Programming in machine code on the Amiga

A. Forness & N. A. Holten

Issue 6

Content

DATA SCHOOL
Postbox 62
Nordengen 18
2980 Kokkedal

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

MACHINE CODE

The following small program is not on course disk #1. Type it in, assemble it and start it up by typing "j". Move the mouse around and look at the power-led.

```
1               move.w#$4000, $dff09a
2
3       main loop:
4               bset            #l, $bfe001
5               bsr             wait1
6               bclr            #l, $bfe001
7               bsr             wait2
8
9               btst            #6, $bfe001
10              BNE             main loop
11
12              move.w          #$c000, $dff09a
13              RTS
14
15      wait1:
16              move.w          $dff00a, d0
17              and.l           #$ff, d0
18      waitloop1:
19              dbra            d0, waitloop1
20              RTS
21
22      wait2:
23              move.w          $dff00a, d0
24              and.l           #$ff, d0
25              not.b           d0
26      waitloop2:
27              dbra            d0, waitloop2
28              RTS
```

Here is an explanation for this small program:

Line 1: Turn off all interrupts.

Line 4: Set bit 1 of $BFE001 to "1" which turns off the power led.

Line 5: Branch to the "wait1" label.

Line 6: Sets bit 1 of $ BFE001 to "0" which turns on the power led.

Line 7: Branch to the "wait2" label.

Line 9-10: Check if the mouse button is pressed. If not, branch back to "main loop". If it is it pressed, then continue to execute instructions at lines 12 and 13.

Line 12: Enable all interrupt again.

Line 13: Return to the calling instance – here, return back to K-Seka (i.e. quit program).

Line 16: Fetches the position of the mouse-pointer and put it into D0.

Line 17: Performs a logical AND with D0 so that we again only get BIT 0-7 (BIT 8-31 set "0" – i.e. masked out). With this instruction we mask out "not relevant bits" so that only the mouse x-position is left

Line 18-19: This loop is used as a delay it runs through the loop as many times as the value in register D0 (mouse position).

Line 20: Return to the program line 6

Line 23: Move the mouse position into D0.

Line 24: Mask out all bits except bit 0-7

Line 25: Invert (i.e. 0 becomes 1 and vice versa) bit 0-7 in D0. Only the bits 0-7 are inverted because we use ".b" in the instruction (a byte is eight bits).

Lines 26-27: Again a delay loop it runs through the loop the number of times stored in D0.

Line 28: Return to the program line 9

The above example works so that when you move the mouse from side to side, the POWER led is switched on and off. The whole secret is that if you turn on and off the power led you would not notice the change because it is done so fast.

Try to find out how it works as it does before you read on.

Did you find out, probably not. It is not that easy to behind the functionality by yourself. But when you read the explanation below so you understand it - surely!

If we let the led be lit as long as it is off, we will experience it as if the led is lit with a half its intensity. If we leave the led switched off for 90% of time and turned on 10% of the time, we experience it as if the intensity is 10% of full- therefore its light emission is very low. This regulation is performed at the routines "wait1" and "wait2".

LOGICAL MATHEMATICS

Now we look at an area which is used in many programs and seems to be difficult to understand:

Arithmetic operations are carried out by means of the logical operators:

AND, **OR** and **NOT**.

However, there is no reason to be frightened. Everything is very logical, but as with everything else in life - we must practice before we become a master.

An expression can, for example. look like this:

D = (A AND B) OR ((NOT A) AND C)

Just like in ordinary mathematics first begin to solve the innermost parentheses. This means that in expression (NOT A), A will be inverted. So if A was in 1010, it becomes after (NOT A) performed 0101 This is written in logical mathematics with an A with a line atop to mark the inverting - like this:

$\overline{A}$

The new reduced expression would look like this:

$D = (A \text{ AND } B) \text{ OR } (\overline{A} \text{ AND } C)$

In logical mathematics, a logical AND is written as a multiplication. It means that the brackets (A and B) can be written as follows:

A * B

In ordinary mathematics we often omit multiplication signs. It is jus not written.

Mathematicians are lazy people!

Well, so does the expression (A AND B) - which we could write as A * B – which can be further simplified to AB.

