Programming in machine code on the Amiga

A. Forness & N. A. Holten
Copyright 1989 Arcus
Copyright 1989 DATA SCHOOL

Issue 4

Content

Machine Code III
Bitmap
Color Registers
Bitplane DMA time
Over Scan

DATA SCHOOL
Postbox 62
Nordengen 18
2980 Kokkedal

Phone: 49 18 00 77
Postgiro: 7 24 23 44

MACHINE CODE III

So we need to program a bit in the MC again. We start with an easy program example that illustrates how different things are done in MC. First, the program's layout:

Program example 0401:

```
1	start:
2	move.l		#$00, D0
3	move.l		#$04, D1
4	lea.l		table, A0
5
6	loop0l:
7	add.w		(A0)+,D0
8	dbra		D1, loop01
9
10	lea.l		result, A0
11	move.l		D0,(A0)
13	RTS
14
15	result:
16	blk.l		1,0
17
18	table:
19	dc.w		2,4,6,8,10
```

This program takes the numbers defined in line 19 and adds them. The result ends up in a longword in the data register D0. The program performs not much, but it gives useful insights for later use.

Line 1: This is the start of the program and the starting point has been labeled "start". K-Seka has now a reference named "start" and puts the address of the beginning of the program into this variable.

Line 2: We need a data register to sum up the numbers (2, 4, 6, 8, and 10) – here we chose register D0. We clear the register by moving the constant value $00 into it. We could have used the command “clr.l D0” (clear long word of data register D0) instead. It would have given a faster code than “move.l #$00, D0”.

Line 3: We must sum up 5 numbers, so we put a counter into data register D1. The loop01 we let run five times and each time load the next number, which we sum to the number in D0. Since the loop01 has to be executed five times, we move the constant value of 4 into register D1 in which we count the number of loops. We count down this way: 4, 3, 2, 1 and 0 – in total 5.

Line 4: Loads the effective address of the start of the five numbers (from Line 19) into the address register A0. In line 19 we define five constant words and store the numbers 2,4,6,8 and 10. The assembler creates a variable called "table" (in line 18) and puts the address of the first of our five numbers (2) into the variable; to know where it can reference the numbers. We can properly access the subsequent numbers by using an offset. This offset is the distance (measured in number of bytes) between the first defined number (where the label points to), and the number we want to access. In this example it plays no role, where our machine program ends. There will always be the same distance between program startup and our five constants we have defined.

Line 6: Is a label we have called the “loop0l”. Add note the colon (:) to show that this is a label, and that the word “loop0l” is unique. See also the explanation for line 1 if you have many loops in your program, and this is very likely, it may make sense either to enumerate them sequentially: loop0l, loop02, loop03 etc. or providing them with descriptive name like e.g. “test”, “copper”, “buffer”, etc. In K-Seka the labels can consist of as many characters (and almost all characters) you want. You are not bound by a maximum length for your label names. Labels are not case-sensitive, so if you have two labels, called “dataschool” and "DATASCHOOL" K-Seka will give you an error because it sees the two labels as the same.

Line 7: In the first execution run of our loop “2” is fetched from the memory - A0 contains the address of the first number. It is then added to the register D0 and the address in A0 is increased by two bytes (remember we're working with words = 2 bytes) so the address in A0 points to the next number in memory.

Line 8: The instruction "dbra" decreases the register D1 about 1 and then D1 is tested if the content is less than zero (actually “-1”, it is explained in the second issue). If not the program jumps back to line 7, and the loop is executed once more.

Line 10: When all our numbers are added, we load the effective address of the memory we labeled "result" into A0 – which is defined at line 15.

Line 11: The result of the addition (as a longword) is moved from D0 to the memory part labeled “result” which address was previously loaded to A0 - see Line 10

Line 16: The program ends here and returns to the calling instance (e,g, to K-Seka if you started the program from within K-Seka or to the CLI if you started it from there).

Line 15: The label of the memory part called "result".

Line 16: Here a longword is reserved which is set to 0. In this longword our result is stored.

Line 18: Read the explanation for line 4 again (was explained there). Once you have written this program into the editor, you must assemble it. You have to type "a", and when K-Seka prompts for OPTIONS> write “vh” and press RETURN. It will now assemble the program and stop on each line. Press the spacebar to continue. If you simply type "v" as an option then the assembler will scroll non-stop. If you do not enter any options there is no output on the screen.

You start the program by typing "jstart" (jump to label "start").

At the list of registers the K-Seka shows after the program was executed, one can see that D0 contains $1E (30) - and it is just the sum of 2+4+6+8+10.

But we put the result into memory as a word (line 11). Can we get K-Seka to show it to us? If you writes "qresult" ("q" stands for query) K-Seka shows up the portion of memory that contains our longword (with sum $1E). In the label “result” the address was stored pointing to the definition of our longword. Beside the result you will see the figures $0002, $0004, $0006,$0008, $000A, which is the hexadecimal representation of our numbers.

TASK 0401: We hope you understand everything so far and you are able and willing to experiment with the program and change here and there. You can, for example, change the values of the numbers to be summed.

BITPLANES I

A bitplane is a memory area of the Amiga, which contains screen graphics information (data). The more colors you want to use, the more bitplanes you must reserved for video data. A pixel (short for picture element) is the smallest point you can see on the screen. A pixel can be "on" or "off". Information on a pixel whether it is "on" or "off" are stored in a bit that is part of the memory which was reserved for video data (the bitplanes).

Let's say that a bitplane starts at address $010000
BYTEs are arranged in this way:

```
PIXEL:     * * * * * * * * * * * * * * * * * * * * * ......
BIT No:    7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 ......
ADDRESS:   $010000         $010001         $010002
```

The first screen line downloads its pixel data from address $10000, $010001, $010002 and so on. When the first line is drawn, the electro-beam continues on the next line by download more data from video memory.

