Programming in machine code on the Amiga

A. Forness & N. A. Holten

Issue 1

Content:

DATA SCHOOL
Postbox 62
North Engen 18
2980 Kokkedal

Phone 49 18 00 77
Postgiro 7 24 23 44

Dear Customer, students and pupils of DATA SCHOOL!
Welcome to DATA SCHOOL and thanks to you for choosing our course. It is very important that you do not let your family, friends or others copy this course. If you do, you break the first law of the copyright, and this leads to "piracy" and that DATA SCHOOL cannot afford to develop and distribute the issues of further courses - to the detriment of other programmers across the country. Hence please keep this urgent appeal in mind and DO NOT COPY THIS COURSE.

Sincerely,
DATA SCHOOL

Carsten Nordenhof

TABLE OF CONTENTS

INTRODUCTION

Welcome to this course in machine code programming on the amazing Amiga from Commodore. There are many around the world who have acquired this advanced home computer, but only very few who can take advantage of all its inherent possibilities. Commodore has already sold over 1,000,000 units, so if you decide you want to write programs for it, you have a large market to earn money. Once you have reviewed all the chapters - 12 in total - you'll be able to write your own programs on the machine, whether it's games or other applications. But remember, it is not enough just to sit down and read the issues of this course and then believe that knowledge comes. You must prepare you to try your way, experiment and practice a lot to master the programming. It is very important that you learn everything from scratch. The better your basic knowledge, the easier it becomes master Amiga's advanced topics. From chapter 1-11 a disk is published, where all examples from the course are stored (both as source- and object- code). The last chapter - chapter number 12 - will also be released together with a disk. This chapter will include tips and tricks specifically for both Amiga and generally for machine code programming. However, you must have successfully completed all 12 chapters in order to take full advantage of the disk. If you experience difficulties in getting a small program to work, you can send it to us on diskette together with a pre-paid envelope. We will do our best to find the error and return the disk with an explanation of what you did wrong. Unfortunately, we can not assume that the correct programming problems by telephone. We would like to conclude this introduction by wishing you luck and happiness. Do not give up! Although this course can be sometimes difficult to understand, but after all it’s easier than you think. Our tip: read the section again, and write a small program you can test.

Remember, Practice makes perfect!

Binary Numbering System

One of the first to need when you program in C is a numbering system that looks slightly different than what you are used to. (If you're already familiar with both the binary and hexadecimal numbering system, you can skip this section and go directly to the next section.) The decimal numbering system counts in groups of 10. If we, for example, enter the number 4362, so everybody knows that it means we have $2 *10^0$, $6*10^1$, $3*10^2$ and $4*10^3$.
Notice that the exponent of 10 is increasing by 1 by each digit from right to left.

But what is a binary numbering system and why you should use it? When you use a computer everything the computer stores is binary. A computer can only distinct between 0 and 1. Every character on the screen is internal in the machine encoded in binary. These 0 and 1 are called Bit and in the computer's memory those bits are organized in groups of eight. This bit-groups are called BYTE.

As already mentioned a byte consists of 8 digits each of which is called a bit. A bit in principle can be seen as a switch which can be on or off (voltage or no voltage).Imagine you have a data link between two computer interfaces - these two modes can be described as when there is current through the wire and 0 (zero) when there is no current through the wire.

If we now write 0100 1100 to describe the condition of a bytes, then it means that there is current 3 wires and 5 wires are without power. The rightmost bit is called bit 0. The bit left of bit 0 is called bit 1, the next is called BIT 2 etc. until the last bit, which is called bit 7. This gives a total of 8 bits which are numbered from 0 to 7 from the right to the left. It might be useful to split a byte in two: four bits in each group. These half-bytes are called NIBBLES. We will need these nibbles in the Hexadecimal numbering system, but it is also often used in the binary numbering system as you will see soon.

