Programming in machine code on the Amiga

A. Forness & N. A. Holten

Issue 12

Content:

DATA SCHOOL
Postbox 62
Nordengen 18
2980 Kokkedal

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

INTRODUCTION

You have now reached chapter 12 - the last chapter in this course. On the following pages we will deal with some TIPS & TRICKS. You will see how you can use the things you have learned in the previous chapters. We will not explain the programs instruction by instruction, but only explain the individual sub-routines.
When you finish this chapter, you are done with the whole course. Of course there is much more literature on the Amiga, which may be useful to study. The famous "Hardware Manual" from Commodore is often purchased first. This book contains everything about the hardware basis for Amiga, which includes, e.g. all hardware registers (which begins with $DFFxxx), bitplanes, blitts, disk, serial port, parallel port, etc. Notice that the “Hardware Manual” is a reference book for Amiga. Furthermore, you can get the entire series of reference books from Commodore. At the time of writing there are four books:

1. Hardware Reference Manual
2. Intuition Reference Manual
3. Exec Reference Manual
4. Libraries & Devices Reference Manual

With these books you're sure to get enough information to solve all your programming problems - but remember that they are reference books (encyclopedias).

There are many textbooks dealing with AMIGA programming. At this point, it is difficult for us to make some recommendations because there are constantly appearing new books on the subject.

Actually, we do not believe that you need more books than this course plus the four books mentioned above.

HAM, HIRES AND INTERLACE

In this chapter we will look at the following program examples (found on Disk 2):

- HAM
- HiRes
- Interlace

These program examples have in common that they set up a screen. First, let us examine the HAM-example. The HAM-examples can be found on Disk 2 in DIRECTORY "HAM". There are following files:

HAM.S	=	source code
HAM	=	a pre-assembled (executable) version
SCREEN	=	the file containing the image

The image is 320*256 pixels. It has 6 BITPLANES and use the first 16 color registers. Remember that a HAM-image almost always have 6 BITPLANES (it is possible to use 5). As you know, a HAM image shows 4096 (all) colors. Let us take a closer look at how this works.

The abbreviation HAM stands for HOLD AND MODIFY. Let us take an example:
Imagine the first pixel (top left corner) at the screen is red. In the AMIGA's palette system every basic color can have a value between 0 and 15. This red pixel has the following values:

RED = 15, GREEN = 0 and Blue = 0. In HAM mode, next pixel’s (the pixel to the right of the first) color often depends on the previous pixel. It is possible to set pixel number 2 to orange without using the color registers. To make the transition from red to orange, do the following:

The AMIGA reads the color data for the red pixel in a buffer register. Then only the color value for green is changed, for example. 8. The color values for red and blue remain unchanged. We now have a new color combination that looks as follows:
RED = 15, GREEN =8 and BLUE = 0. Let's see how they 6 bitplanes are exploited:

BITPLANE 1-4 = Color intensity (0-15) or color register.
BITPLANE 5-6 = Adjustment Options.

BIT COMBINATIONS

BIT 5	BIT 6		
0	0	=	The colors of related pixels are retrieved directly from a color register. The color is determined through bitplanes 1-4. This is the same as it happens in, e.g., common lores.
0	1	=	Here, the color of the previous PIXEL is used. The BLUE value is defined through the bit combination from the bitplanes 1-4.
1	0	=	Here, the color of the previous PIXEL is used. The RED value is defined through the bit combination from the bitplanes 1-4.
1	1	=	Here, the color of the previous PIXEL is used. The BLUE value is defined thorough the bit combination from the bitplanes 1-4.

In this way the AMIGA is able to display 4096 colors at once. The disadvantage of HAM-mode is that you can only change one of the three basic colors at a time. So you cannot put a red pixels next of a green. Therefore, all the HAM-images have "blurry" color transitions.

To start the HAM example, first assemble it, of course. Then you'll load "SCREEN" file, located in the same DIRECTORY. The example needs no further explanation, because it is very similar to the other examples, which sets up a screen. It is important that you understand the functioning of the HAM mode.

Let us now move directly to the next program example that is called HiRes. The abbreviation stands for: High Resolution. In the directory "HiRes" on disk 2 you find the following files:

HIRES.S	=	the source code.
HiRes	=	the executable of the example
SCREEN	=	the image

The difference between the Lores and HiRes example is that the horizontal resolution in the latter example is twice as high as in the lores example. The default size of a lores display is 320*256. In HiRes the resolution is 640*256.