If you have a screen that is 320 pixels wide, it corresponds to 320/8 = 40 bytes per. line. This means that in our example the screen data for the second line is fetched from the address $010028 (which represents an offset of 40 from the start of our Bitplane).

Task 0402: What is the starting address for line 4 with our example above?

The Amiga has many different ways to display graphics which are called the graphic- modes. The following table lists the different graphics modes:

TYPE	Resolution (X*Y)	#BITPLANES	#COLORS
Lores	320 * 256	1 – 6	2 - 64
Hires	640 * 256	1-4	2-16
Lores-LACE	320 * 512	1-6	2-64
Hires-LACE	640 * 512	1-4	2-16
HAM	320 * 256	6	4096
HAM-LACE	320 * 512	6	4096

Abbreviations in the left column:

Lores	low resolution
Hires	high resolution
Lores-lace	low-resolution interlaced
Hires-lace	high resolution interlaced
HAM	hold and modify
HAM-lace	hold and modify interlaced

All these modes will be explained as we come in contact with them during the course.

When several bitplanes are used, one can imagine them placed "on top of each other” as a stack of cards. Try to imagine you 2 cards which have a small hole-raster over the whole surface. These two cards are added so that the holes are exactly above each other.

Imagine that each card represents a BITPLANE and a hole represents a BIT (which may be “1” or “0”). The cards are now exactly stacked upon each other - edge to edge and imagine you saw that a pin is stuck through one of the holes.

These two holes (read bits) as the pin is stuck through, the Amiga uses them to find out what color register to be used to color the point on the screen as the bits represent. For example if there through the holes at the top left corner the needle was punctured through this shows that the upper left pixels is set on your screen.

Let us use the Workbench screen as an example.

Workbench screen appears in hires with 640*256 pixels resolution and uses 2 bitplanes. In this setup (2 planes) there can not be more than 4 colors on the screen simultaneously. Study FIGURE 3 at the end of this issue and read the explanation below:

Let's say that bitplane 1 starts at address $010000 and bitplane 2 starts at address $020000. Then imagine that bitplane 1 is positioned over bitplane 2. Each pixel is now represented by 2 bits, one from each bitplane. As you know, with a 2-bit numbers one can express values from 0 to 3

%00 = 0
%01 = 1
%10 = 2
%11 = 3

If the first bit in bitplane 1 is set to "1" and the first BIT in bitplane 2 is set to "0" will be: %01 (also number "1" in decimal). Notice that the top bitplane contains the LSB bit (the least significant bit - that is why you get $01 and not $10, when they are coupled).

Let us study the so-called color registers in AMIGA. The Amiga has 32 color registers, each consisting of a word (16 bits) which defines the color. The bits are divided as follows:

BitNo:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Content:	-	-	-	-	R3	R2	R1	R0	G3	G2	G1	G0	B3	B2	B1	B0

As you may have guessed, that means R, G and B: red, green and blue. These colors are called the basic colors. By mixing different quantities of red, green and blue the accurate color is produced to be shown on the screen.

There is one small limitation: The Amiga can only mix them so that it will have 4096 different colors to "play" with (16 shades of each basic color): 16 * 16 * 16 = 4096). A TV-set uses the same basic color, but can mix several million different colors.

R0 to R3 is a 4-bit group (Nibble), which may contain a value starting with 0 up to 15 (see NOTE 1 if you can not remember why). It offers 16 different values, or with other words, 16 different shades. This also applies to G0 to G3, and B0to B3. The bits with number 12-15 are not used, so you can forget them. (Commodore strongly warns using the "remaining" bits to put data into for the sake of compatibility if some hardware will change in the future).

As already mentioned each color is adjusted in 16 steps. Let us take an example.

COLOR	RED	GREEN	BLUE	hexadecimal
Black	0	0	0	$0000
White	15	15	15	$0FFF
Yellow	15	15	0	$0FF0
Red	15	0	0	$0F00
Light blue	0	15	15	$00FF
Dark gray	7	7	7	$0777
Light gray	12	12	12	$0CCC
Pink	15	0	15	$0F0F

As you see the color becomes more intense the higher the number is. Therefore, the $0000 black (no color used), and $0FFF becomes white (full intensity of the three basic colors red, green and blue). Given that the values consist of 4 bits (1 Nibble) it is very easy to write the hexadecimal color:

Hexadecimal:	0	F	F	F
WHITE:	-	RED	GREEN	BLUE

Note the difference to the general color model where the basic colors are red, blue and yellow.

Now let’s get back to our example of bitplanes. So, when we are using 2 bitplanes as such the Workbench screen gets a value from 0 to 3 for each pixel (remember two bitplanes are two bits and two bits mean four colors). It works in the sense that if the value of a pixel is 2 – the pixel will be colored with the color the color register with the index 2 contains.

The first color register (number 0) is at $DFF180 and is a word broad. The next color register (number 1) is therefore at $DFF182 (a word = 2 Bytes). The third color register (number 2) is at address $DFF184. If you put the value $0F00 into the third register (number 2), then the pixels, we mentioned in the section above will be colored light red. Clear as black ink, isn’t it?

The color registers are numbered from 0 to 31. The edge of screen, the border retrieves the color of the first color register (number 0 the $DFF180). So: if the value of the two bits of a pixel are 0, it will have the same color as the border of the screen or the so called “background” color.

