# Programming in machine code on the Amiga

A. Forness & N. A. Holten

Issue 5

Content

sprite
"Follow Me"
Machine Code IV

DATA SCHOOL
Postbox 62
Nordengen 18
2980 Kokkedal

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

## SPRITES 1

A is a graphical object i.e. a small image that can be displayed and moved around the screen independent of the bitmaps (or the rest of the screen). The Amiga has 8 sprites, which can be displayed at once. A sprite can contain 4 colors (with a little trick up to 16 colors - more on that in issue XII). The width is 16 pixels, but the height can be freely defined. The Workbench arrow is an example of a sprite.

We work out an example. Notice that the example is not shown in this issue. You need the course disk 1 and K-Seka, in order to keep up. Therefore you load the program MC0501 using K-Seka and have the program listing on the screen while reading the explanation below:

MC0501 Application Example:

| | |
|---|---|
| Line 1: | This MOVE switches off all interrupts. The whole system is "frozen" so that only our program runs. Interrupts will be examined and explained in issue IX. |
| Line 3-4: | If you start a program directly from the disk and turn off all interrupts it has the effect that the disk station does not stop by itself when program is read and immediately started. We solve this problem stopping the motor inside our program. This will be explained in detail in the issue X. |
| Lines 6-17: | These lines set up a screen (bitmap) and should be already known. |
| Lines 19-24: | Adds sprite addresses into copper-list sprite pointers (sprites are numbered from 0 to 27). A sprite pointer is like a bitplane-pointer divided into high and low address. |
| Line 26-39: | Retrieving the address of the "blank" (or empty) sprite and put it in the copper list area to the sprites for sprite 1-7. You should configure all 8 sprites, even if you only want to use one. |

| | |
|---|---|
| Line 41-51: | These lines retrieve the address to the screen and put it in copper list. Then the copper-DMA, the biplane-DMA and sprite-DMA is started. Note the program line 50 – you might not have seen this MOVE to this address before. It is not entirely necessary but it should be in order to ensure that the copper list is truly started at all Amiga (also in future). |
| Line 53-58: | This little routine will have the processor to wait until the electron beam reaches line 0. Remember that line 0 is not visible and it's not the bitmap's top line but the screen's top line – the upper line on bitmap is usually line 44 ($2C). Imagine that the line 0 lies above the monitor's "plastic edge". The screen lines 0-19 is often called the “vertical-blanking”. |
| Line 60-65: | This routine waits for the electron beam to reach line 1. The reason why we wait on line 0, and then line 1 is that if we just waiting in line 20, we risk getting an uneven movement of the sprite. The main routine performs so fast that when jumping back to program line 53 (wait:), there is some possibility that the electron beam is still drawing line 0 - the Amiga is a fast computer! |
| Line 67: | This instruction jumps to the move sprite routine and jumps back again when the processor encounters the RTS. For those who have knowledge of BASIC: this instruction can be compared to the GOSUB and RETURN instructions. The instruction will be reviewed in the section about “machine code” in this issue. |
| Lines 69-70: | Check if the left mouse button is pressed. If not, then jump back to label "wait". |
| Line 72: | Turns off the copper DMA. |
| Line 74-76: | Retrieving the address of the old copper list (that one which belongs to the Workbench) and puts it into the copper-pointer. |
| Line 78: | Starts copper-DMA again. |
| Line 80: | This MOVE enables all interrupts again. |
| Line 81: | Exits the program. |

| | |
|---|---|
| Line 84-94: | This routine moves the sprite on the screen. View explanation below. |
| Line 97-130: | The copper-list. |
| Lines 132-133: | Here is our memory four our screen defined. |
| Line 136-152: | Sprite data is defined here (explained below.). |
| Lines 154-155: | Here's the "blank" sprite defined. Study the sprite data in the example. As you see the first longword (2 words) are set to 0. This longword contains the position word, the position the sprite will have on the screen. Lines 84-94 update these values, so that the sprite moves around the screen. Let us now look at sprite data: |

| | |
|---|---|
| LONGWORD 1: | position and height of the sprite |
| LONGWORD 2: | graphics data |
| LONGWORD 3: | graphics data |
| LONGWORD 4: | graphics data |
| ... (as high as you want) | |
| LONGWORD ?: | must be 0 (the last data line) |

We divide the first longword in bytes:

| | | | | |
|---|---|---|---|---|
| LONGWORD: | $00 | 00, | $00 | 00 |
| BYTE-NR.: | 0 | 1 | 2 | 3 |

BYTE 0: Bit 0-7 defines the vertical position of the sprite's top line.

BYTE 1: Does the bit 1-8 in the horizontal position of  sprite left edge.

BYTE 2: Does the bit 0-7 for the vertical position of the sprite bottom line.