An example of a HiRes screen is the Workbench screen. It is HiRes, 640*256 with 2 bitplanes (or 4 colors). The restriction for HiRes mode compared to LoRes is that you cannot have more than 16 colors on screen (4 BITPLANES) and that you cannot use the HAM mode. We can also mention that HiRes is rarely used in games and the demos.

To start the HiRes example, you must load the file "SCREEN" (which contains the image). It is in the same directory like the source. Notice that both the HAM-image above and this HiRes image are digitized with a video camera (they are definitely not drawn by hand). The explanation of the HiRes example is also necessary.

Now we come to the last topic in this section, namely Interlace. You can find this program example in the directory "Interlace" which contains these files:

LACE.S	=	source code
LACE	=	an executable version
SCREEN.IFF	=	image in IFF format
SCREEN	=	image in "raw" format

The interlace mode allows you to double the vertical resolution. In other words: Twice as many lines on the screen. In HiRes + Interlace you have a default resolution of 640*512 pixels. You've probably seen that the image "flickers" when AMIGA is in interlace mode. This is because the image on the screen is only updated 25 times per second. Usually it is updated 50 times per second. Why draws the AMIGA the image only 25 times per second in interlace?

Actually the Amiga draws the image 50 times per second, but draws 2 different parts of the image every time. Confusing? Let us try to clarify it:

First draw line 1 of the image, then draw line 3, then line 5, and so it goes with every other display line right down to the line of 355. The AMIGA then starts from the top again displaying line 2, line 4, line 6, etc. This results in that you see twice as many lines on the screen. Interlace mode can be used in combination with HAM or HiRes.

This program sets up an example Interlaced + HiRes screen – which has 640*512 pixel. The image has one (1) BIT-PLANE (2 colors).

To start the program, you must first assemble the code, and then load the file "SCREEN" found in the same directory.

WAVE

The "wave" effect is often used in demos. It is an effect, which will have the image on the screen to "meander" in horizontal direction. The AMIGA has a hardware register – the so called SCROLL register. This register has address $DFF102. The setup for this register looks like:

SCROLL $DFF102 (write only)

BIT No.	Function
0-3	scroll value for bitplane 1, 3 and 5
4-7	scroll value for bitplane 2,.4 and 6
8-15	not used and set to 0

With this register you can then slide the screen 0-15 pixels. In order to get wave effect make use of the copper an write new values in the register for each line. We can individually shift the lines independent of each other.

In this program example, we have made a small sinus-table which gives a "natural" wave motion. The program sets up a LoRes display with 320*256 pixels resolution and 2 bitplanes.

For the wave effect we manipulate the lines from line 30 (top of the screen) down to the line 230. The data in the sine table is continuously fed into the copper list to provide a constant wave motion.

Especially notice, that a portion of Copper-list becomes generated by the program itself.

To try this program example, you find it in directory "WAVE" at course disk 2. There are following files:

WAVE.S	=	source code
WAVE	=	the executable example
SCREEN.IFF	=	image stored in IFF format
SCREEN	=	image in "raw" format
SIN	=	sinus table

To start the program, you must assemble it as usual, then load the file "SCREEN" and the "sine" file, located in same directory to its location in the code. You can of course try to change the wave speed. This is done by changing the numbers on program line 68
Enjoy!

ROTATING AN IMAGE

To get a picture to rotate seems to many as a "cumbersome" task. Actually it is not that complicated. Given that the screen is flat (2 dimensions), it is only a matter of a “compression”. If you press the picture together and pull it out in a sinus (circular) motion around the center of the screen, it will look as if the image rotates.

This can be done with the copper. What you are doing, when you compacts the image, is that you just skip some lines above and below the center. If you jump over every second line the image will only be half as high. So you start with all lines visible and the image upright. Then for instance skip 1 line per 100 lines. When you do this in a smooth sinus way, it will look as if it rotates.

The program example is on the course disk 2 in directory "ROTATE" and it contains these files:

ROT.S	=	source code
ROT	=	executable
SCREEN.IFF	=	image in IFF format
SCREEN	=	image in "raw" format
SIN	=	sinus table

In order to run this program example, after you assembled the source, load the files "SCREEN" and "Sin". To change the rotation speed, change the numbers on the program line 15. Just to be complete we note that the picture is Lores, 320*256, 3 bitplanes or 8 colors.