The capital letter "A" is stored as the decimal number 65 in a byte. Let us see how this is done. As mentioned before the machine uses groups of 8 bits (lines) with voltage of some of the wires to select a value. The decimal value 65 (for "A") is written in binary way like this: 01000001. The bit far right (bit 0) exclusive counts (2^0). bit 1 - just one digit to the left of bit 0 counts 2^1. The next bit to the left counts 4 (2^2), the next counts 8 (2^3) and this value is doubled each time we move one digit to left. As you see it is similar to the decimal system where the exponent increases from the right to the left – the only difference is the base. In the decimal system the base is 10, in the binary system the base is 2.

Read this the following table carefully – it represents the character “A” which is 65 in decimal:

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	1	0	0	0	0	0	1

The binary numbe 0100001 we used in this example is in the bottom line. As you can see the digits below the two values 64 and 1 are 1 which means that these values count. The sum of the two values is 65 (64+1) and represent the letter "A". Let us look at bit 0 and bit 1. In binary the numbering system has the base of 2 unlike in decimal system where the base is 10. The rightmost digit in binary is $2^0 = 1$, in decimal is $10^0 = 1$ and in hexadecimal $16^0 = 1$. All numbers with the power of 1 are equal to the number itself. So in binary $2^1 = 2$, in decimal $10^1 = 10$ and in hexadecimal $16^1 = 16$

We can now write our binary numbers 0100001 out mathematically we did in the DECIMAL numbering system:

$0 * 2^7 + 1 * 2^6 + 0 * 2^5 + 0 * 2^4 + 0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 =$
$0 * 128 + 1 * 64 + 0 * 32 + 0 * 16 + 0 * 8 + 0 * 4 + 0 * 2 + 1 * 1 =$
$0 + 64 + 0 + 0 + 0 + 0 + 0 + 1 = 65$

Read this table carefully on the next page we have translated the binary numbers from 1 to 12 to decimal numbers. When we want to show that a number (e.g. 65) is written in the decimal numbering system, we write to: 65_{10} which indicated that this number has the base of 10.

If we want to indicate that it is a binary number (e.g. 01000001) we writes 01000001_2 which indicates that this number uses the base of 2.

Here are the first twelve numbers in binary numbering system - the system having number 2 as the base:

$0000\ 0000_2 = 0_{10}$
$0000\ 0001_2 = 1_{10}$
$0000\ 0010_2 = 2_{10}$
$0000\ 0011_2 = 3_{10}$
$0000\ 0100_2 = 4_{10}$
$0000\ 0101_2 = 5_{10}$
$0000\ 0110_2 = 6_{10}$
$0000\ 0111_2 = 7_{10}$
$0000\ 1000_2 = 8_{10}$
$0000\ 1001_2 = 9_{10}$
$0000\ 1010_2 = 10_{10}$
$0000\ 1011_2 = 11_{10}$
$0000\ 1100_2 = 12_{10}$

Using these eight bits in a byte makes us able to write 256 numbers - from 0 to 255. 255 is the case if all bits are set to 1. Then the sum of all numbers which are represented through the position of the 8 digits just gives 255:

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	
128	+ 64	+ 32	+ 16	+ 8	+ 4	+ 2	+ 1	= 255
Binary: 1111 1111								

It is very important that you understand this. Yes it is so important that we have to write it again: It is very IMPORTANT THAT YOU UNDERSTAND THIS. Read it carefully so many times that you understand the system without any kind of doubt. Try to practice to convert from decimal to binary and vice versa as much as you can. You will save much time later if this part of the knowledge is located on the spinal cord, although you might not understand it all from the beginning. Give you a little time and try again later!

Try to solve the task 0101. You might send us the solution together with the other tasks in this issue. Do not forget to enclose an addressed and stamped reply envelope. The solution will be presented in the next issue, so it is not a must to send your solution to us. You do this only if you like - it but we recommend to do so!

Task 0101:
Rewrite the DECIMAL NUMBERS 0, 14, 15, 16, 27, 45, 63, 64, 65, 98, 99, 100, 101, 133, 147, 155, 156, 157, 200, 213, 228, 229, 240, 245, 253, 254, and 255 into binary numbers.