The whole term can now be written as follows:

$D = (AB) \text{ OR } (\overline{A} \text{ AND } C)$

We hope you are with us so far. Let's go on with the last bracket. As mentioned before a logical AND can be written as a multiplication in ordinary math. This leads to the expression ...

$(\overline{A}\ \text{AND}\ C)$ can be written: $(\overline{A}C)$

The term has now been reduced to:

$D = (AB)\ \text{OR}\ (\overline{A}C)$

So far everything should be clear, but we have another logical operation, namely the OR. In logical mathematics it is written as an addition, therefore, with a plus sign (+). We can now reduce the expression again:

$D = (AB) + (\overline{A}C)$

The parentheses can now be completely removed and the term looks in its simplest form like this:

$D = AB + \overline{A}C$

When reading the final expression, then it says:

D is equal to A times B plus the inverse A times C.

You use the same order as the arithmetic in ordinary mathematics:
Multiplications (and also divisions) are carried out before any addition (or subtraction), so the calculation is carried out like this:

A and B AND-linked with one another
A must be inverted and AND-linked with C
The two results must be OR-linked with each other.
The result appears in D.

NOTE: The logical mathematics does not use division and subtraction.

In this course you will not need to know more about logic mathematics than what we've reviewed. However, you may later have a major benefit from reading special literature on logical mathematics. It is invaluable and necessary knowledge, if you want to be a good programmer.

BLITTER I

The word blitter is an abbreviation for **BL**ock **I**mage **TR**ansferrer (a graphics block mover). The blitter in the Amiga can be used for many things. Let us start to count the scenarios the blitter can be used for:

- Copy large amounts of data from one location in memory to another.
- Move data bit-wise in memory (used in particular for scrolling).
- Move the shapes on a graphics screen.
- Move the shapes around on the screen, without destroying any underlying graphics (background).
- Drawing lines on the screen.

So the blitter can retrieve data from the memory for example and perform a logical operation for putting the data back again on the same memory location (or elsewhere).

The blitter can open up to 4 DMA channels at once. These DMA channels are called A, B, C and D. The channels A, B and C can only be used to retrieve data from memory, while the channel D is only used to write data to memory. DMA channels A, B and C are often called the SOURCE channels because they can only retrieve data. Channel D is called DESTINATION channel because it can only write data to the memory.

Imagine that we must copy the data located in memory area $10000 - $100FF (total 256 bytes) to the address $20000 - $200FF.

We can then use the operation D = A (D-channel is equal to A-Channel). Then, we set channel A to point to address $10000, and the channel D at $20000. The length is set at $100 and finally the blitter is started. It will then work on its own, so the main processor can continue its mission.

We can also perform logical operations with the blitter. The three functions: AND, OR, NOT. can be combined in various ways. For example, as in the previous chapter:

$D = AB + \overline{A}C$ (simplified from the term D = (A AND B) OR ((NOT A) AND C))

We come back to this later when we need it in practice. Let us look at the registers the blitter uses.

Table of register addresses of A, B, C and D:

BITS	Name	Address
BLTAPTH (A)	16-31	$DFF050
BLTAPTL (A)	0-15	$DFF052
BLTBPTH (B)	16-31	$DFF04C
BLTBPTL (B)	0-15	$DFF04E
BLTCPTH (C)	16-31	$DFF048
BLTCPTL (C)	0-15	$DFF04A
BLTDPTH (D)	16-31	$DFF054
BLTDPTL (D)	0-15	$DFF056

Modulo registers for A, B, C and D:

Name	Address
BLTAMOD	$DFF064
BLTBMOD	$DFF062
BLTCMOD	$DFF060
BLTDMOD	$DFF066

The blitter control registers and their configuration:

BLTCON0 - $DFF040

Bit	Function
15	ASH3
14	ASH2
13	ASH1
12	ASH0
11	USE A
10	USE B
9	USE C
8	USE D
7	LF7
6	LF6
5	LF5
4	LF4
3	LF3
2	LF2
1	LF1
0	LF0

ASHO - ASH3 = shifting values for the A-channel (A Shift). It can contain a value from 0 to 15.
USE A - USE D = with these bits the DMA channels are selected which are to be used in the blitter operation.
LF0 - LF7 = is used to set the logical function(s) the blitter is using on the channels (LF = LOGIC FUNKCTION).