BYTE 3: Bit 0 contains bit 0 of the horizontal position of the sprite's left edge.

Bit 1 contains bit 8 of the vertical position of the sprite's bottom line.

Bit 2 contains bit 8 for the vertical position of top line of sprite.

Bit 3-6 are not used.

Bit 7 is explained in issue XII. The bit is used to indicate "16 colors" sprite (1 = 16 colors, and 0 = 4 colors).

As you see there are 9-bit values used for the positions in the sprite. As you already know, one byte can only hold 8 bits.

Therefore, the bit which does not fit into the byte is stored in byte 3. Please notice that the horizontal position in byte 1 contains bit 1 to 8, not 0-7, so that bit 0 becomes the overflow bit, and is presented in byte 3

Sample position:	$5080, $5500

The sprite's top line lands at vertical screen line of $50 (or 80 in decimal). Left edge lands on $80 * 2 (256 decimal). We must multiply the position value by 2 because the bit 0 is not present. The height of the sprite will be $55 - $50 (i.e. 5 pixels or lines). As mentioned earlier, the width of a sprite always 16 pixels. It should also be stressed here that the sprite is always in lores (lores = low resolution) regardless of the display resolution being used.

$5080, $5501

This example will place sprite in the same place as in example above, but with the exception that the horizontal position will be 257 ($101).

The following explains the layout (or appearance) of sprite pixel data:

A long-word is a line (16 pixels) with sprite data. The longword splits into two words (as in the example). As mentioned earlier, a sprite displays 4 colors (3 + background). If we compare with bitplanes it would be like that: the first word would be in bitplane 1, and second word would be in bitplane 2. Here some examples:

| | | |
|---|---|---|
| $0000, $0000 | = | one line (16 pixels) with sprite color 0 |
| $FFFF, $0000 | = | a line with sprite color 1 |
| $0000, $FFFF | = | a line with sprite color 2 |
| $FFFF, $FFFF | = | a line with sprite color 3 |

The colors of the sprites are composed as follows:

| Sprite Nr. | Sprite-COLOR | COLOR-Register | TYPE |
|---|---|---|---|
| 0 & 1 | 0 | $DFF180 | COLOR 00 |
| 0 & 1 | 1 | $DFF1A2 | COLOR 17 |
| 0 & 1 | 2 | $DFF1A4 | COLOR 18 |
| 0 & 1 | 3 | $DFF1A6 | COLOR 19 |
| 2 & 3 | 0 | $DFF180 | COLOR 00 |
| 2 & 3 | 1 | $DFF1AA | COLOR 21 |
| 2 & 3 | 2 | SDFF1AC | COLOR 22 |
| 2 & 3 | 3 | $DFF1AE | COLOR 23 |
| 4 & 5 | 0 | $DFF180 | COLOR 00 |
| 4 & 5 | 1 | $DFF1B2 | COLOR 25 |
| 4 & 5 | 2 | $DFF1B4 | COLOR 26 |
| 4 & 5 | 3 | $DFF1B6 | COLOR 27 |
| 6 & 7 | 0 | $DFF180 | COLOR 00 |
| 6 & 7 | 1 | $DFF1BA | COLOR 29 |
| 6 & 7 | 2 | $DFF1BC | COLOR 30 |
| 6 & 7 | 3 | $DFF1BE | COLOR 31 |

As you see the sprite share the same color register pairs. Furthermore notice that color 0 of all sprites becomes color index 0 (This is the background color of the screen).

An example will show the sprite-color behavior (Let's say that this applies sprite No. 5).
Sprite data: $FFFF, $0000 = a line (16 pixels) in the sprite No. 5 will be the color register 25 ($DFF1B2, color 25). See Figure 3 at the end of this issue.

## SPRITE II

We start this chapter with the setup of the sprite-pointers.

SPRITEPOINTERS:

| Name | Bits | Address |
|---|---|---|
| SPROPTH | 16-31 | $DFF120 |
| SPROPTL | 0-15 | $DFF122 |
| SPR1PTH | 16-31 | $ DFF124 |
| SPR1PTL | 0-15 | $DFF126 |
| SPR2PTH | 16-31 | $DFF128 |
| SPR2PTL | 0-15 | $DFF12A |
| SPR3PTH | 16-31 | $DFF12C |
| SPR3PTL | 0-15 | $DFF12E |
| SPR4PTH | 16-31 | $DFF130 |
| SPR4PTL | 0-15 | $DFF132 |
| SPR5PTH | 16-31 | $DFF134 |
| SPR5PTL | 0-15 | $DFF136 |
| SPR6PTH | 16-31 | $DFF138 |
| SPR6PTL | 0-15 | $DFF13A |
| SPR7PTL | 16-31 | $DFF13C |
| SPR7PTL | 0-15 | $DFF13E |