Let us now imagine a lores display with 3 bitplanes. It can display 8 different colors (three bitplanes mean three bit and 3 bit represent eight colors). Let's figure out how much memory is needed: We know that the screen is 320 pixels wide. We know also that a byte has eight bits. So: 320/8 = 40 bytes per. line. 40*256 lines = 10240 Bytes per bitplane. Finally we multiply the number of planes: 10240 * 3 = 30720 bytes. So the answer is that the screen – which is to display 8 colors – needs 30720 bytes of the Amiga’s chip memory.

This corresponds to 30 kB. So what is 30 kB? The small "k" means “kilo”. In everyday speech the little “k” means 1000 units of something, e.g. kilometers or kilograms. In the data world, however, the little “k” means 1024 units. Here we are talking about bytes, this means 30 kilobytes (30 kB). This number we get by dividing the number of bytes (30720) by 1024 (number of bytes in a "kilo"). The result is 30 kB. Note that the spelling of “kB”: small “k” and big “B” it is not a standardized notation and so you can find different variations!

Now you can figure out that each bitplane you use take 10 kB of the Amiga's chip memory. (10240/1024 = 10).

We now check our new knowledge with an application example:

Program example 0402 8(does work out of the box – see the stripes ;-):

```
1       move.w      #$01a0, $dff096          ; dmacon
2
3       move.w      #$1200, $dff100          ; bplcon0 1plane, color
4       move.w      #$0000, $dff102          ; bplcon1 pf2,1 scroll
5       move.w      #$0000, $dff104          ; bplcon2 pf prio
6       move.w      #0, $dff108              ; bpl1mod
7       move.w      #0, $dff10a              ; bpl2mod
8       move.w      #$2c81, $dff08e          ; diwstrt
9       move.w      #$f4c1, $dff090          ; diwstop high-bytes added (line below)
10      move.w      #$38c1, $dff090          ; diwstop
11      move.w      #$0038, $dff092          ; ddfstrt
12      move.w      #$00d0, $dff094          ; ddfstop
13
14      lea.l       screen, A1               ; address of screen mem.
15      lea.l       bplcop, A2               ; address of cl to a2
16      move.l      A1, D1                   ; copy addr. to d1
17      move.w      D1, 6(A2)                ; low word into the cl
18      swap        D1
19      move.w      D1, 2(A2)                ; hi word into the cl
20
21      lea.l       copper, A1               ; load address of copperlist
22      move.l      A1, $dff080              ; copcon
23
24      move.w      #$8180, $dff096          ; dmacon
25
26 wait:
27      btst        #6, $bfe001              ; wait for left mouse
28      bne         wait
29
30      move.w      #$0080, $dff096          ; dmacon
31      move.l      $04, A6                  ; exec base
32      move.l      156(A6), A1
33      move.l      38(A1), $dff080          ; restore original copper list
34      move.w      #$80a0, $dff096          ; dmacon
35      rts
36
37 copper:
38      dc.w        $2c01, $fffe
39      dc.w        $0100, $1200
40
41 bplcop:
42      dc.w        $00e0, $0000
43      dc.w        $00e2, $0000
44
45      dc.w        $0180, $0000             ; color 0 = black
46      dc.w        $0182, $0ff0             ; color 1 = yellow
47
```

```
48      dc.w        $ffdf, $fffe
49      dc.w        $2c01, $fffe
50      dc.w        $0100, $0200
51      dc.w        $ffff, $fffe
52
53 screen:
54      blk.b       10240, $80          ; try other patterns like $ff, $f0, $00
```

BITPLANES II

This section of bitplanes we start by explaining what is happening in the copper-list in the program of Issue III representing the red, white and blue lines:

Line 18 is waiting for monitor-line $90, h-position $01.
Line 19 is quite red in the color-register 0.
Line 20 is waiting for the monitor line $A0, h-position $01.
Line 21 is completely white in color-register 0.
Line 22 is waiting for the monitor line $A4, h-position $01.
Line 23 is quite blue in the color-register 0.
Line 24 is waiting for the monitor line $AA, h-position $01.
Line 25 is completely white in color-register 0.
Line 26 is waiting for the monitor line $AE, h-position $01.
Line 27 is quite red in the color-register 0.
Line 28 is waiting for monitor-line $BE, h- position $01.
Line 29 is completely black in color-register 0.

The copper-list is started by the main program and runs independently of everything else, over and over again. This is also an example of to use the copper with only one color-register to get more colors on the screen.

Task 0403: Study the copper list thoroughly. Then try to make different changes. See if you can change it so that the lines are blue and yellow.

We will continue with the custom chip registers, their functions and settings:

$DFF100 (WRITE) - BPLCON0 (BITPLANE CONTROL 0):

BIT No.	**FUNCTION**
15	hires
14	BPU2
13	BPU1
12	BPU0
11	HAM mode
10	dual play field
9	color enable
8	GAUD
7	--
6	--
5	--
4	--
3	LPEN enable
2	interlace
1	external resync
0	--

BIT 15: this bit should be set to “1” if to use hires (640 pixels horizontally)

BIT 12 - 14: This is a 3-bit group to specify how many bitplanes are used.

BIT 11: this bit should be set to “1” to use HAM (hold and modify) mode.

BIT 10: this bit should be set to “1” to activate the “dual-playfield” mode.

BIT 9: This bit will be set to “1” to get the color signal video output (no effect on A500).

8 BIT: This bit is set to “1”, to get the sound to the genlock (GENLOCK AUDIO ENABLE).

BIT 4-7: Not used.

BIT 3: this bit should be set to “1” if you want to use a light-pen.

BIT 2: this bit is set to “1” to activate interlace mode (512 lines vertically).

BIT 1: this bit is set to “1” when using a genlock.

BIT 0: Not used.