BLTCON0 - $DFF040

Bit	Function
15	BSH3
14	BSH2
13	BSH1
12	BSH0
11	--
10	--
9	--
8	--
7	--
6	--
5	--
4	EFE
3	IFE
2	FCI
1	DESC
0	LINE

BSH0 - BSH3 = shifting values for B-Channel (B Shift). It can contain a valuee from 0 to 15

EFE, IFE, FCI = are used in connection with filling shapes, don't worry about it we come back to this topic later.

DESC = used to set the blitter in "reverse" (descending) mode.

This topic is explained in issue VII.

The blitter size register and its configuration:

BLTSIZE - $ DFF058

Bit	Function
15	H9
14	H8
13	H7
12	H6
11	H5
10	H4
9	H3
8	H2
7	H1
6	H0
5	W5
4	W4
3	W3
2	W2
1	W1
0	W0

H0 - H9 = height of the "blitt" operation; it can have values from 0 to 1023

W0 - W5 = width of the "blitt" operation; it can have values from 0 to 63

Notice that all the registers, as we have shown here are registers that you can only write (WRITE ONLY).

THE MODULO CONCEPT

We now continue with explaining how the MODULO registers (BLTxMOD) work. Study FIGURE 1 at the end of this issue. The figure shows a sample of a graphics display. A window in the image corresponds to 16 pixels in width, and 1 line in height. The numbers in the windows are the screen addresses. As you see a line consists of 40 bytes, which corresponds to a width of 320 pixels.

There is a rectangle to be drawn over a part of the screen. This rectangle is to be filled with data located in a buffer. The first address (upper left) of the destination of the rectangle is 126, while the first address in buffer is 0.

Imagine video memory is at address $10000 and that the buffer is at address $20000. Since the first address inside the rectangle $10000 + 126 = $1007E ($ 7E = 126) and the first address in the buffer will be $20000 + 0 = $20000.

The logical operation for the "blitt" must be D = A.

The size of the "blitt" is specified in the height and width. Height expressed the number of lines, while the width specified in blocks of 16 pixels. The width of a "blitt" in this case is 4. (4 * 16 = 64 pixels) and height is 3 (3 lines).

The addresses inside the rectangle do not lie in sequence such as in buffer. Let us see how the data must be moved so that it is copied correct.

BUFFER		SCREEN
0	->	126
2	->	128
4	->	130
6	->	132
8	->	166
10	->	168
12	->	170
14	->	172
16	->	206
18	->	208
20	->	210
22	->	212

As you see, there is a "hop" in order to display the address 132-166 and 172-206. This "hop", is to be entered into the MODULO registers and is used to get correct alignment called the MODULO.

MODULO is in our case 0 for the A-channel (which identifies the buffer), because all data in the buffer is located just after each other.

The MODULO for the D-channel (which points to video memory) is calculated as following:

The width of "blitt" is 4 WORDS (or 4 windows in the figure 1). A word contains 16 bits, that is 16 pixels. We know also that the screen width (line length) is 40 bytes, so we calculate the MODULO the like this:

40 - (64 / 8) = 40 - 8 = 32

We now have the following:

- Starting address of both DMA channels (A = $20000 and D = $1007E).
- MODULO for both channels (A = 0 and D = 32).
- The logical operation the blitter has to perform (D = A), direct copying from A to D9.
- Size of the "blitt" (height = 3 and width = 4)

The complete configuration in machine code would look like this:

```
1       MOVE.L      #$20000, $DFF050
2       MOVE.L      #$1007E, $DFF054
3       MOVE.W      #0, $DFF064
4       MOVE.W      #32, $DFF066
5       MOVE.W      #$09F0, $DFF040
6       MOVE.W      #$0000, $DFF042
7       MOVE.L      #$FFFFFFFF, $DFF044
8       MOVE.W      #$00C4, $DFF058
```

Line 1: the address of the source #$20000 (the address of the buffer) is moved into the A-channel pointer

Line 2: Move the destination address (the address of screen memory) #$1007E into the D-channel pointer.

Line 3: Sets the MODULO to 0 (zero) for channel A.

