

Some commands have a 3 bit field (e.g. »ADDQ«, »SUBQ«). This can carry a value from 1 to 8 and a value out of this range is »Out of Range«

36. Out of Range 4 bit

A command like »TRAP« has a 4 bit value field telling the instruction which vector to jump to. This equals a value from »$0« to »$f«.

37. Out of Range 8 bit

A command like »MOVEQ« has a 8 bit field telling the instruction what signed value to put into the destination register. The signed 8 bit equals a value from -128 to 127. All normal byte-sized data will be accepted if the data is inside the range -256 to 255.

38. Out of Range 16 bit

Signed word sized data can be in the range -32768 to 32767. Normal word sized data will be accepted if the data is inside the range -65536 to 65535.

39. Relative Mode Error

If try to use a relativ value as a constant (e.g. word size or in calculations other than »+ -«) this is the error message you will get.

40. Reserved Word

You can use nearly any name for your labels and macros. But register names are reserved, because it would be impossible to distinguish an addressing mode using the name »d0« as an register from an addressing mode using »d0« as an address (e.g. »Move.W d0,d1«).

41. Right parenthes Expected

If you specify an arithmetic expression or an addressing mode using brackets.

42. String expected

If a directive wants a string as an argument you must give it as a string.

43. Undefined Symbol

Using a not defined symbol will cause this error. The only always defined symbols are the registers and »NARG« (see the macro directive).

44. Register Expected

Some commands can only accept a register as an argument. So if anything else is specified, this is the error message you get.

45. Word at Odd Address

Trying to assemble an »op-code« at an »odd boundary«, causes an »Word at Odd Address« error. This is because the MC68000 family is restricted to run commands on an even boundary.

46. Not local area

Local labels can only be placed after a global label. So trying to place a local label as the first label in your source will cause this error.

47. Code moved during pass 2

This is a critical error. »Pass« is used to define all labels to their appropriate value. »Pass two« is used to put »amm« code down into memory using all the right values from the defined labels. If suddenly a label points to another value during pass two something is very wrong. Usually this error occures if you have messed up some conditional assembly directives.

48. Bcc.B out of range in Macro

Normally »Bcc.B« commands will be forced to word (».W«) if the branch is out of range. But because this is not done in macros, you get this error. If you want to be sure that a branch is »byte size« ,place the branch in a macro.

49. Out of range (20 to 100)

Using the command »page length «(PLEN) takes a value between 20 and 100 as an argument.

50. Out of range (60 to 132)

Using the command »line length« (LLEN) takes a value between 60 and 132 as an argument.

51. Linker limitation

If you use »XREF's« it is not possible to use calculation between two »XREF's«.

Possible errors occuring during I/O activity:

52. File Error

The file was either not found or there was an error in the file that prevented it from beeing loaded.

53. No Files

Using the line command directory command »V - View« will return this message if the disk is empty.

54. No Object

If you have used »XREF« or »XDEF« in your source, it prevents an object file from beeing saved. This error is also returned if you haven´t assembled the source (having edited to much in the source after assembling it last time).

55. No File Space

The disk is full.

56. Printer Device Mission

If you want to print something, the computer has to load: »Printer.Device«, »Serial.Device«, »Parallel.Device« and the »printer configuration« from the disk. If one of these files is missing, printing is aborted.

57. Req.Library not found

This assembler uses a library placed on the disk. It is optional if you want to use this library. But if you have set the flag (default) telling the assembler to use this library, and this library isn't placed on the disk, the assembler will tell you this error message. This error can occur if you have booted from the workbench disk without copying »req.library« on to the workbench.

58. Illegal Path

You get this message if you use the intern »V - View« directory, specifying an non-existing directory.

59. Illegal Device

The built in disk commands »read/write« »sector/track« have an optional device specification (0 - 3). If the device does not exist, the device is illegal.

60. Write Protected

Like »illegal device« this is an error from the »track disk device«. Trying to write a sector or track on a write protected disk will cause this message.

61. No disk in drive

Like »illegal device« this is an error message from the »track disk device«, trying to write a sector or track on an empty drive will cause this message.

62. Not done

Cancelling an operation, will cause this message. Normally this is done by pressing <esc>.