$DFF08E (WRITE) - DIWSTRT (DISPLAY WINDOW START) is organized as follows:

BIT No.	FUNCTION
15	V7
14	V6
13	V5
12	V4
11	V3
10	V2
9	V1
8	V0
7	H7
6	H6
5	H5
4	H4
3	H3
2	H2
1	H1
0	H0

$DFF090 (WRITE) - DIWSTOP (DISPLAY WINDOW STOP) is organized as follows:

BIT No.	FUNCTION
15	V7
14	V6
13	V5
12	V4
11	V3
10	V2
9	V1
8	V0
7	H7
6	H6
5	H5
4	H4
3	H3
2	H2
1	H1
0	H0

$DFF092 (WRITE) - DDFSTRT (DISPLAY DATAFETCH START):

BIT No.	FUNCTION
15	--
14	--
13	--
12	--
11	--
10	--
9	--
8	--
7	H8
6	H7
5	H6
4	H5
3	H4
2	H3
1	--
0	--

$DFF094 (WRITE) - DDFSTOP (DISPLAY DATAFETCH STOP):

BIT No.	FUNCTION
15	--
14	--
13	--
12	--
11	--
10	--
9	--
8	--
7	H8
6	H7
5	H6
4	H5
3	H4
2	H3
1	--
0	--

BITPLANE POINTERS:

BITPLANE	ADDRESS	HI / LO
1	$DFF0E0	HI (BIT 16-31)
1	$DFF0E2	LO (BIT 0-15)
2	$DFF0E4	HI (BIT 16-31)
2	$DFF0E6	LO (BIT 0-15)
3	$DFF0E8	HI (BIT 16-31)
3	$DFF0EA	LO (BIT 0-15)
4	$DFF0EC	HI (BIT 16-31)
4	$DFF0EE	LO (BIT 0-15)
5	$DFF0F0	HI (BIT 16-31)
5	$DFF0F2	LO (BIT 0-15)
6	$DFF0F4	HI (BIT 16-31)
6	$DFF0F6	LO (BIT 0-15)

BPLCONO ($DFF100): This register contains many new options. We start to explain them at the moment we need them. The rest comes in issue XII.

Hires set to "0" to set the width to 320 pixels (lores) or "1" to set the width to 640 pixels (Hires).

BPU0-2 is a 3-bit group. The Bits are used to set the number of bitplanes to be used (0-6). Color set to "1" to activate the color signal on video output. The bit is almost always to "1" (no effect on A500).

BPLCON1 register ($DFF102 used in context with scrolling the screen. We return to this register in a later issue.

BPLC0N2 register ($DFF104) used for the sprites and "dual-play-fields". We return to this register in issue V and issue XII.

MODUL0 registers ($DFF108 and $DFF10A) will be explained in issue VI in connection with the blitter.

DIWSTRT ($ DFF08E) used to determine where the upper left corner must of the screen is displayed on the monitor. The top line is usually line $2C (44 decimal). This position can be between $15 and $FF (21 and 255 decimal). Another "standard"-position is $1C (28 decimal). It leads to that the top line starts above the "plastic edge" of the monitor. This is called "overscan". Left position is usually $81 (lores) or $80 (Hires). If you want the picture to go beyond the monitor's edge (over-scan) use the position $71 (lores) or $70 (hires) that is 16 pixels wide.

Example: $2C81, $1C71, $2C80

We have now examined how to fix the sreen's top left corner. The next register is called DIWSTOP ($DFF090).

This register is used to determine the lower right corner of the screen. It is more complicated. Let us begin with the right position. This value has, as in the second register, 8 bits, i.e. a byte, which may contain values from 0 ($00) to 255 ($FF).

Let's assume we must set up a lores display which is has a dimension of 320 x 256 pixels. If we say that the left position is $81 (standard), it will say right position is $81 + $140 (320) = $1C1. As you see, this is too large to fit in a byte. Therefore, the machine was designed to complete the value. Thus, we just put $C1 into the register. The machine will add 256 ($100) to. ATTENTION! This happens in the chip's logic, rather than in software. This means that if you put value $00 into the register, the actual value being $100.

As the last position we must enter the "bottom line" of the screen. If you try to count this out, the numbers are also in this case too big. If the "top line" has the position $2C and we want a screen that is 256 lines high, we get the figure $2C + $100 (256) = $12C (you think it might work the same as before - unfortunately not!). The NTSC version of the Amiga which is used e.g. in the U.S. has only 200 lines vertically, and has no problems with this register for $2C + $C8 (200) = $F4. In Europe however, where we have 256 lines, we must do this: First put $F4 in the register and then added $38 (56) to the register. These two numbers will be added inside the chip and we get the value $F4 + $38 = $12C. This is done like this:

```
move.w      #$F4C1, $DFF090
move.w      #$38C1, $ DFF090
```

<u>These two operations must be executed right after each other, otherwise #$38C1 will be considered as a new value, and not be added to the register. Remember that "right position" ($C1) stays unchanged.</u>

Another thing we want to emphasize is that if you must set up a hires screen (640 pixels wide) so you need lores size (320 pixels) when you count the values out.

So: If you want the "left position" of the hires using $80 (which is standard hires) and add up to the 640, you will get a wrong number. You must take 640 / 2 = 320, for. We say that DIWSTRT ($DFF08E) and DIWSTOP ($DFF090) are "lores numbers".

DDFSTRT ($DFF092) and DDFSTOP ($DFF094) are two other registers, used to set up the screen. You do not really need to know what is happening with these registers because they depend on the DIWSTRT and DIWSTOP values.