The hexadecimal numbering system

Now we hope you understand the binary number system by heart, and we therefore continue with the hexadecimal number system. The binary number system has the base of 2. The system we daily use – the decimal number system has 10 as the base number. The hexadecimal number system that we will learn now, has the base of 16. In the part of this chapter, which referred to binary number, we used the number 2 in subscript to indicate that it was a binary number. To mark the decimal numbers we wrote the number 10 in subscript to indicate it was in decimal. In almost all computer literature different indicators are used which are way more suited for typewriters. We introduce here the method to indicate the number system like it is used in the rest of this course: In front of a binary number we write the percentage character (%). In front of a hexadecimal number we write the dollar character ($).In front of a decimal number nothing is written.

The figure of 65 as we used before can be written in the following ways:

Binary:	% 010001
Decimal:	65
Hexadecimal:	$41

When one spells a binary number every digit is spelled separately, e.g. 10001 is spelled: one, , zero, zero, zero, one, not “ten thousand and one”. In the same way a hexadecimal number is spelled: four, one and not forty-one. Those who have already worked a lot with these number system often omit these rules and say “hex-forty-one” but the most important thing in this context is that the others person understands what you mean. It is therefore correct: Hex four-one when it is spelled and $41 when it is written on paper or screen. In some literature you will find a "&" to indicate hexadecimal numbers. On the Amiga everybody uses a dollar sign. Let us now look at how a hexadecimal number system works.

The DECIMAL system we use numbers from 0 to 9 (the base is 10). In the binary system, we use the numbers 0 and 1 (the base is 2). The hexadecimal system has 16 as the base and there are numbers from 0 to 15 to use but within one digit. The problem is that we have only 10 different numbers available, namely 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. This problem has been resolved in using letters beyond values of 9. The first six letters of the alphabet are used: A, B, C, D, E and F. (You may use the lowercase, but in this course we will use capital characters.) When we count Hexadecimal from 0 to 15, we must say: 0,1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

This means that in this number system A is the same as in 10_{10}, B is 11_{10}, C refers to 12_{10}, D matches 13_{10}, E represent 14_{10} and F is used for 15_{10}. Now we are able to create any numbers of these building blocks. This is done in the same way as already showed with the binary and DECIMAL numbers.

In the following number only 6 digits are used: $DFF180

Digit rightmost digit "0" in $DFF180 show how the multiplier for the 1-position (16^0). In this case: its zero. The digit to the left of "0" is "8." This is the multiplier for position 16 (16^1). The next figure in the row is "1", which is the multiplier for position 256 (16^2). The digit to the is "F". This represents the multiplier 15 for the position 4096 (16^3). Let us show what we think before it becomes too difficult to follow:

POS VALUE	16^5	16^4	16^3	16^2	16^1	16^0
HEX	D	F	F	1	8	0

POS VALUE (position value) is the value of each digit added, depending on its location (position) in the figure. The two DECIMAL numbers 400 and 4000 the number 4 are the same, but through the position of the number (the hundreds) 4 in 400 is less than 4000. This is because in 4000, the number four has a position (the thousands) with a value which is 10 times larger. Hexadecimal numbers works the same way as both binary and DECIMAL number. The farther at the right a digit in a number is located; the lower is its position value.

The digit 8 in number $DFF180 therefore has the lower POS VALUE than the digit 1 to its left although it's obviously the number 8 itself as such is greater than the number 1 itself. The different positions in a hexadecimal number, are as follows:

Position 6: $16^5 = 1048576$
Position 5: $16^4 = 65536$
Position 4: $16^3 = 4096$
Position 3: $16^2 = 256$
Position 2: $16^1 = 16$
Position 1: $16^0 = 1$

Let us figure out what the hex- number $DFF180 looks like if converted to a decimal number.