D. Instruction timings

If you want to optimize very time-critical sections of your code this appendix can come in handy. Here it is possible to calculate all instruction evaluation times. The time you get here is in cycles. The normal Amiga run at a clock frequence of 7.16 MHz (7.14 MHz in Europe). So if you want to calculate how much time your routine takes, divide the number of clock cycles by the clock frequence like this:

```
ExeTime = Cycles/7160000
```

So if your routine uses 10000 cycles, it will execute in 0.013 sec.

How to use the tables:

First find your instruction. Make sure that you have found the right size and right addressing mode. If there is an »+« after the cycle value you will have to add the effective address calculation time. (<ea> time).

Here are a few precalculated instructions:

```
ADD.L    D0,10(A1,D1.W)     = 12 + 14 = 26 cycles
MOVE.W   (A0)+,100(A2)      = 18 cycles
ADD.L    D0,D0              = 8 cycles
```

Table D.1.
Effective Address Calculation timing

<ea>	Byte/Word	Long
Dn	0	0
An	0	0
(An)	4	8
(An)+	4	8
-(An)	6	10
d16(An)	8	12
d8(An,Rn)	10	14
xxxx.W	8	12
xxxx.L	12	16
d16(PC)	8	12
d8(PC,Rn)	10	14
#xxxx	4	8

Table D.2. Move Byte and Word Instruction Clock Periods

Src\Dest	Dn	An	(An)	(An)+	-(An)	d16(An)	d8(An,Rn)	xxx.W	xxx.L
Dn	4	4	8	8	8	12	14	12	16
An	4	4	8	8	8	12	14	12	16
(An)	8	8	12	12	12	16	18	16	20
(An)+	8	8	12	12	12	16	18	16	20
-(An)	10	10	14	14	14	18	20	18	22
d16(An)	12	12	16	16	16	20	22	20	24
d8(An,Rn)	14	14	18	18	18	22	24	22	26
xxxx.W	12	12	16	16	16	20	22	20	24
xxxx.L	16	16	20	20	20	24	26	24	28
d16(PC)	12	12	16	16	16	20	22	20	24
d8(PC,Rn)	14	14	18	18	18	22	24	22	26
#xxxx	8	8	12	12	12	16	18	16	20

Table D.3. Move Long Instruction Clock Periods

Src\Dest	Dn	An	(An)	(An)+	-(An)	d16(An)	d8(An,Rn)	xxx.W	xxx.L
Dn	4	4	12	12	14	16	18	16	20
An	4	4	12	12	14	16	18	16	20
(An)	12	12	20	20	20	24	26	24	28
(An)+	12	12	20	20	20	24	26	24	28
-(An)	14	14	22	22	22	26	28	26	30
d16(An)	16	16	24	24	24	28	30	28	32
d8(An,Rn)	18	18	26	26	26	30	32	30	34
xxxx.W	16	16	24	24	24	28	30	28	32
xxxx.L	20	20	28	28	28	32	34	32	36
d16(PC)	16	16	24	24	24	28	30	28	32
d8(PC,Rn)	18	18	26	26	26	30	32	30	34
#xxxx	12	12	20	20	20	24	26	24	28

Table D.4. Arithmetic, Logical and Compare Instruction Clock Periods

Opcode	Size	<ea>,An	<ea>,Dn	Dn,<ea>
ADD	B W	8+	4+	8+
	L	6+ **	6+ **	12+
AND	B W	-	4+	8+
	L	-	6+ **	12+
CMP	B W	6+	4+	-
	L	6+	6+	-
DIVS	W	-	158+ *	-
DIVU	W	-1	40+ *	-
EOR	B W	-	4+	8+
	L	-	8+	12+
MULS	W	-	70+ *	-
MULU	W	-	70+ *	-
OR	B W	-	4+	8+
	L	-	6+ **	12+
SUB	B W	8+	4+	8+
	L	6+ **	6+ **	12+

+	Add Effective address calculation time
*	Maximum value
**	Total of 8 cycles if <ea> is data register direct.