Let us start with lores first. If we say that the "left position" in DIWSTRT is $81 and we use lores DDFSTRT must be $0038. Let us show it with the help of a formula:

LORES:

DDFSTRT: ($\$81/2_{10}) – 8{,}5_{10} = \$0038$

DDFSTOP: ($(((320_{10}/16_{10}) - 1_{10}) * 8_{10}) + \$0038 = \$00D0$

Remember that if DDFSTRT does not end with $0 or $8 then it is rounded down (e.g.: $0030, $0038, $0040, etc.). DDFSTOP values must also end at $0 or $8, but are rounded up (e.g.: $00E8, $00D0, $00D8, etc.).

HIRES:

DDFSTRT: ($\$80/2_{10}) - 4_{10} = \$003C$

DDFSTOP: ($(((640_{10}/16_{10}) - 2_{10}) * 4_{10}) + \$003C = \$00D4$

Values in DDFSTRT and DDFSTOP in hires mode always end with $4 or $C, and rounded down in the same manner as in lores. DDFSTRT is rounded down and DDFSTOP is rounded up.

Let us go back to the program example 0402.

Line 14: loads the effective address of our buffer into A1.

Line 15: loads the effective address of "bplcop:" (bottom of our copper list) into A2.

Line 16: copy content of A1 to D1.

Line 17: Insert the first 16 bits of D1 where the address in A2+6 points to. (A2 + 6) points to the low-word of bitplane pointer 1 in the copper-list. The number 6 of this instruction is called an offset. This means that 6 bytes are added to the address in A2 but only for this instruction – the address in A2 is not changed.

Line 18: Swap the first 16 and last 16 bits in D1. This means swap the high-word with the low-word of the register so you can access the high-word with a “move.w” instruction which can only move the lower 16 bit of a register.

Line 19: Insert the first 16 bits of D1 (which actually was the high-word before) into the address as (A2 + 2) which points the high-word of the bitplane pointer 1 in the copper list.

Line 53: This is a label for our screen.

Line 53: Here a block of data is reserved – it is used four our video data. 10240 bytes are enough for a screen with one bitplane and the dimensions of 320 x 256. (320 / 8 = 40. 40 * 256 = 10240. 10240 * 1 = 10240).

In line 54 we set the whole memory to 0 (blk.b 10240,0).

Try now to put a line into this:

```
53 screen:
54          dc.b $80
55          blk.b 10240, 0
```

You will now see a pixel at the top of the screen's left corner. You also can experiment with the fill-pattern of the block, e.g., if you change (the second parameter of) the instruction blk.b 10240, 0 to blk.b 10240, $80 you will see one pixel wide stripes with a clearance of 8 pixel.

Task 0404: Experiment with other numbers and fill patterns.

In directory "BEV04" of the course disk 1, you find the file labeled "SCREEN". This file contains an image you can load into your screen buffer. This is done in K-Seka via the "ri" (READ IMAGE) command. Do this:

```
Seka> ri
FILENAME> brev4/screen
BEGIN> screen
END>
```

When prompted with "BEGIN" that indicates the question where you want to load the file to. You could now enter an absolute address or a label. In this case, you enter our screen buffer label: "screen" and the file ends up there. On the question "END" press only RETURN to load the entire file. Remember that you must assemble the program first. The display buffer is cleared between each assembling, so you must read it back again.

BITPLANE III

```
1     move.w    #$01a0, $dff096
2
3     move.w    #$5200, $dff100
4     move.w    #0, $dff102
5     move.w    #0, $dff104
6     move.w    #0, $dff108
7     move.w    #0, $dff10a
8
9     move.w    #$1c71, $dff08e
10    move.w    #$f4d1, $dff090
11    move.w    #$40d1 ,$dff090
12
13    move.w    #$0030, $dff092
14    move.w    #$00d8, $dff094
15
```

```
16      lea.l           screen, A1
17      move.l          #$dff180, A2
18      moveq           #31, D0
19      colorloop:
20      move.w          (A1)+, (A2)+
21      dbra            D0, colorloop
22
23      move.l          A1, D1
24      lea.l           bplcop, A2
25      addq.l          #2, A2
26      moveq           #4, D0
27
28      bplloop:
29      swap            D1
30      move.w          D1, (A2)
31      addq.l          #4, A2
32      swap d1
33      move.w          D1, (A2)
34      addq.l          #4, A2
35      add.l           #12320, D1
36      dbra            D0, bplloop
37
38      lea.l           copper, A1
39      move.l          A1, $dff080
40
41      move.w          #$8180, $dff096
42
43      wait:
44      btst            #6, $bfe001
45      bne             wait
46
47      move.w          #$0080, $dff096
48
49      move.l          $4, A6
50      move.l          156(A6) ,A1
51      move.l          38(A1), $dff080
52
53      move.w          #$80a0, $dff096
54
55      rts
56
57      copper:
58      dc.w            $1c01, $fffe
59      dc.w            $0100, $5200
60      dc.w            $180, $000f
61      bplcop:
62      dc.w            $00e0, $0000
63      dc.w            $00e2, $0000
64      dc.w            $00e4, $0000
65      dc.w            $00e6, $0000
66      dc.w            $00e8, $0000
67      dc.w            $00ea, $0000
68      dc.w            $00ec, $0000
```

```
69      dc.w        $00ee, $0000
70      dc.w        $00f0, $0000
71      dc.w        $00f2, $0000
72
73      dc.w        $ffdf, $fffe
74
75      dc.w        $3401, $fffe
76
77      dc.w        $0100, $0200
78
79      dc.w        $ffff, $fffe
80
81      screen:
82      blk.l       $3c38, $8000
```

Line 1: Turn off bitplane-, copper- and sprite-DMA.

Line 3: sets 5 bitplanes.

Line 4: scroll value to 0
Line 5: bitplane priority to 0
Line 6-7: even and odd modulo to 0
Line 9: Sets the left screen position to $71 and upper line to $1C. This results in overscan – meaning both the left and upper position 16 PIXEL longer than normal.