POS VALUE	16^5	16^4	16^3	16^2	16^1	16^0
HEX	D	F	F	1	8	0
Decimal	1048576*13	+ 65536*15	+ 4096*15	+ 256*1	+ 16*8	+ 1*0

Position 6:	1048576	*	13	=	13,631,488
Position 5:	65536	*	15	=	983040
Position 4:	4096	*	15	=	61440
Position 3:	256	*	1	=	256
Position 2:	16	*	8	=	128
Position 1:	1	*	0	=	0
Decimal	TOTAL				14676352

Already here we can see one of the advantages of hexadecimal numbers. They are much shorter and therefore much easier to remember. $DFF180 furthermore represents the color register number 0 (the background color) at Amiga. We return to this number many times, so we will not explain more about it right now, but it is certainly very easy to remember $DFF180 than 14676353, isn't it?

Here are the twenty first decimal numbers in sequence translated

Hexadecimal:			
1	$0001	11	$000B
2	$0002	12	$000C
3	$0003	13	$000D
4	$0004	14	$000E
5	$0005	15	$000F
6	$0006	16	$0010
7	$0007	17	$0011
8	$0008	18	$0012
9	$0009	19	$0013
10	$000A	20	$0014

So when to a HEX number 15 ($F) the value of 1 is added, then it becomes $10 which is $1*10^1 + 0*16^0$. This digit the new position to the left starts with 1 and the digit at the right position starts again with 0. In other words, once a value at a position has come to digit "F" and 1 is added, then an overflow of this position occurs and the number is enhanced with a new position to the left with 1. It is the same as in the decimal system from 9 to 10, from 19 to 20 or from 1499 to 1500.

TASK 0102: Here are some DECIMAL numbers you can convert to Hexadecimal: 30, 31, 32, 33, 63, 64, 65, 66, 127, 128, 129, 130, 170, 172, 254, 255, 256, 257, 2747, 2748, 2749, 4095, 4096, 4097, 4098, 4099, 5000, 10000, 20000, 43980, 43981.

Conversions between binary and hexadecimal numbers

It can be both inconvenient and boring to convert from decimal to binary, or decimal to hexadecimal. We recommend that you use a calculator. CASIO and Hewlett Packard have good calculators for basic conversions. These calculators can also convert the octal numbers (number of base 8), but you do not need octal numbers in this course, which is aimed specifically at Amiga.

If you obtain a calculator that can convert between decimal and binary/hexadecimal numbers, you're saving much time. It may also be useful to be able to convert a binary to a hexadecimal, or vice versa. However, it is so easy, that with little exercise you can do it faster than with a calculator.
The good thing about this method is that you do not need to learn why it works as it does. You will automatically know and understand and be able to apply it without effort when you have done it a few times. Let us take two bytes (represented as a binary number), divide them into nibbles as follows: %1001 1110 1011 0101. Each Nibble consists of four binary digits. With 4 bit a number up to 15 can be represented in binary and this is the reason why we take groups of 4 to convert a binary number to hex. You convert each nibble directly to a hex number from “0” to “F”. The position of the hex number corresponds to the position of the nibbles (groups of four binary digits). In short you convert 4 binary digits to 1 hex digit.

Leftmost Nibble (Nr. 3):

POS-VALUE:	8	4	2	1	
Binary:	1	0	0	1	
Decimal:	8	+ 0	+ 0	+ 1	= 9
HEX:	$9				

Next Nibble (Nr. 2):

POS-VALUE:	8	4	2	1	
Binary:	1	1	1	0	
Decimal:	8	+ 4	+ 2	+ 0	= 14
HEX:	$E.				

Next Nibble (Nr. 1):

POS-VALUE:	8	4	2	1	
Nibbles:	1	0	1	1	
Decimal:	8	+ 0	+ 2	+ 1	= 11
HEX:	$B				

Rightmost Nibble (Nr. 0):

POS-VALUE:	8	4	2	1	
Nibbles:	0	1	0	1	
Decimal:	0	+ 4	+ 0	+ 1	= 5
HEX:	$5				

Our binary number in Nibbles %1001 1110 1011 0101 converted to hex is $9EB5. (in decimal 40629). Once you've read this far, you can probably see that it is quite easy to convert from hexadecimal to binary. Now let's just do the opposite conversion:

For simplicity's sake we use the same number $9EB5:

$9	=	%1001 (P0S-VALUE 8 and 1)
$E	=	%1110 (P0S-VALUE 8, 4 and 2)
$B	=	%1011 (P0S-VALUE 8, 2 and 1)
$5	=	%0101 (P0S-VALUE 4 and 1)

Then set the four Nibbles together to form one binary number:
% 1001 1110 1011 0101.
Easy and straightforward, right?