Line 4: Sets the MODULO to 32 for channel D.

Line 5: It is probably a bit complicated: see the table for Register (BLTON0). There you will see that this register is used among other things, to set the logical function ("miniterms") the blitter needs to perform during the memory transfer. In addition the needed DMA channels are selected (in this case A and D). When it comes to the logical operation, we have made a table of the most frequently used operations at the end of this issue. The value of the logical operation "A = D" being used in this case is $F0.

Line 6: This register is usually set to 0. We will com back to this register later when need it. See the table of records (BLTC0N1).

Line 7: This register is set to $FFFFFFFF. We will not explain what this register does in this context, but if you want to experiment with the blitter yourself, you can always safely put this register to $FFFFFFFF (for those who are curious it's the blitter A-MASK).

Line 8: This is the register that determines how much the blitter has to transfer (height and width). This value is calculated by a simple formula:

(height * 64) + width

So: 3 (lines) * 64 + 4 (word, which corresponds to 64 pixel) = 196 (decimal) = $00C4 (hexadecimal)

Let us now have a look at the example MC0601. Due to the length of the example we have not printed the code in this issue but you find it on the course disk #1).

First a simple explanation of the program:

Line 1-28: Interrupts are and the floppy are disabled. A screen with a bitplane of 320*256 pixel is set up in lores and the copper is started.

Line 30-42: This is the main program loop.

Line 44-53: Retrieving the old screen back and finish the program.

Line 55-67: This is a sub-routine which by the using blitter clears the screen (writes 0 in all display bitplanes).

Line 72 -103: "This sub-routine" blitt "figure into the screen. This is done through blitt.

Line 105-120: The copper-list.

Line 122-126: The first "BLK" command reserves 10240 bytes for the screen memory. The second set 640 bytes of data for the image (also on diskette #1).

Here is a detailed explanation of the new and unknown parts in the program:

Line 1-28: This should be known to all now.

Line 31-35: This routine waits until the electron beam has reached the screen line 300

Line 37: Branching to the routine "clear"

Line 39: Branching to the routinge "blitin"

Lines 41-41: Check whether left mouse button is pressed. If not - then branch back to the "main loop".

Line 44-53: Here the program ends.

Line 55: This routine clears the screen (the value 0 is copied in display bitplanes). We only use the D-channel for this blitt (i.e., no SOURCE channels). This results in only zeros are copied to the memory.

Line 56: Load the effective address of the screen memory (screen) into A1.

Line 58-60: This routine is waiting for the blitter to finish its previous job (or "blitt").

Line 62: Moves the address of the display memory into the D-channel register.

Line 63: Sets the modulo for D-channel to 0.

Line 64: Selects only D-channel and no logical operation.

Line 65: Move the value of 0 into BLTCON1

Line 66: Move the size of the blitt to the BLITSIZE register. The height is set to 256, and the width to 20 words (20 * 16 = 320). Write access to this register starts the blitter (DMA channels) automatically. So you must at last write to the register BLITSIZE ($DFF058). You need not to set any bits in the DMACON register ($DFF096) to start the blitter.

Line 67: Branches back to the main routine.

Line 69-70: Here we reserved one long-word to store the position of the blitter-object on the screen.

Line 72: This routine copies the object into the screen using the blitter.

Line 73: Load the effective address of object position to the Al register.

Line 74: Moves the object position, Al points to, into the D1 register.

Line 75: Increase the object position by 1 – remember A1 points to it. This means that 1 is added to our long-word and not in the register D1.

Line 77: Compare 216 with D1

Line 78: If D1 is not equal to 216, branch to the label "notbottom". This is a check whether the object has reached the bottom of the screen.

Line 80: Clear all bits in D1 (set all bits to zero)

Line 81: Clear all bits (set all bits to zero) in the object-position - A1 still points to.

Line 84: Loads the effective address of the "screen" to the register A1.

Line 85: Multiply D1 by 40 This is done because a display line contains 40 bytes. This instruction, we have not encountered before, but we think it is self-explanatory. However, we will talk about multiplication instructions (signed and unsigned) in a later issue. For the curious the MC-68000 also has an instruction that can perform a division. It looks like this: divu #5, D1

Line 86: Add the value located in D1 to the value of A1.