DEMO

In this section we look more closely at how to set up a DEMO.
On Disk 2 in the directory "DEMO" you find these files:

DEMO1.S	=	source code set up a HAM-image
DEMO2.S	=	source code adds an additional screen
DEMO3.S	=	source code includes scroll
DEMO4.S	=	source code the finished DEMO
DEMO4	=	an executable version of the finished DEMO
SCREEN.IFF	=	HAM-image in IFF format
SCREEN	=	image in "raw" format
FONT.IFF	=	font in IFF format
FONT	=	font pre-converted
SPR.IFF	=	sprite in IFF format
SPR	=	sprite finished in "raw" format
MOVETAB	=	a table containing motion data for the sprites

Application examples Demo1, DEMO2, DEMO3 and DEMO4 show step by step how the new effects are composed for a demo.

DEMO 1

Here we begin by setting up the HAM-image which is 320*230 pixel large. We have sacrificed some of the height, so that we have space for a scroll-text at the bottom of the screen.

Line 1-2: Switch off Interrupts, bitplane-DMA, copper-DMA AND sprite-DMA.

Line 4-10: Sets 16 colors into the copper-list. We let the colors be updated by copper because we must change some of the colors further down the screen.

Line 12-23: Sets the addresses of the individual bitplanes into the copper-list.

Line 25-28: Sets the address of the COPPER-list and start bitplane-DMA, copper-DMA and sprite-DMA.

Line 31-39: Waiting for pressing the mouse button. Then reset the Workbench screen and terminate the program.

Line 42-91: The declaration of the copper-list.

Line 94: The declaration of screen memory.

DEMO 2

The next step is to configure a second display at the bottom, for the scroll-text. This part of the screen is 320*32 pixels wide with one bitplane, and 4 bytes in modulo. We only explain only the new parts of the source:

Line 25-30: Here the address of the new screen is inserted into copper-list.

Line 99-112: That we have paid for COPPER-list to program will display the new screen. We change here modulo to 4 and one bitplane in LoRes.

Line 115: Here we have reserved the screen memory of the scroll area with the following size: 44*32 = 1408 bytes

DEMO 3

We continue to put a font in and show a moving SCROLL-text moving. Here is an explanation for the new program lines:

Line 38-42: Waiting for the electron beam that it reaches line 100.

Line 44: Branch to sub-routine “scroll” which updates the text.

Line 56-122: Here is the sub-routine to move the scroll-text. This a routine was already explained and should be clear by now.

Line 192-250: Declaration of the conversion table for the scroll routine. Notice that we use both x- and y-position, to find out where the character in the font is located.

Line 253: Here we reserve memory for the font - as in this demo the font-table is an image of 320 * 128 pixels (1 plane = 5120 byte). One character is 24 * 32 pixels, and there are 13 characters on a line. So we have 13 * 4 = 52 different characters in the font.

Line 262: Here the text is declared. The legal character are:

A B C D E F G H I J K L M N P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 ! ( ) . ,: ; = + / - * <> '

Notice that the text must end with the 255.

DEMO 4

Here, we put the finishing touch by creating a “sprite-snake” -routine. This “snake” consists of eight sprites. Following the explanation:

Line 32-43: Here the addresses of the sprites are copied into the copper-list.

Line 59-64: This routine waits for the electron beam to pass line 280

Line 66: Branch to the routine for moving the sprites on the screen.

Line 79-129: The routine which moves the “sprite-snake” on the display.

Line 200-216: The copper-list where the sprite-pointers are set.

Line 229-240: The sprite colors are set.

Line 359: Here the block of memory is reserved for the table of motion-data for the for the sprites.

Line 362: Here the block of memory is reserved where the sprite data is stored. Each sprite is 16 pixels high and uses 72 bytes.

We have now created a simple DEMO. To run this program (DEMO4), you must load these files: SCREEN, FONT, SPR and MOVETAB.

If you are making your own font, do the following:

Start up a drawing program (e.g. Deluxe Paint) in Lores 320 * 256 with a BITPLANE (two colors). First character must be in the top left corner and the size the character must have a width of 24 pixels and a height of 32 pixels. You will then have 13 characters per. line. The next line begins after the 32th line – the font can be up to four character lines. Here comes the placement of signs:

A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9! ( )
. , : ; = + / - * < > '

Once you have finished your font-design it must be converted. Start iff-converter and load the font. Select “No cmap” and then store the font with “Frame + sawing”. You must cut off position (0,0) to (320128). The font is now ready for use.
Good luck!