Look at the example MC0502 while you read the explanations:

| | |
|---|---|
| Line 1: | Disables all interrupts |
| Line 3-4: | Stops engine in all diskette stations. |
| Line 6: | Disable bitplane-, copper- and sprite-DMA. |
| Line 8-17: | Sets up a 256 * 320 pixels display with a bitplane (two colors, 1 + background) up. |
| Lines 19-33: | Adds sprite addresses into copper list. Please note that all sprite data are defined successively in program. In line 32 an offset of 68 is added for each loop. It mages the register Dl pointing to the next start of the sprite data. |
| Lines 35-40: | Adds the address of the bitmap (screen) into the copper-list. |

| | |
|---|---|
| Lines 42-44: | Loads the effective address of our copper-list and moves it into copper-pointer register ($DFF080). |
| Line 45: | Opens bitplane, copper- and sprite-DMA again. |
| Line 48-52: | Waiting (goes into loop) until the electron beam reaches line 0 |
| Line 54: | Jumps to the routine to move the sprites. |
| Lines 56-57: | Check if the left mouse button is pressed. If not jumping back to the label "wait". |
| Line 59: | Closes copper-DMA |
| Line 61-63: | Retrieving the old (previous) COPPER-list address and puts it into copper-pointer. |
| Line 65: | Starts copper-DMA again. |
| Line 67: | Starts all interrupt again. |
| Line 68: | End the program returning to the calling instance (e.g. back to K-Seka or CLI). |
| Line 71: | Here we have defined a word to keep the offset to the table containing Sprite coordinates. |
| Line 73: | Here begins the routine for moving the sprites. |
| Line 74: | adds the address to "move count" (line 71) and puts it in A5. |
| Line 75: | Retrieving the value from the address which is contained in A5. |
| Line 76: | Adds an offset of 4 to the value of the address in A5 |
| Line 78: | Compares the value in D5 with 12000. |
| Line 79: | If D5 is less than 12000 jump op to the label "notend". |
| Line 81: | Put the value 0 in D5(clear D5) |
| Line 82: | Clear the word the address in A5 points to (the move count). |
| Lines 85-86: | Move the value 0 in Dl and D2. |
| Line 87: | Load the effective address of "move table" into the A5. |
| Line 88: | Move the value 15 in D3. |

| | |
|---|---|
| Line 90: | This variation of MOVE, you probably have not seen before. In this instruction D5 will be given as an offset to A5. |

Let us give an example:

```
MOVE.W      10(A1), D1
```

Performs the same as:

```
MOVE.L      #10, D2
MOVE.W      (A1, D2), D1
```

This in turn performs the same as:

```
MOVE.L      #8, D2
MOVE.W      2(A1, D2), D1
```

We will explain this variation in more detail in the “machine code section” in this issue. In any case it loads the value of the address that A5 points to plus the offset from D5, and moves it into D1 (which represents the sprite x-position).

| | |
|---|---|
| Line 91: | Performs the same as the instructions above but also has a fixed offset of 2 and moves the value to D2 (which represents the y-position of the sprite). |
| Line 92: | Loads the effective address of "S8" (sprite data on sprite 8) into the Al. |
| Line 93: | Jumps to the routine "setspr" which sets the new sprite position. |
| Line 95-128: | Repeat the process for sprite 7 to 1. The only thing which is different is the fixed OFFSET in the coordinate table (move table). The result is that we have a snake-like effect (displacement) when the sprite is moved. |
| Line 129: | Jumps back to line 54, and continues with the next instruction (program line 56). |
| Line 131: | This routine updates sprite positions in sprite table of data. |
| Line 132: | This variant of the MOVE is explained later in the “machine code” chapter  of this issue. |
| Line 133: | Adds the constant of $81 to the previous content of D1. In D1 the sprite x-position is stored. We must add $81 (129) to get to the 0-position and to be at the edge of the screen. |