When you program hardware oriented on the Amiga - and this is also valid for other computers – you use a tool called an assembler. It is a program that lets you program on a simple way, since it translates your assembler code you write into machine code – namely ones and zeros. These ones and zeros basically are the instructions that tell the different processors in the Amiga, to switch certain "wires" on (equals 1) of off (equals 0). This will be explained much better and in more detail in a later section. Theoretically we can type machine code directly into the machine's memory, but it would be virtually impossible to keep track of all ones and zeros which are needed. Imagine soon a number of more than 500000 digits are reached and finding an error like a lack of 1 or 0 – this is impossible. In assembler that is needed for Amiga, the following for example shows how to add the number $D to register A0 (much more on this later):

```
add.l   #$D, a0
```

Represented in machine code (ones and zeros) it looks like:
11010001111111000000000000000000000000000001101

It's clear which representation is easier to read and understand!

Task 0103:
Convert the following numbers to binary or hexadecimal:
%0000001001101000, %1010101111001101,
$ABCD, $FFFF, $D5C6.

What is an Assembler?

To be able to program in machine code on the Amiga, you need an assembler. It is a program that translates your instructions to machine code: ones and zeros. You write short commands (e.g. MOVE, ADD or CMP) and the assembler translates this. The translation process is called to assemble.

In these chapters we will use an assembler named K-Seka. If you do not already have this assembler, we recommend you to acquire it. If you want, you can buy it from us for 760,- kr Recommended retail price (October 1989) is 848,- kr.

Let us begin by explaining the principle of assembling a program in K-Seka. When you write your instructions, the K-Seka stores them in the Amiga's memory. When you think your program is ready to be tested, you ask the assembler (K-Seka) to convert your instructions into machine code (ones and zeros).

The instructions you wrote are called the source-code. The translation is called to assemble. The ones and zeros as the K-Seka produces (and store somewhere else in the Amiga's memory) is called the object-code.

If you already have K-Seka, you are probably familiar with the handling. For our students who have just purchased the program, we will explain how K-Seka started – and how to write, assemble and try running a small program example.

First start Amiga in the normal manner with Workbench. Then double-click on the icon labeled CLI.

When AMIGADOS shows the prompt: 1> so you write seka and press RETURN. When the K-Seka is once loaded, it asks you how many kilobytes (KB - 1 KB = 1024 bytes) you want to use as a working memory. 150 KB are usually enough. Therefore write: 150 and press RETURN. Now you (and K-Seka) are ready to program in machine-code.

Press the ESC-key. You will now enter the K-Seka's text editor (a separate part of the K-Seka). This is where you should write your programs. We show now an example you can type in. Whether you use uppercase or lowercase does not matter: (the example is on next page)

```
        move.w      #$4000, $DFF09A
        move.w      #$03A0, $DFF096
loop:
        move.w      $DFF006,$DFF180         ;Program Example MC0101
        btst        #6, $BFE001
        bne.s       loop
        move.w      #$83A0, $DFF096
        move.w      #$C000, $ DFF09A
        RTS
```

Press again on the ESC-key to get out of the text editor. At the moment don't worry about the program and what it does. Type it in exactly as it is and follow the instructions below.

After we have typed in the program in K-Seka's editor you must assemble it before it can be started. This is done with command "a". Type "a" (without the quotation marks) and press RETURN. K-Seka prompts now with OPTIONS>

This means that you need to specify how your program should be assembled. You can choose to have it written into the printer or on the screen. You type "v" + RETURN for the screen printing, or "p" + RETURN to print the output onto your printer. If you type "vh" the assembling messages appear on screen page by page. Skipping pages can be done with pressing the spacebar.