DISK

In this paragraph we look more closely at how to "handle" the disk. The DIRECTORY "DISK", contains the following files:

DISK.S = the source code.

The program is divided into different routines:

DRIVEON	DRIVEOFF
WRITEDISK	READ DISK
TRACK	TRACK00
CODEMFM	DECODEMFM

With these routines you can create a "track loader". The routines DRIVEON and DRIVEOFF used to start and stop a specified drive.

WRITEDISK-routine writes MFM data to disk. MFM is the data format the AMlGA disk system uses. For instance, if you have 1000 bytes of data to be stored on disk, these are first converted to MFM format. The encoding to MFM data doubles the size of the original data, to a size of 2000 bytes. The routine READDISK is used to load MFM data from the disk again.

The TRACK routine is used to move the read/write heads to a specific track while the TRACK00-routine is used to reset the read/write heads. This zero (0) position is used as a reference point for the read/write heads so that the other tracks can be found.

The routine CODEMFM is used to encode the data information into the MFM format, while the DECODEMFM-routine does the opposite – to decode the MFM data to the original data again
.
The Amiga floppy disk has 80 tracks on each side of the disk - in total 160 tracks. Such a track can theoretically store 12500 MFM data bytes or 6,250 bytes of ordinary (real) data. In practice, you can store 6000 bytes of raw data on each track. The total storage capacity is then 6000 * 160 = 960,000 bytes = 937.5 KB.

The procedure to write data to a track on a disk are:
First start the floppy motor with the routine DRIVEON. In the register D0 you specify which drive you want to use:

```
moveq     #?, D0      values in D0 can be 0-3 (0 = DF0:, etc.)
BSR       driveon
```

Then you jump to the routine TRACK00. This routine needs no parameter.

```
BSR       track00
```

Now you put your data into "diskbuf". There can be no more than 6000 bytes in the buffer.

Then you jump to the routine CODEMFM. This routine will encode the data in "diskbuf" to the MFM format. This routine does not need any parameters.

```
BSR       codemfm
```

Now select the track to which the data is to be written. This is done with the TRACK routine. You specify in D0 which tracks you want to move to.

```
moveq     #?, D0      values in D0 can be 0-159 (track number)
BSR       track
```

Finally the actual writing of the data to the disk is started with the routine WRITEDISK.

```
BSR       writedisk
```

If you must write to multiple tracks simultaneously, you call routines CODEMFM, TRACK and WRITEDISK consecutively. You do not branch to DRIVEON or TRACK00.When you're finished writing, then stop the floppy motor with DRIVEOFF. You indicate in D0, which drive is to be stopped:

```
moveq       #?, D0          values in D0 can be 0-3 (0 = DF0:, etc.)
BSR         driveoff
```

If you want to read a track from the disk, the procedure is as follows:
Start up the floppy motor calling DRIVEON the same way as you did writing to disk.

Jump to TRACK00 to reset the heads position to zero.

Write track to be read into the register D0 and jump to the TRACK routine.
Jump to the READDISK-routine to load the MFM data from disk. Then decode the MFM data calling the DECODEMFM-routine. This routine does not need any parameters.

The data is now ready to "pick up" in "diskbuf". If you must read multiple tracks simultaneously, simply repeat calling the routines TRACK, READDISK and DECODEMFM consecutively. When you are finished reading, stop the drvie motor calling DRIVEOFF.

You are now ready to try your way with "trackloading". We will just point out one very important thing before you start:

DO NOT USE ORIGINAL AMIGA PROGRAM DISKS. IF YOU USE THESE ROUTINES TO WRITE (The READ routine is not the problem) TO A AMIGA-FORMATTED DISK, WHICH ALREADY CONTAINS DATA AND/OR PROGRAMS YOU WILL MOST LIKELY LOSE YOUR DATA!

STARS

This chapter deals with the famous "star sky". This program-example animates 256 "stars" (pixels). The effect looks like as if you move out into space at a terrible speed. The "stars" flow from the middle of the screen, and looks as though they move towards you.

This effect is quite simple to program. You have a table (STARS) - which contains a series of random numbers, indicating the starting position and exit angle for each pixel. There is also pre-calculated a sine/cosine table (SIN) to determine the pixel rate.

"Stars" must have a slow speed in the middle of the screen to look natural, and then increase their speed towards the edge of the screen. For this we have created an "acceleration"- table (ACC).

At last we have a routine that still rotates the entire "star cluster".
This is done by rotating the exit angle of all "stars".