| | |
|---|---|
| Line 134: | Adding $2C (44) to D2 (which contains sprite y-position). We use $2C to get to the 0 position – meaning to be on the top line of the screen. |
| Line 135: | Setting the register D5 to 0. |
| Line 136: | Moving the first byte of D2 into the address A1 points to. |
| Line 137: | Moving (i.e. copy as you know) the content of D2 to D4. |
| Line 138: | This instruction will be explained in machine code chapter. It shifts the content of the register 8 bits to the right so the high-byte is at the lo-byte's place. |
| Line 139: | Shifts the content of D4 2 bits to the left. |
| Line 140: | Adding D4 to D5. |
| Line 141: | Adding D3 to D2. |
| Line 142: | Moving the low-byte (bit 0-7) from D2 to the address where A1 + 2 points to. The value 2, which is added to the address in A2 is called an offset. |
| Line 143: | Copies the whole content of D2 (32bit) to register D4. |
| Line 144: | Shifts the lower 16 bit of D4 for 8 bits to the right. |
| Line 145: | Shifting the lower 16 bit of D4 1 bit to the left. |
| Line 146: | Adds the lower 16 bit of D4 to D5. |
| Line 147: | Moves the whole content of D1 (32 bit) to D3. |
| Line 148: | Performs a logical AND to D1, masking out all bits but the first (bits 1-31 are set to 0). |
| Line 149: | Adds the lower 16 bit of Dl to D5. |
| Line 150: | Moving the low-byte from D5 into the address A1+3 points to. |
| Line 151: | Shifts the content of D3 for 1 bit to the right. |
| Line 152: | Moves the low-byte (bit 0-7) in D3 to the address, A1 +1 points to |
| Line 153: | restore registers d0-d5 from stack (details in chapter on machine code) |
| Line 154: | Go back after the last "bsr setspr" instruction and continue with the next instruction there. |

| | |
|---|---|
| Lines 156-203: | The copper list is defined here. The first sprite pointer starts at program line 157 ($DFF000 + $0120 = $DFF120). Please note that we put color registers to the copper list (program line 181-198), you should be familiar with the rest of the copper list. |
| Line 206: | At this line a block of memory (10240 bytes or 2560 long-words) for the screen (bitmaps) is reserved. |
| Lines 208-359: | Here the data for all sprites are defined one after another. |
| Line 362: | Here we have reserved a block of memory (12400 bytes or 3100 long-words) to keep the coordinates of the sprites. |

To run this example, read the file "SCREEN" into the display buffer (labeled "screen" in the source) and the file "MOVETABLE" into "movetable" buffer as follows:

```
Seka> ri
FILENAME> screen
BEGIN> screen
END> (press RETURN or write -1 (logical END OF FILE))
Seka> ri
FILENAME> movetable
BEGIN> movetable
END> (press enter or return -1 (logical END OF FILE))
```

It is also possible to make your own movement table or "waves". This is done with a program that is on the course disk 1.

Boot from the course disk
Put in a floppy disk you want to store your "wave"-data on, for example to "DF1:" and enter at the command line:

```
1> wavegen DF1:mywave
```

The screen gets black and the machine is waiting for the left mouse button.

After pressing the mouse button the program starts the "recording" of your mouse movements.

Move the mouse around the screen. When you are satisfied with the pattern, press again the mouse button and the recorded movement-data is stored onto disk.

Note that one second of movement, use 200 bytes of your memory.

When you finish this, go into the K-Seka again. Assemble your program and load your own file (df1:mywave) instead to the label "movetable". Please note that you must adjust the value at line 78 in the program to the actual length of your file. It may also be necessary to adjust the buffer size of line 362 (if your file is longer than 12400 bytes).

## MACHINE CODE IV

The first instruction, we must review is a special variant of the move instruction - and it looks like:

```
MOVE.W    (A1, D1), D2
```

This addressing method is called register offset. It works as follows (see example):
The contents of A1 and the content of Dl are added. Neither the content of A1 or D1 is changed in this aggregation. The result of the addition is used as the address to move the data (in this example 16bit – a word) into D2.

It all can also be written in a direct way using the absolute address, namely:

```
MOVE.W    #$10010, D2
```

But because of using tables with data, the exact address is not always known from which the data in the table is to be retrieved. So the register offset is ideal to move through the table and just change the offset. So if we use the register offset the above example it looks like this:

```
MOVE.L    #$100000, A1
MOVE.L    #$10, D1
MOVE.W    (A1, D1), D2
```

This instruction may also the have a solid offset. this can be very handy to have when it comes the use of tables (lists) of data. Try to guess what this instruction performs:

```
MOVE.L    10(A1, D1), D2
```

Did you find out? The explanation is that the address in A1, the value of D1 and the fixed offset are added to form the address the data is moved from to the register D2. (the fixed So: A1 + D1 + 10 forms the effective address without changing neither A1 or D1. This is an addressing method can be very handy. Read this section several times, so you are sure that you understand how it works and what it means.

The next instruction we need to look at may seem a little more complicated at a first glance, but they are quite logical in their functionality and therefore not so difficult to understand, after all. These instructions work on bit level.

```
        LSL and LSR.
```

They mean the Logical Shift Left and Logical Shift Right.

We illustrate the functionality with an example:

```
        MOVE.B      #%00101100, D0
```

D0 (bit 0-7) contains the bit-pattern 00101100 after this instruction is executed.

Then we perform, for example, the following instructions:

```
        LSL.B       #1, D0 (logical shift of one bit to the left)
```

After this instruction is executed, the contents of D0 (bit 0-7) has changed to the following pattern: 01011000 (binary of course).