If you do not want a transcript of your source, press RETURN. If you select "p" (for printing) you will get prompted for: NAME> You can then write a title - a name - in your program. This name will be written at the top of the printed page.

If your program has an error, K-Seka stops the assembling process and tells you what line the error is. Once you have corrected your source code you have to assemble again. When the program is flawlessly, K-Seka will end the process with "No errors".

Now the program may be stored on disk. The "w" – command (w = write) stores your source code (source code), while the "wo" command (wo = write object) stores the executable object code. K-Seka will also ask you for a name for your object code or source code.

After the source code of your program is saved to disk, you can execute the assembled object code. This is done with the "j" (jump) command. ATTENTION! Wait before you start your program that the floppy drive motor has stopped - if you start the program before the floppy drive's led is off, you get write errors on the disk, and your file can't be read afterwards any more.

To stop the example, press the left mouse button.

You load an assembler from disk with the command "r" (r = read). If you have another program available in working memory, delete this first otherwise the new will be added to the old. You delete a source code by typing "ks" (kill source) and press RETURN. When K-Seka asks: "Are you sure?" you type "y" if that is what you want, or "n" (no), if you changed your mind.

When you want to exit K-Seka, type "!" and press RETURN. You will then be asked if you are sure (you could, e.g. have forgotten to store a program). Answer "y" if you want to exit or "n" if you don’t want to exit.

You now have learned the most basic commands in K-Seka. Continue to read program's manual and experiments yourself. Of course we will need several other commands but they will be explained later when needed.

CHIP MEMORY AND FAST MEMORY

Here is a little table showing how much memory an Amiga machines has. The abbreviation "KB" means, kilobytes and corresponds to 1024 bytes. Notice that difference in our “kg”-daily life, where “kilo” means 1000 in the data world “kilo” means “1024” the reason is that in binary 2 with the power of 10 (2^{10}) = 1024.

MACHINE	CHIP	FAST
Amiga 500	512 Kb	0 (can be expanded to 8.5 Mb)
Amiga 1000	512 Kb	0 (can be expanded to 8 MB)
Amiga 2000	1024 Kb	0 (can be expanded to 8 MB)

MB = megabytes, 1 MB = 1,048,576 bytes (2^{20}).

We should make one thing clear right away. When we talk about addresses in the Amiga, it is because the Amiga's memory can be referred to as a large number of empty storage locations (bytes). One can address these locations beginning with the first bytes which has the address 0, next to the address 1 and all following bytes are numbered sequentially until the end of the memory. The sixth byte has the address $5 (Bytes 0, 1, 2, 3, 4 and 5). For brevity and better readability Amiga addresses shall always be written in hexadecimal and preferably 6 digits: $000005.

Chip memory is the first Amiga 512 KB of memory. It is called like this because it is also accessible by the CUSTOM-CHIPS. To this part of memory, all processors (including the custom chips) have access to - even if they occasionally have to wait for each other. The custom chips Agnus (including floppy disk and memory copying), PAULA (including keyboard, joystick, sound) and Denis (including bitplane, screen update) can only use the chip memory.

And if you expand your Amiga with more memory, it is usually FAST memory. The term “fast” is because the main processor (68000) has exclusive access rights since the custom chips cannot access it. This prevents the cpu (68000) having to wait for the other processors and it leads to a faster access to the memory.

Keep in mind that not only the data processing in fast-ram is sped up, but also reading the next machine instruction of the currently running program. This leads to the effect that a program located in fast-ram is often carried out more quickly than if it is located in chip-memory. Therefore we name fast memory and chip memory as FAST-RAM and CHIP-RAM.

The term RAM comes from the English expression RANDOM ACCESS MEMORY and means freely translated "memory which can be freely written and read." This is in contrast to the part of memory called ROM (READ ONLY MEMORY) – which can only be read. Your programs will therefore be available in ram, for example. The Kickstart of the Amiga 500 and Amiga 2000 is in rom. As you probably know Kickstart is the name of the Amiga OS routines which are stored in the rom.