Line 87: Add the constant of 12 to the value in A1. This makes the object is centered horizontally on the screen.

Line 89: Load the effective address of the object into A2.

Line 91-93: Wait until a previously started blitt-job is finished.

Line 95: Move the address in A1 (the display address), to the D-channel register.

Line 96: Move the address in the A2 (the object address) to the A-channel register.

Line 97: Set the D-channel modulo to 24

Line 98: Set A-channel modulo to 0

Line 99: Put A-mask to $FFFFFFFF. The use of this register, will be explained later.

Line 100: Select the channels A and D, and set the logical operation: D = A.

Line 101: Move the value 0 into BLTC0N1 ($DFF042)

Line 102: Set blitt-size, height = 40 (lines) and width = 8 words (8 * 16 = 128 pixels) which starts the blitt.

Line 103: Return back to the calling instance here the main routine.

Line 105-120: The copper list is declare here

Line 122-126: The memory for the screen and the object is reserved.

To run this program you must first assemble it and then load the file "object" with the "ri" command to the label "object". The file is located in the same directory as the source code. Remember that the data for the blitter, the copper, the sprites, etc. must reside in chip ram – since the custom-chips have only access to this kind of ram.

BLITTER II

This chapter explains how the blitter can move a graphic on the screen without touching (or destroying) the graphics data in the background of the screen - that already existed on the screen before.

Study Figure 2 at the end of this issue. The object you see in this figure is to be blitted into the screen. The Mask has the same shape and size as the object, but has only one bitplane. It works like this:

- First, read the mouse position.
- Then copy the part (determined by the mouse position) of video memory, where the object will be copied to.
- Now blitt the object to the screen. If we'd copied the object-buffer directly into the video memory (bitmap), we would get a black border around the circle shaped object. This avoids the following:

The logical operation of this blitt is $D = AB + \overline{A}C$ or written another way:

D = (mask * object) + ($\overline{\text{mask}}$ * background)

The line above mask (A) means that the bits loaded from the mask-buffer by the blitter will be inverted (you remember: A zero (0) becomes a (1) and one (1) becomes zero (0)). It results in that:

Where the mask-buffer contains zeros (outside the circle-shape) the data located in the background buffer is copied (blitted) into the screen. Where the mask-buffer contains the bits set to 1 (that is inside the circle-shape), the data from the object-buffer (our object) will be copied into the screen.

- Finally, copy the background to the screen so that it looks as it was at the beginning.

This whole routine is repeated continuously so it looks like the object moves around the screen.

We now continue with a simple explanation of the example MC0602:

Line 1-17: Sets up a screen with a resolution of 320 * 256 pixels and with 5 bitplanes (32 colors).

Line 19-25: Here you put the colors (which are located at the beginning of the display buffer) into the color registers.

Line 27-40: This code copies the addresses of the 5 bitplanes into the copper-list.

Line 42-44: Move the address of the copper-list to the copper register.

Line 46: Start the bitplane and copper-DMA. As you see we have put an extra bit in the register. This is bit 10 of the DMACON register. If you set this bit then the blitter will be somewhat faster at every blitt. The reason will be explained in a later issue.

Line 52-65: This is the main routine, which is already known.

Line 67-77: Retrieving the original copper-list back and exit the program.

Line 79-121: This routine copies our object to the screen.

Line 123-171: This routine shifts the object and mask (bitwise) so that the figure can also be moved a horizontally.

Line173-185: This part reads out the mouse x- and y-position.

Line 187-222: This routine copies the screen background (screen memory) into background buffer ("backbuffer").

Line 224-259: This routine backs-up the screen from the background buffer.

Line 261-280: Here is the copper list declared.

Line 282-298: Here memory for all necessary buffers is declared.

Line 300-303: Here we have declared two long-words to keep the mouse position (x-and y-coordinates).

Here follows a brief explanation for all (sub routines):

Line 80: Loads the effective address of the "maskbuffer" to A1.

Line 81: Loads the effective address of the "backbuffer" to A3.

Line 82: Loads the effective address of "figbuffer" to A2.

Line 83: Loads the effective address of "screen" to A4.

Line 84: Adds 64 to the address in A4. This is done to skip the color data that is at the beginning of the display buffer.