Line 10: Sets the right screen position to $D1, bottom line to $F4. This will result in that we get right 16 pixel more. The lower line is now 244 - in other words, not overscan. It comes in the next line.

Line 11: Sets the right screen position to $D1, adds $48 the bottom line position. The bottom line is now been set at $F4 + $40, so overscan.

Line 9-11 has resulted in a screen which is 352 * 280

Line 13: Sets the display data fetch start to $0030 (see earlier for explanation).

Line 14: Sets display data fetch stop to $00D8. (See earlier for explanation).

Line 16: Loads the effective address of the "screen" into A1.

Line 17: Loads #$DFF180 into A2. $DFF180 as we know is the Color register 0. Please be aware that it is the actual figure is $DFF180 to be loaded into A2 and not a value that address $DFF180 suggests.

Line 18: Loads the number 31 (decimal) into D0. This register is used as numerator.

Line 19: The label “colorloop”.

Line 20: Copies the value address in A1 points to where the address A2 points to (one word at a time - 2 Bytes). Then increases both addresses in A1 and A2 – meaning that the next time the address in A1 and A2 point to the next word in memory. This instruction will read the color data which are located in the beginning of our image (which we must include after assembling with “ri”) into color registers.

Line 21: Decreases D0 by one and tests D0 to be “-1” if not jump back to the "color loop". It will run through the loop 32 times, so that we have initialized all color registers.

Line 23: Copy the content of A1 into D1. A1 points to the first image display data byte.

Line 24: Load the effective address of "bplcop:" into the register A2. This is the location where the bitplane pointers are initialized in the copperlist.

Line 25: “Add quick” the constant number of 2 to A2.

Line 26: Move “quick” the constant number 4 into D0. D0 is also used to count. This time it counts the Bitplanes (5 planes => it loops until -1)

Line 28: The label bplloop.

Line 29: SWAP is a new instruction, which is used for exchanging the high- with the low-word of a longword (bits 0-15 and bits16-31 respectively) in a register.

Line 30: Copy the word content (BIT 0-15 = the low-word) to the address A2 points to. As you see, it will give the upper word from Dl (note that we performed a SWAP before) into the copper-list’s move instruction (into the high-word of the first bitplane pointer). In this case it’s a MOVE instruction to a register. $DFF000 + $00E0 = $DFF0E0 (see explanation copper in issue III), which is bitplane-pointer to bitplane 1. The Bitplane-pointer at the address $DFF0E0 must contain the upper word (BIT 16-31 = high-word) of the address of our bitplane data. The address $DFF0E2, which also belongs to bitplane 1, should contain the lower-word (BIT 0-15 = low-word) of the address of our bitplane data. This word is also put into the COPPER-list in line 33. (see table of bitplane-registers in the previous section).

Line 31: Adds (“quick”) the constant 4 to A2. This causes A2 to point to next COPPER-instruction.

Line 32: Exchange high and low-word in D1 again => again in original position.

Line 33: Insert the low-word (BIT 0-15) from D1 into address that A2 points to.

Line 34: Adds (“quick”) the constant 4 to A2.

Line 35: Add the number 12320 (decimal) in D1. This leads that the address in D1 points to the next begin of the bitplane. This is used for copying the address into COPPER-list. The number is calculated as follows: Our screen is 352 pixels wide (because we have 16 PIXEL extra on each side) and 352 / 8 = 44 bytes. The height of the screen is 280 since we defined 16 PIXEL additional top and bottom, and 44 * 280 = 12320.

Line 36: Decrease the register D0 about 1. Test the register D0 for “-1”. If not jump back to "bplloop:" label. This instruction performs the loop 5 times so that we cover all bitplane addresses and copy them into the copper list.

Line 38: Load the address of the beginning of the copper-list into A1.

Line 39: Move the content of A1 at $DFF080 address - which is copper-pointer.

Line 41: Turn on bitplane- and copper-DMA.

Line 43: The label “wait”.

Line 44: Check if the seventh bit (Bit nr. 6) of address $BFE001 is not set (0). Bit No. 6 of this address is high ("1") if the left mouse button is not pressed - and "0" if the button is pressed. This instruction will set the Z-flag to "1" if the test was true (i.e. the bit was zero) - and "0" if not (i.e. the bit was set).

Line 45: This instruction makes a jump to "wait" if the Z-flag is "1". If the Z-flag is "0" it will continue to the next instruction.

Line 47: Turns off copper-DMA.

Line 49: Loads the exec-base to register A6.

Line 50-51: Moves the address of the old copper-list back (this is NOT good coding style!) to the copper-pointer. When the old copper-list starts again, it will show the Seka screen (if you started the program from Seka).

Line 53: Enable copper- and sprite-DMA.

Line 55: End – return from subroutine meaning that it will give control back to Seka.

Line 57: Label to the copper-list.

Line 58: The copper instruction represents a WAIT instruction ($01, $1C). This lets the copper wait until the electron beam has reached the start ($01) of line $1C (28 decimal). Remember that the copper is a separate processor with its own instructions that MUST not be confused with the main processor (MC68000) and its instructions.

While the copper carries out its program (the copper-list) in this program-example the MC68000 most of the time does nothing but going through its loop checking if the mouse button has been pressed. Another very important thing is that $01 the position on the line (in this case line $1C) is NOT the position 1 on the screen. The first position of the image (in this case) is $71, representing 112 PIXEL longer to right than the position 1 You must imagine that a screen line starts well outside the "visible" screen (about 5 cm left in the air from the edge of the screen).

Line 59: Here the COPPER would move the number $5200 at the address $DFF100 which is BITPLANE CONTROL 0 register. View previous example.