Notice that only the first byte (bit 0-7) at D0, was affected by the previous instruction. This is because we used the ".B" in the LSL instruction. They other 24 bits (bit 8-32) in D0 will be left unchanged.

Let us take an example:

```
        MOVE.L      #$645A4364, D0
```

D0 will now look as in Figure 1a. (Have a look at the end of this issue).

Then we perform the following instructions:

```
        LSL.W       #1, D0
```

Bit 0-15 will now have changed / moved one bit to the left as shown in Figure 1b (have a look at the end of this issue). Bit 0 will have a "0" coming from the right, while the contents of the bit 15 will "fall out". It does not disappear completely, but lands in carry. This will be explained in more detail in a later issue. The results can be seen in Figure 1c (have a look at the end of this issue).

As you see only the first word (bit 0-15) is shifted one bit to the left. The bits 16-31 remain untouched, because we used the ".W" in the LSL instruction. Shifting in the other direction – to the right (LSR) works the same way. You can also shift the register content for several bits (up to 8 bits) at a time, like this:

```
        LSR    #5, D0
```

This is a way to write shift operations but there are three other variants which will be explained in a later issue.

## BSR – BRANCH TO SUBROUTINE

The next instruction is BSR (branch to subroutine).

This instruction can be compared with the BASIC command GOSUB.

Let us begin with an example:

```
1       MOVE.L      #5, D0
2       BSR         routine
3       RTS
4
5       routine:
6       ADD.L       #1, D0
7       RTS
```

| | |
|---|---|
| Line 1: | Moving the constant value of 5 into D0. |
| Line 2: | Go to the label "routine" an continue to execute instructions |
| Line 6: | Adding the constant value of 1 to D0. |
| Line 7: | Go back after line 2 and continue with the next instruction |
| Line 3: | Exit the program (return to the calling instance) |

If you are unsure of this nonetheless continue to read about the stack - which helps to understand the BSR instruction.

## WHAT IS A STACK?

In the Amiga (and all other computers) there is an area in memory which is used as a temporary storage for data that is needed for program execution. Among other things, it must be able to store addresses when jumping from one point in a program to a another, in order to be able to come back again.

The word "stack" means "pillar / column" or "pile". The last translation describes it best in this case. We must explain how a stack works and how it is used. For explanatory purpose we have created a graph (see Figure 2a, 2b, 2c, 2d and 2e on the pages at the end of this issue). Look at the figures while we explain the term stack and the concept behind it.

A Stack is used to store values / data for a short time during the execution of a program. The processor (MC-68000) uses the stack to store data - data that must be remembered for further processing later in the program.

You may have wondered how the processor can remember which address it has to jump back to where it jumped to the sub-routine. It uses the stack to remember the return address. When the program, for example, needs to jump to a sub-routine, it stores the address of the current location (the return address) at the stack. When the sub-routine is completed, the processor retrieves the previous address from the stack it has stored before it made the jump. This happens quite ("automatic") without your involvement.

You can also use the stack to store data temporarily. However, it is important that you know what you are doing, so that when - or if - the processor must retrieve the address it has stored previously, do not get your data instead of the return address. Your program will inevitably crash if this happens.

To get a visual image of the stack, one can imagine the following:

Imagine the stack as a pipe that is vertical. Down the pipe there is space for a number of plates. At the bottom of the pipe one can imagine that there is a spring which makes the upper plate to be as high at the edge of the pipe. If we put a plate in the pipe, it will be available at the top. If we put another plate in the pipe, it will be placed at the top and hence the previous plate will be available under the new plate.

The more plates we put in the pipe, the more the spring is pressed down and the plates which were put first are the lowest – therefore a stack is also called LIFO – Last In First Out.

Now we try to connect this image with what is happening in the Amiga when the stack is used. Go back to the program example MC0502, and the program line 132 (which looks like this):

```
MOVE.L      D0-D5, -(A7)
```

This instruction will place the content of the registers from D0, D1, D2, D3, D4 and D5 onto the stack.

Let's now have a look at the program line 153:

```
MOVE.L      (A7)+, D0-D5
```

This instruction does exactly the opposite of the previous instruction from line 132. It retrieves the register content from the stack and puts it back to into D0, Dl, D2, D3, D4 and D5 again.

What is the reason for this? It is because the registers are used in the program routine in a way that the values in the registers are changed, so when the processor jumps back to the main routine again (program line 47-57), it is necessary to save the previous values. This is done at the beginning of the routine and at the end of the routine those values are restored before the jump back.