Line 86: Loads the effective address of "mousex" to A0.

Line 87: Moves the value A0 points to D0.

Line 88-89: Do the same for the mouse y-position.

Line 91: Shift the contents of D0 four bits to the right. This instruction actually performs a division by 16 (2^4 or 2 * 2 * 2 * 2).

Line 92: Shift the contents of D0 one (1) bit to the left. This instruction actually performs multiplication by 2 (2^1).

Line 93: Multiply Dl by 40. This takes the y-position in relation to the screen (line 1 on the screen is 40 bytes).

Line 94: Add the contents of D0 (the x-position) to the value of A4 (screen).

Line 95: Add the contents of D1 (the y-position) to the value of A4.(screen)

Line 97: Moves the value 4 into D7 quickly.

Line 99-101: Wait until the blitter is available.

Line 103: Moves the address in A4 ("screen") into the D-channel register.

Line 104: Moves the address in A1 ("maskbuffer") into the A-channel register.

Line 105: Moves the address in A2 ("figbuffer") into the B-channel register.

Line 106: Moves the address in A3 ("backbuffer") into the C-channel register.

Line 107: Set the modulo for the D-channel to 32 (64 pixels wide by 40 - (64 / 8) = 32).

Line 108-110: Put the modulo for A, B and C channels to 0

Line 111: Add value SFFFFFFFF into a mask register.

Line 112: Here are the mini-terms $D = AB + \overline{A}C$ defined and all needed channels selected.

Line 113: Move the value 0 to BLTCON1.

Line 114: Move the blitt size to the BLITTSIZE register (height: 45 lines, width: 4 words = 64 pixels).

Line 116: Add 360 to the address in A2 so it points to the next bitplane in the "figbuffer" (45 * (64 / 8) = 360).

Line 117: Add 360 to the address in A3 so it points to the next bitplane in the "backbuffer".

Line 118: Add 10240 to address in A4 so it points to the next bitplane in the "screen". The address of the "mask" buffer stays the same since it has only one bitplane.

Line 120: Repeat this blitt 5 times - once for each of the 5 bitplanes.

Line 121: Go back to the calling instance here the main routine.

Line 124: Load the effective address of the object "fig" to A1.

Line 125: Load the effective address of "figbuffer" to A2.

Line 127-128: Get the mouse x-position in D1.

Line 130: Mask out (set to 0) all bits except bit 0-3 in D1

Line 131-132: Shift the content of Dl about 8 and then 4 bits to the left which turns out to be 12 bits in total. You have to change twice because MC-68000 (this method) allows only 8 shifts at a time.

Line 133: Add value $09F0 to D1. D1 will now contain $x9F0 where "x" is the number of data bits the blitter is shifting. In a issue VII, we looks at how the blitter performs such a shift. This blitt retrieves all the data from the object-buffer "fig" shifts the data for "x" bits to the right, and writes the result into the "figbuffer". The logical operation (mini-term) is D = A and D and A channels are selected.

Line 135: Move the constant value 4 into D7 quickly (counter for the biplanes).

Line 137-139: Wait until the blitter is available.

Line 141: Move the address in A2 ("figbuffer") to the D-channel register.

Line 142: Move the address in A1 ("fig") to the A-channel register.

Line 143-144: Set modulo for A- and D-channel to 0 since they have both the same size.

Line 145: Set the A-mask register to $FFFFFFFF.

Line 146: Move value located in the D1 (the number shifts) into BLTCON0 register.

Line 147: Move the value 0 into BLTCON1 register.

Line 148: The size of this blitt is also 45 lines * 4 words (8 Bytes or 64 pixels).

Line 150-151: Add 360 to both A1 and A2 to increase the addresses to their next bitplane.

Line 153: Repeats this piece of code for all 5 bitplanes.

Line 155-169: It is the same way for the mask. The only difference is that the mask has only one bitplane and therefore looping is not necessary.

Line 171: Go back to the calling instance – here the main routine.

Line 174-185: This routine reads the mouse positions and stores them into "mousex" and "mousey". The register $DFF00A will be explained in issue XI.

Line 188: Load the effective address of the "screen" to A1.