The Amiga 1000 the Kickstart is managed differently. The Kickstart (256 KB) is read from the disk into a certain portion of ram, then it gets "write protected" so that it can only be read. In this way the certain portion of memory is converted into a rom. When you turn off your Amiga 1000, the Kickstart in this portion of memory is lost and must be read again at the next start of the machine (cold-start).

The Kickstart - memory although it resides in rom is regarded as a FAST memory. The reason is that the portion of memory that used for the Kickstart, is so high that the specialized custom-chips do not have access to this part.

MEMORY CONFIGURATION (Address Fields):

MACHINE	chip RAM	FAST-RAM
Amiga 500	$000000 - $07FFFF	$200000 - $9FFFFF *)
Amiga 1000	$000000 - $07FFFF	$200000 - $9FFFFF
Amiga 2000	$000000 - $07FFFF	$200000 - $9FFFFF

*) Extra Memory (A501) to the underside of the Amiga 500 is at $C00000-$ C7FFFF and therefore FAST RAM. This because it ends up outside the first 512 Kb region (chip RAM) ranging from $000000 - $ 07FFFF.

Notice again that the memory addresses have 6 digits, even if it means that we must put some preceding zeros to get 6 digits.

SHORT COMPUTER DICTIONARY

ASSEMBLER	A program that translates the special assembler instructions into machine code – namely numbers of ones and zeroes
BINARY	Number system with the base 2. The daily used decimal system has the base of 10.
BIT	Is either one or zero (1, 0) – eight bits are composed to a byte.
BYTE	A data unit containing 8 bits.
CHIP-RAM	Memory which is used by both the cpu (MC68000) and the Amiga co-processors (custom-chips) Agnus, Paula and Denise.
CO-PROCESSOR	A processor built into the Amiga to perform specific functions (graphics, sound, disk-operations, memory copy, etc.) and thus support the cpu and increase the overall speed of the system.
CUSTOM-Chip	Special Amiga co-processors: Agnus, Paula and Denise.
DOS	Disk Operating System – the program controlling the disk station.
FAST-RAM	Memory which can only be addressed by the cpu (MC-68000).
COMPONENTS	The “hard” parts of the Amiga, or other computers, e.g., monitor, keyboard and processors.
HEXADECIMAL	Number system with the base of 16. (see also BINARY).
INTERRUPT	Disruption of the current cpu’s task". The cpu (main processor) is told to interrupt its current task and switch to some other part of a program.
K-SEKA	The assembler which is used in this course.
NIBBLE	A half byte, i.e. 4 bits.
RAM	The part of memory you can write to and read from.
ROM	The memory you can only read from. The information is stored once and for all by the manufacturer.
SOFTWARE	All programs you can read and execute with your Amiga, e.g., Workbench.

COMMENTS ON THE CHAPTER I

You have now hopefully read this issue thoroughly. It is a relatively easy issue you have gone through. But we strongly recommend again do not underestimate the information provided in this issue. If you do not learn these basics properly, you will have problems when it later becomes more difficult.

The tasks in this section, as already mentioned earlier, you can send to us and get them back with corrections. However, it is not really necessary because solutions will always be presented in the following issues. So in issue II you will find solutions to the tasks from this issue.

The following issue are automatically sent to you if you prepay the next issue to the attached giro by the 10th of the next months. This saves you delivery fee and postage and therefore, the course is much cheaper for you.

Unfortunately you can not buy, for example issue No 8 alone. You should first have purchased all the previous issues. Once the course has ended and you have reviewed all the issues, you can if you want it, obtain a test which checks how much you have learned. This test you will return to us and we will correct it. Then you will receive a certificate confirming that you have reviewed the course and passed the test.

You are very welcome to make comments to our courses. If you thinks that a section is explained poorly or you think that there is something missing in the issues we will be very glad to hear from you. As a thank you, we will include your name to the list of people who have helped us to realize the course.

Finally, there is only to say: do not give up! You do not need to be super smart to complete the course. It is enough with common sense and a little work.

Continue to enjoy.

Sincerely,
DATA SCHOOL

Carsten Nordenhof