Let us now see what is happening in the Amiga when the stack is used:

Address register A7 is also called stack-pointer. Figure 2a at the end of the issue is a draft od a stack. The numbers on the left side of the diagram represent the addresses of the stack--memory, while the arrow (SP) on the right side represents the stack-pointer. Pointing to the address 1000 (A7 contains the value 1000). Imagine now these instructions are executed:

```
MOVE.L      #10, D0
MOVE.B      D0, -(A7)
```

The stack will now look like in Figure 2b.

You've already learned about the "(A7)+" functionality (See NOTE III) but you have not learned anything about "-(A7)". It works in a slightly different way than the "+" because the minus sign stands in front of the brackets and the "+" is placed after the brackets. The "-(A7)" part of the instruction causes the processor to decrease the address in A7 about 1,2 or 4 bytes (depending whether you specified ".B", ".W" or ".L"). Then the content of the address, which is held by register A7 is retrieved.

The value 10 (decimal) is moved into D0. The address in A7 is decreased by 1 and the content of D0 (in this example the lowest byte of the register) is stored to the address A7 points to.

```
MOVE.L      #25, D0
MOVE.B      D0, -(A7)
```

After these instructions the stack will look like depicted in Figure 2c. And so we continue:

```
        CLR.L       D0
        MOVE.B      (A7)+, D0
```

After these instructions the stack will look like in Figure 2d. The register D0 is set to 0 (CLR) and one byte is restored to register D0 from the address A7 points and the address is increased about 1 afterwards. D0 now contains 25 (decimal). Then we perform following instructions:

```
        MOVE.B      (A7)+, D0
```

After this instruction the stack will look like in Figure 2e and D0 will now contain the value 10 (decimal).

## MORE ON BSR

We will now explain BSR in more detail. It also uses the stack, and as mentioned earlier this is done automatically.

```
        BSR routine
        .....
        RTS

        routine:
        .....
        RTS
```

When the processor performs a BSR, the address the program counter points to (PC, see issue II) is saved onto the stack, then the processor branched "routine". When the processor encounters an RTS instruction at the end of the "routine" the processor will load the previously stored address from back into the program counter.

Imagine that we ran this program from K-Seka:

```
        MOVE.L      #5, D0
        RTS
```

First we assemble the program.

```
Seka>  a
OPTIONS>    <RETURN>
No Errors
Seka>
```

Then we start it.

```
Seka> j
```

Before K-Seka jump to the beginning of your program, it puts the current program counter address (a position in the K-Seka program) onto the stack. Then the processor branches to your program.

Once your application is complete (it ends with an RTS), the processor acts in the same way as explained above, the stored value of the program counter is retrieved from the stack, loaded into the program counter (PC) and you end up back in the K-Seka.

**NOTA BENE**

It is important that you understand how the stack works – this principle applies to all computers. In the Amiga operating system the stack size is usually set to 4000 bytes but you can increase this value through the stack-command in the CLI.

Read the sections on stack over and over again until you are sure you understand the principle. We promise you it is worth it.

## SOLUTIONS FOR THE TASKS IN ISSUE IV

Task 0402: Information to display line number 4 starts at address $010078

## TASKS OF ISSUE V

Task 0501: What position (x and y) and height have a sprite with these position data: $EEB5, $1103

Problem 0502: How wide can be a sprite?

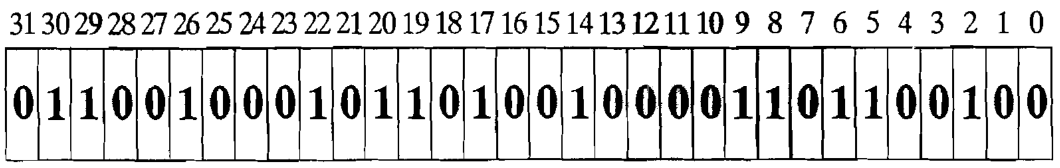
Problem 0503: How many sprites are available at the Amiga?

Task 0504: D0 = 01100101101001101011011110010111. How D0 will look like after a "LSR.B #3, D0"?

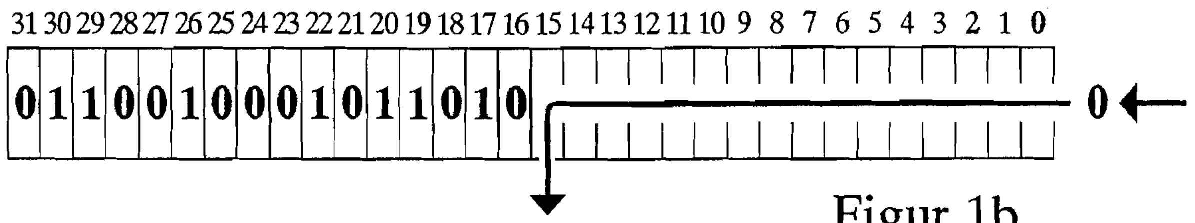
Task 0505: From which address the content of D0 is copied from in this program example:

```
MOVE.L    #$150, D1
MOVE.L    #1010, A1
MOVE.B    18(A1, D1), D0
RTS
```