Line 189: Add 64 to the address in A1. This causes A1 to skip the color data that is in beginning of the display buffer.

Line 190: Load the effective address of the "back buffer" to A2.

Line 192-195: Read the addresses of mouse-position (x-and y-coordinates) and move them to D0 and D1.

Line 197-201: Find the first blitt position on the screen.

Line 203: Move value 4 into the D7 quickly

Line 205-207: Wait until the blitter is available.

Line 209: Move the address in A2 ("backbuffer") into the D-channel register.

Line 210: Move the address in A1 ("screen") into the A-channel register.

Line 211: Set modulo for D-channel to 0

Line 212: Set modulo for A-channel to 32

Line 213: A-mask register is set to $FFFFFFFF.

Line 214: Set the logic operation (mini-terms) to D = A, and select the A- and D-channel.

Line 215: Move the value of 0 into BLTCON1 register.

Line 216: Set the size to 45 lines * 4 words to the BLTSIZE register.

Line 218-219: Add 10240 to the address in A1 ("screen") and add 360 to the address in A2 ("backbuffer") so that the registers point to the next bitplain.

Line 221: This blitt repeat 5 times (each for one bitplane).
Line 222: Go back to the calling instance – here the main routine.
Line 224-259: This routine is almost identical to the "storeback"-routine. The only difference is that the data blitt is done in the opposite direction (from the background buffer to the screen memory).

To run this program you must first assemble it and then load the files: screen, figures and masks (with command "ri"). They are in the same directory as the source file).

We hope that you understand how the blitter works. We do not deny that understanding the blitter is the most complicated topic of the Amiga. Nevertheless it is possible to learn by yourself through experimenting with the code. So even though you might think this chapter is completely incomprehensible, don’t give up. Dig it through over and over again!

TABLE OF THE MOST USED LOGICAL BLITTER FUNCTIONS (MINI-TERMS)

LF	Value	LF	Value
$D = A$	$F0	$D = AB$	$C0
$D = \overline{A}$	$0F	$D = A\overline{B}$	$30
$D = B$	$CC	$D = \overline{A}B$	$0C
$D = \overline{B}$	$33	$D = \overline{AB}$	$03
$D = C$	$AA	$D = \overline{B}C$	$88
$D = \overline{C}$	$55	$D = B\overline{C}$	$44
$D = AC$	$A0	$D = \overline{BC}$	$22
$D = A\overline{C}$	$50	$D = BC$	$11
$D = \overline{A}C$	$0A	$D = A + \overline{B}$	$F3
$D = \overline{AC}$	$05	$D = \overline{A} + \overline{B}$	$3F
$D = A + B$	$FC	$D = A + \overline{C}$	$F5
$D = \overline{A} + B$	$CF	$D = \overline{A} + \overline{C}$	$5F
$D = A + C$	$FA	$D = B + \overline{C}$	$DD
$D = \overline{A} + C$	$AF	$D = \overline{B} + \overline{C}$	$77
$D = B + C$	$EE	$D = AB + A\overline{C}$	$CA
$D = \overline{B} + C$	$BB	$D = \overline{A}B + AC$	$AC

SOLUTIONS FOR TASKS IN ISSUE V

Task 0501: X-position is 363 and y-position is 238 and the height will be 35. All numbers are given in decimal.

Task 0502: A Sprite can be 16 pixels wide.

Task 0503: The Amiga has 8 sprites

Task 0504: D0 will look like this:
10110010110100110101011100010010

Task 0505: D0 will get its value from address $000554

Task 0506: The stack of the Amiga is used to:

- temporarily store data
- store the return addresses for the processor (BSR/JSR and RTS).

TASKS FOR ISSUE VI

Task 0601: Try to translate MODULO to a good short English explanation.