Line 61: The label “bplcop”

Line 62: The copper will move the words, which were inserted by the main processor in the line 23-36 and which represent the low-word of the first bitplane address to the bitplane pointer $DFF0E0. As previously mentioned, these are the bits 16-31 of the address of the bitplane pointer (the high word) to bitplane 1

Line 63: This line performs the same as above, but moves the low-word of the first bitplane address to $DFF0E2, which are the bits 0-15 of the address of the bitplane pointer (the low word) for bitplane 1.

Line 64-71: Performs the same as lines 62 and 63, but copies the high and low words of the addresses of bitplanes 2, 3, 4 and 5. Understood? We hope so since this is very important that you fully understand this!

Line 73: The copper waits for that the electron beam has reached the position on line $DF $FF. Remember that you specify pixel in pairs for the horizontal position in the WAIT instruction, so the actual position in this case is $FD * 2 = $1FA or 506 decimal. The copper-instruction makes it possible to wait for a position which is greater than $FF vertically. Line $100 will now be Line $00. So if we want to wait in line $120 we first carry out this instruction, then we wait for line $20

Line 75: Wait until line $34 - which is actually the line $134.

Line 77: The copper moves the number $0200 at $DFF100 address. As you see we have at the top of a list a copper-move, which moves $5200 into the same address. You may discover that the value $5200 puts the number of 5 bitplanes (see previous chapter). The number $0200 will put the number to 0 bitplanes, which gives a blank screen. If we don’t do that, we risk that the machine continues drawing up the screen below the defined screen bottom. You will now think of the screen size we defined with DIWSTRT and DIWSTOP?

Yes, this is correct… but some machines have an old version of the PAL chip and they have problems when put to a PAL video resolution (a screen which is higher than 200 lines). It is therefore best to use this method so that it works on all AMIGAs.

Line 79: This copper-instruction is often called STOP. The copper stops and jumps to the beginning of the copper list starting again processing the program. The copper will do this indefinitely until we turn off the copper-DMA.

Line 81: Label “screen” to our screen buffer.

Line 82: Here we declare our screen buffer. Note that we have set a block of longwords (blk.l). It can be handy if you have much memory, because in some individual cases Seka will assemble your program twice as fast. Play with the fill-pattern of the block definition to change the content of the bitplanes. This is handy if you not include the graphics for this example…

To get this program to show anything other than a black screen, you must put something into the display buffer. You find on the course disk in the directory "BREV04" a file called "SCRÉEN2". You have already learned how to read in such a file with Seka to a certain position (address/label) of your assembled code. Try to load this file to the "screen" label and run the program. HAVE FUN!

BITPLANE DMA TIMING

Finally in this issue, we must take a look at the "time slot allocation per horizontal line"which we had in issue II.
Let us first explain what a cycle is. A cycle is a period, which is as long as the time it takes for the electron beam to draw 2 pixels on the screen (4 hires Pixels). During this time you get an Access (reading or writing) to the memory.

Study first the bitplane-DMA portion of the figure in issue II. Figure 2 in issue IV is a sample of how the bitplane-DMA looks like when using lores, 2 bitplanes (4 colors) screen. The fields are numbered cycles and represent which bitplane-DMA is in use and downloads video data from memory. The number indicates the bitplane which retrieves the data.

The white (or unnumbered) fields are the available cycles, which may be used by the main processor (MC68000), copper- or blitter-DMA. The distance from the point of $38 in the figure for point of $40 corresponds to 16 pixels on the screen. The data to be used to display 16 pixels with 2 bitplanes is 4 Bytes data (a byte has 8 bits or 8 pixels). In one field bitplane-DMA retrieves a word (2 bytes). You can see in the figure, there are two numbered boxes each 16 pixel wide. It shows that it will get 2 per word. 16 pixels are enough data to show 2 bitplanes.

This way you can use the figure from issue II to see what happens when you use, e.g., 3 bitplanes (8 colors). As you see the line is split in two along the figure in issue II. The upper part of the line is true if you are in lores (320) while the lower part is true if you are in hires (640). As you see the lower part has two 1-bitplanes per 16 pixels width, while the top has only a single 1-bitplane. This is because the machine in hires shows twice as many pixels on the same width as in Lores.

So, the more bitplanes are witched on and operate the less cycles will be left for, e.g., the main processor. The machine will therefore be slower and slower with each activated bitplane.

COMMENTS ON ISSUE IV

When we examine such a broad field as the Amiga, it often happens that we just jump back and forth in our explanations. You must therefore sometimes take the former issues to refresh your knowledge. Sometimes you even have to wait for the next issue, to fully understand the current one you started with. Learning to program in MC on Amiga is done dynamically - it requires action. Read difficult sections several times! All the information you need is there, so if something seems foggy, read the section again and again ... and ... and.

It is very important that you use K-Seka (or another assembler) and practice your skills. Type in all the examples and try to change them. If the program crashes, then try again. It is amazing how much you learn from your own mistakes. We have learned Amiga MC using this trial and error "method. It worked for us and it will also work for you.

Whatever you find out - do not give up!

If you have not already purchased your course disks #1 then now is the time to do it. In the next issue is a very large example, which is only contained on the disk. Besides programming examples the disk also contains an IFF-converter, a hunker-, a wave converter and a sinus generator; so it is anyway good to have.

Sincerely,
DATA SCHOOL
Carsten Nordenhof

Mechanical, electronic, photographic or any other reproduction of this issue or parts thereof is not permitted according Danish copyright law. Amiga is a trademark and copyrighted by Commodore Computers.