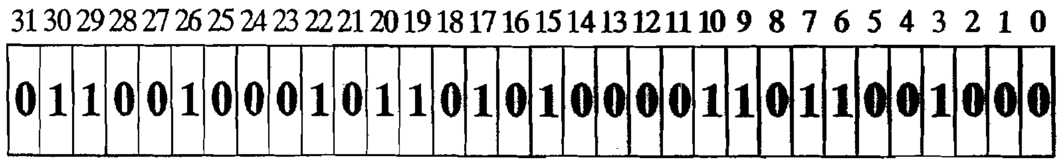
Task 0506: What is a stack used for in the Amiga?



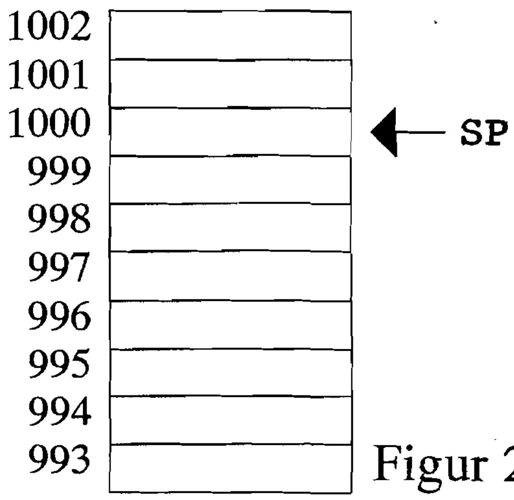
Figur 1a.



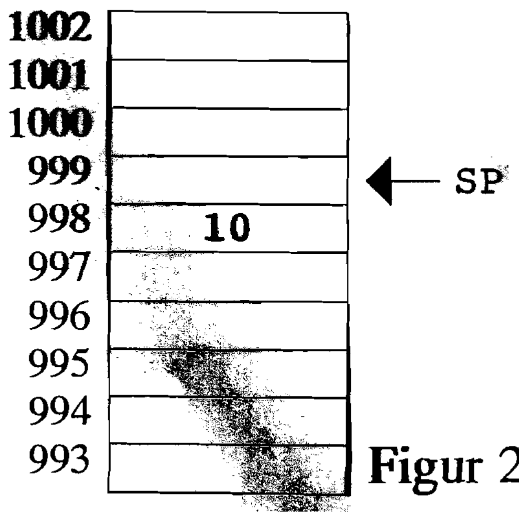
Figur 1b.



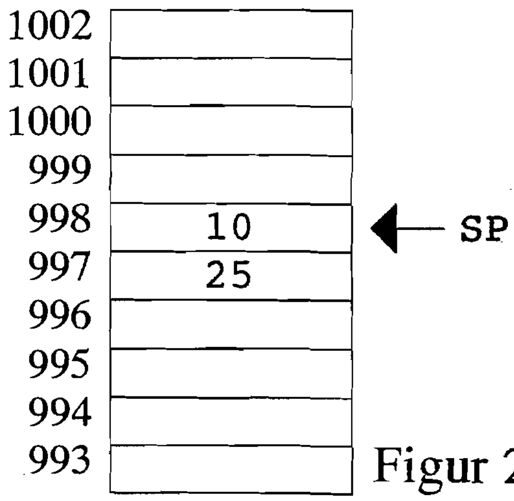
Figur 1c.



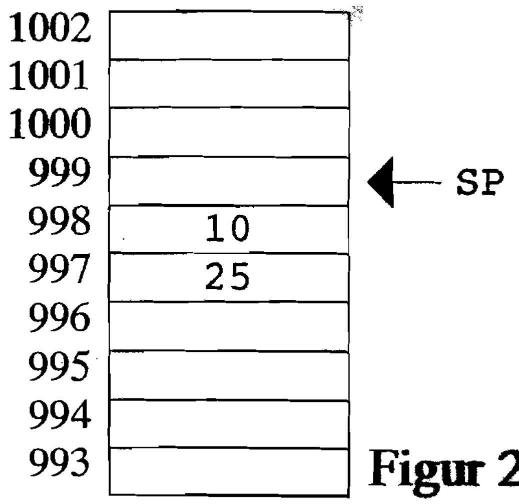
Figur 2a.