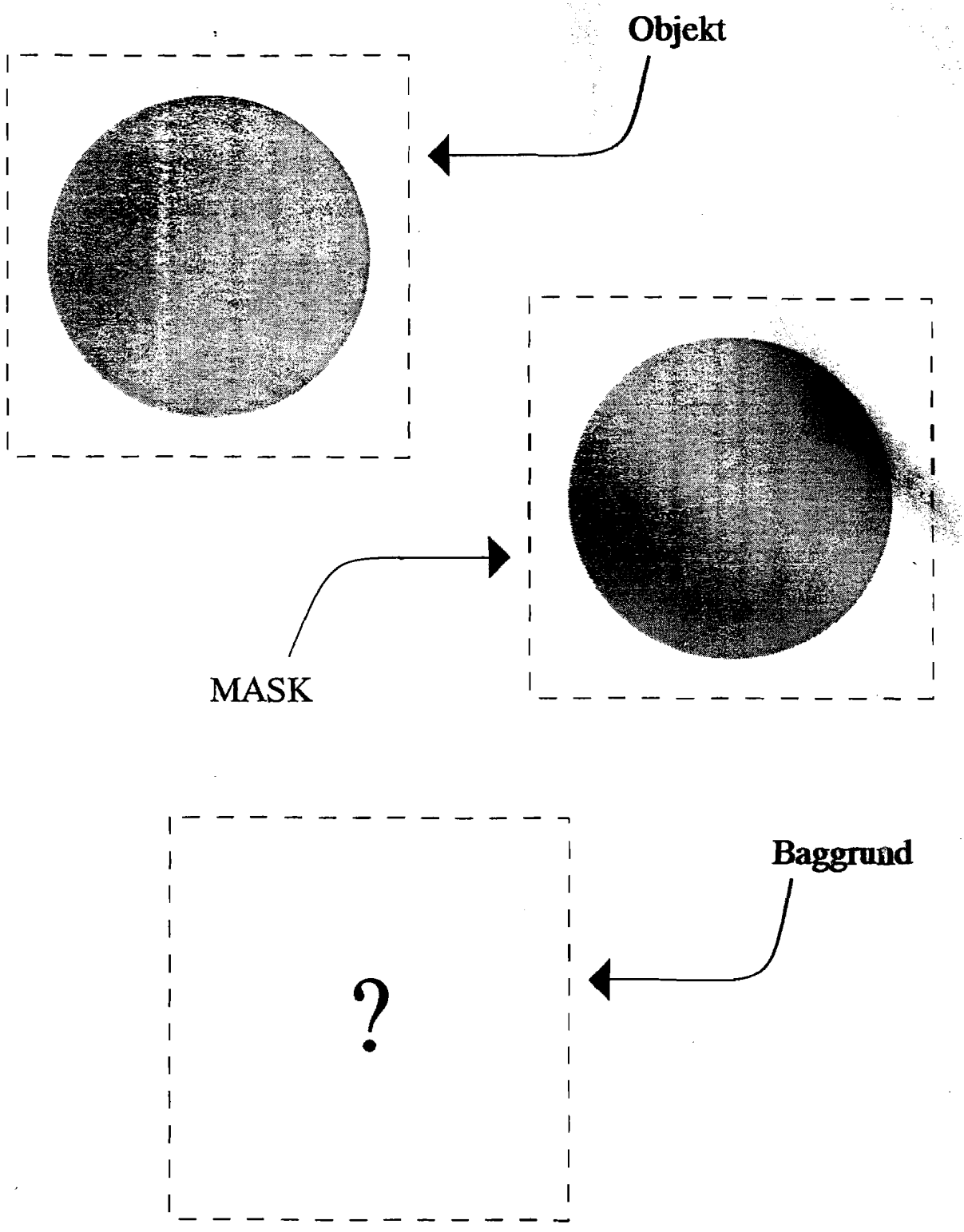
Task 0602: How high and how wide is a blitt with this BLITSIZE: $1A69

Task 0603: What happens with A in this case: $\overline{A}$

Task 0604: How many pixel horizontally is the minimum wdith which can be defined in bltsize (except 0).

0	2	4	6	8	10	12	14	16	
40	42	44	46	48	50	52	54	56	
80	82	84	86	88	90	92	94	96	
120	122	124	126	128	130	132	134	136	
160	162	164	166	168	170	172	174	176	
200	202	204	206	208	210	212	214	216	
240	242	244	246	248	250	252	254	256	
280	282	284	286	288	290	292	294	296	
320	322	324	326	328	330	332	334	336	
360	362	364	366	368	370	372	374	376	
400	402	404	406	408					
440	442	444							

Figur 1



Figur 2.