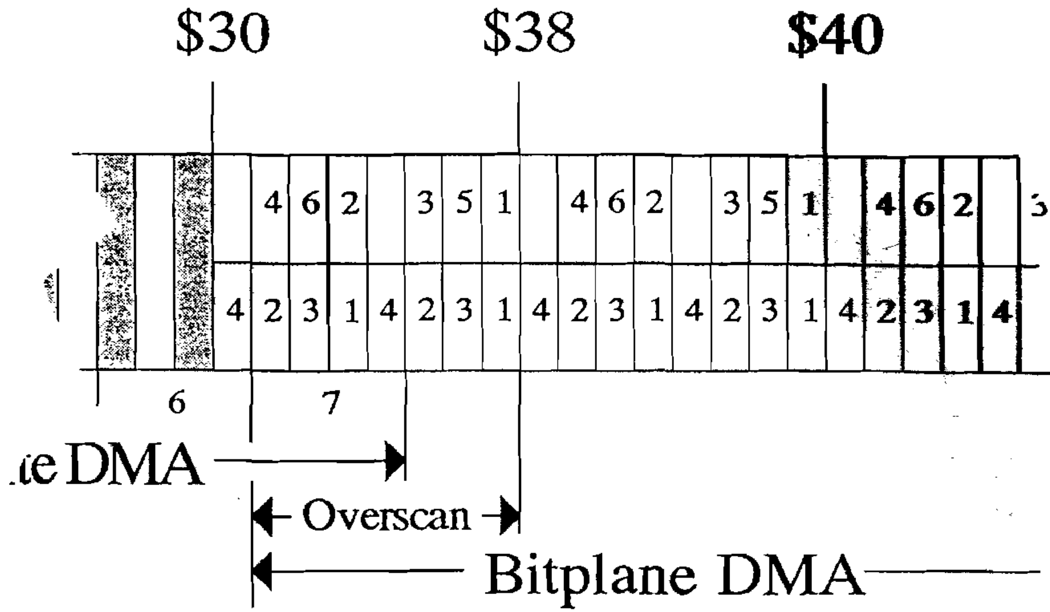
Solutions to tasks in issue III

Task 0301: The sum is: %100110100 (CARRY set)

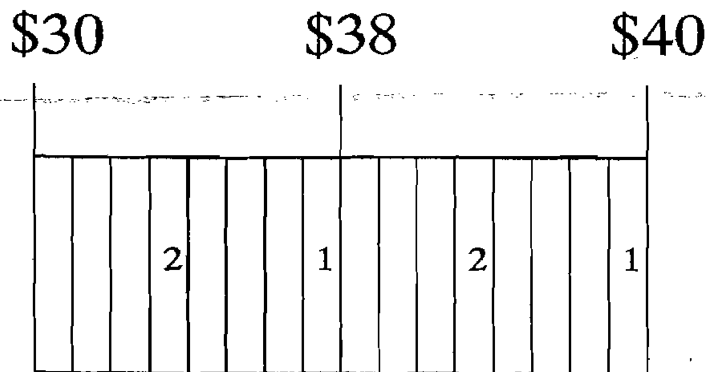
Task 0302: After an AND: %10000110

Task 0303: After a OR: %11011101

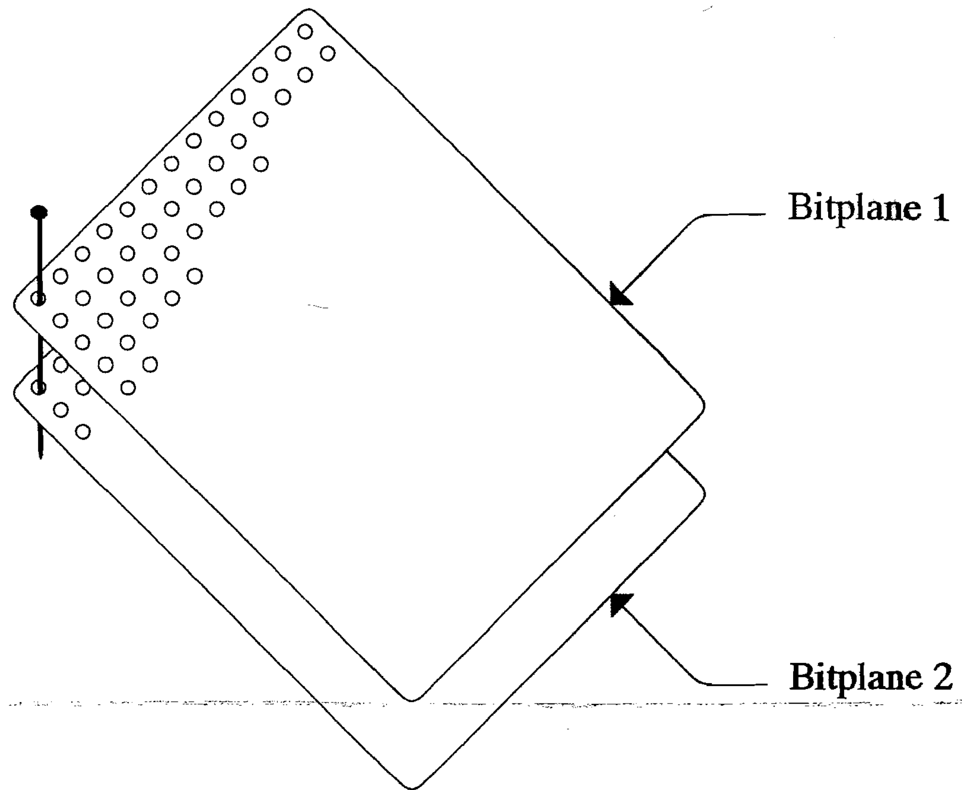
Task 0304: After an XOR: %00110101



Figur 1.



Figur 2.



Figur 3.