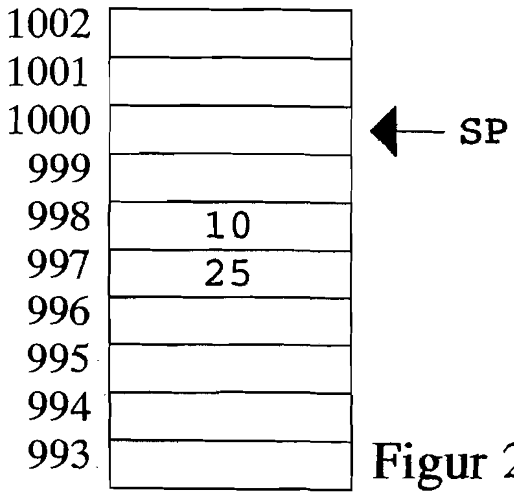
Figur 2b.



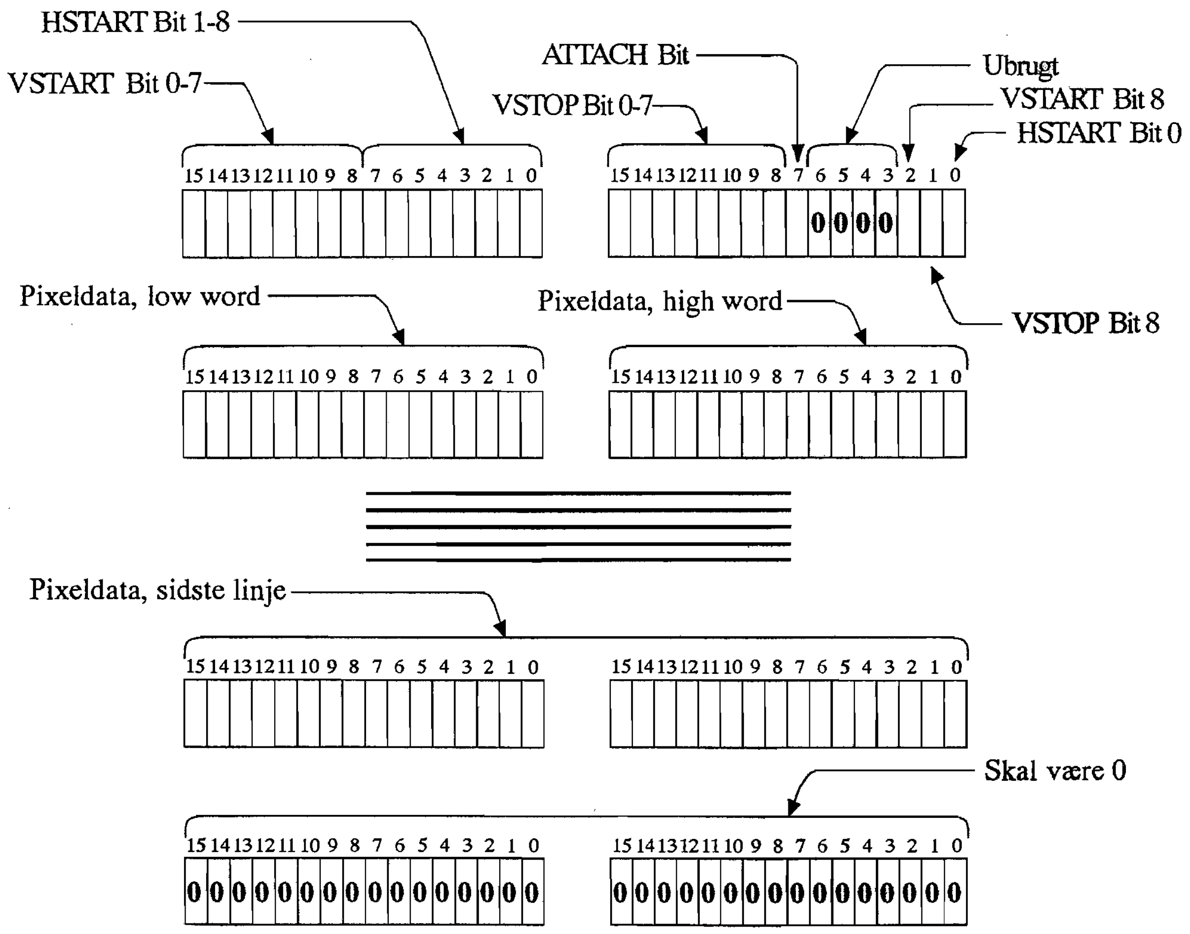
Figur 2c.



Figur 2d.



Figur 2e.


HSTART Bit 1-8
VSTART Bit 0-7
ATTACH Bit
VSTOP Bit 0-7
Ubrugt
VSTART Bit 8
HSTART Bit 0
VSTOP Bit 8
Pixeldata, low word
Pixeldata, high word
Pixeldata, sidste linje
Skal være 0

Figur 3.