Table D.5. Immediate Instruction Clock Periods

Opcode	Size	#,Dn	#,An	#,<ea>
ADDI	B W	8	-	12+
	L	16	-	20+
ADDQ	B W	4	8	8+
	L	8	8	12+
ANDI	B W	8	-	12+
	L	16	-	20+
CMPI	B W	8	8	8+
	L	14	14	12+
EORI	B W	8	-	12+
	L	16	-	20+
MOVEQ	L	4	-	-
ORI	B W	8	-	12+
	L	16	-	20+
SUBI	B W	8	-	12+
	L	16	-	20+
SUBQ	B W	4	8	8+
	L	8	8	12+

+ Add Effective address calculation time

Table D.6. Single Operand Instuction Clock Periods

Opcode	Size	Register	Memory
CLR	B W	4	8+
	L	6	12+
NBCD	B	6	8+
NEG	B W	4	8+
	L	6	12+
NEGX	B W	4	8+
	L	6	12+
NOT	B W	4	8+
	L	6	12+
Scc	B false	4	8+
	B true	6	8+
TAS	B	4	10+
TST	B W 4 4+		
	L	4	4+

+ Add Effective address calculation time

Table D.7.
Shift and Rotate Instruction Clock Periods

Opcode	Size	Register	Memory
ASR	B W	6+2n	8+
ASL	L	8+2n	-
LSR	B W	6+2n	8+
LSL	L	8+2n	-
ROR	B W	6+2n	8+
ROL	L	8+2n	-
ROXR	B W	6+2n	8+
ROXL	L	8+2n	-

+ Add Effective address calculation time

Table D.8.
Bit Manipulation Instruction Clock Periods

Opcode	Size	Dn,Dn	Dn,<ea>	#n,Dn	#n,<ea>
BCHG	B	-	8+	-	12+
	L	8*	-	12*	-
BCLR	B	-	8+	-	12+
	L	10*	-	14*	-
BSET	B	-	8+	-	12+
	L	8*	-	12*	-
BTST	B	-	4+	-	8+
	L	6*	-	10*	-

+ Add Effective address calculation time
* Maximum value

Table D.9. Branch and Trap Instruction Clock Periods

Opcode	Disp	Taken	Not Taken
Bcc	B	10	8
	L	10	12
BRA	B	10	-
	W	10	-
BSR	B	18	-
	W	18	-
DBcc	cc True	-	12
	cc False	10	14
CHK	-	43+*	8+
TRAP	-	34	-
TRAPV	-	34	4

+ Add Effective address calculation time
* Maximum value

Table D.10. JMP, JSR, LEA, PEA and MOVEM Instruction Clock Periods

Opcode	Size	(An)	(An)+	-(An)	d16(An)	d8(An,Rn)	xx.W	xx.L	d16(PC)	d8(PC,Rn)
JMP	-	8	-	-	10	14	10	12	10	14
JSR	-	16	-	-	18	22	18	20	18	24
LEA	-	4	-	-	8	12	8	12	8	12
PEA	-	12	-	-	16	20	16	20	16	20
MOVEM	W	12+4n	12+4n	-	16+4n	18+4n	16+4n	20+4n	16+4n	18+4n
M-R	L	12+8n	12+8n	-	16+8n	18+8n	16+8n	20+8n	16+8n	18+8n
MOVEM	W	8+4n	-	8+4n	12+4n	14+4n	12+4n	16+4n	-	-
R-M	L	8+8n	-	8+8n	12+8n	14+8n	12+8n	16+8n	-	-

n is the number of registers to move

Table D.11. Multiprecision instruction Clock Periods.

Opcode	Size	Dn,Dn	M,M
ADDX	B W	4	18
	L	8	30
CMPM	B W	-	12
	L	-	20
SUBX	B W	4	18
	L	8	30
ABCD	B	6	18
SBCD	B	6	18

Table D.12. Miscellaneous Instruction Clock Periods

Opcode	Size	Rn	<ea>	Rn,<ea>	<ea>,Rn
MOVE SR,?	-	6	8+	-	-
MOVE ?,CCR	-	12	12+	-	-
MOVE ?,SR	-	12	12+	-	-
MOVEP	W	-	-	16	16
	L	-	-	24	24
EXG	-	6	-	-	-
EXT	W	4	-	-	-
	L	4	-	-	-
LINK	-	16	-	-	-
MOVE USP,?	-	4	-	-	-
MOVE ?,USP	-	4	-	-	-
NOP	-	4	-	-	-
RESET	-	132	-	-	-
RTE	-	20	-	-	-
RTR	-	20	-	-	-
RTS	-	16	-	-	-
STOP	-	4	-	-	-
SWAP	-	4	-	-	-
UNLK	-	12	-	-	-

+ Add Effective address calculation time

Table D.13. Exception Processing Clock Periods

Exception	Periods
Address Error	50
Bus Error	50
Interrupt	44*
Illegal Instruction	34
Privileged Instruction	34
Trace	34

* The interrupt acknowledge bus cycle is assumed to take 4 clock cycles.

Index