The program example is available on DISK02 in DIRECTORY "STAR" which contains these files:

STAR.S = SOURCE code.
STAR = An executable version.
SIN = sine / cosine table.
ACC = Acceleration table.
STARS = Positions table.

To start the program example you must load the data-table files: SIN, ACC and STARS into their reserved blocks of memory.

You can of course experiment and try to change the rotation speed and the direction. This is done by changing the number of the line 59 in the program. To change direction, you can use "subq.w" instead of "addq.w".

Have fun - drive yourself crazy!

LINEDRAW

We must now look a little closer to line drawing on the Amiga. On DISC2 in directory "linedraw" you will find these files:

LINEDRAW.S	=	SOURCE code for line drawing.
LINETEST.S	=	SOURCE code for the test program.
LINE TEST	=	An executable file of the test program.

Let us begin with the “LINE DRAW”-routine. We do not go into detail, how this works, but just to show how to use it.

Let's say that you have set up a Lores screen with 320 * 256 pixels and one bitplane. The width of the screen is then 40 bytes. In the first line of the program you set the screen-width variable "swid to the value of 40.

The next step is to jump to the "initlinedraw" routine, which sets up the Blitter and puts the screen address and the octant address as well as the base of the custom-chips into the address register A0-A2.

Now you are ready to draw. You specify the first position on the display in the D0 and D1 (X and Y) and the second position in D2 and D3 (X; Y). Then you jump to "line draw" and draw a line between the two specified positions.

Remember that if you use the blitter for anything else than the line-drawing, you must jump to "initlinedraw" routine before you can draw the next line. This also applies if you've used A0, A1 or A2 for something else meanwhile. The speed of the line routine is about 94 µS (microseconds) for an "average long line" (cannot be said more inaccurately…).

The next program routine - called "LINE TEST" – sets up a screen up and lets you draw lines by moving the mouse. You can also see the SOURCE code for this program and how to and call the "line draw" routine.

Finally, we just mention that if you try to draw a line from the same position (point), so you will not get any results.

VECTOR

In this chapter, we look closely at VECTOR animation. The word VECTOR actually means "line between two points". At the Course floppy I DIRECTORY "VECTOR", the following files:

VEC1.S	=	SOURCE code for example 1
VEC1	=	The executable file.
VEC2.S	=	SOURCE code for example 2
VEC2	=	The executable for the second example
SIN	=	The sine/cosine table.

If we are to enter mathematics in these program examples it would fill a book. We concentrate therefore only about how to use the routines.

When you define an object, you define the position of each point in the object. As an example, a cube has eight points, one in each corner. What makes it so interesting is that you specify both X, Y and Z position for each point. You are defining three-dimensional (3D) objects. When the object is displayed on the screen, it is converted to a two-dimensional (2D) image keeping the perspective for the sake of an effect of depth.

In the program line 229 the points of the object are defined. It has following setup:

```
vectors:
dc.w        X, Y, Z (point 1)
dc.w        X, Y, Z (point 2)
... etc.
```

Remember that the position 0 is the rotation point, so that you can specify both positive and negative values. Values can range from -10,000 to + 10000. You can use larger values, but then you have the problem that the object is sticking out of the screen. When you have defined all the vectors you set the number of points in the "PNR" variable at the beginning of the program in line 2.

Now it's your turn to decide between which points lines are to be drawn. This is indicated at program line 239 as follows: If you need to draw lines between, for example point 1-2, 2-3 and 3-1, do this:

```
lines:
dc.w        0,1,1,2,2,0
```

Then set your number of lines (in this case 3) in variable "LNR" at the beginning of the program at line 1.

The example uses the file "SIN". It is possible to change the speed of rotation in all three axes. This is done by changing values in the program at lines 31, 32 and 33. These values determine respectively the X, Y and Z rotation.

You will probably notice that you cannot define infinite complex objects. Objects with little more than 20 points and 50 lines can be difficult to get to display. You can will see that you parts of the object are not displayed or that it will start to flicker.

MISCELLANEOUS

In this paragraph we deal with - as the title already suggests – with miscellaneous topics. On the course disk in directory "diverse" you find the following files:

READMOUSE.S
GETWB.S
INITSCREEN.S
SQR.S

Let's start with READMOUSE routine. This routine is used to read the mouse position. It is almost self explanatory, but we explain the use anyway. When you need to put this routine into your program, you must jump to the routine "initmouse at top of your program. This makes resets the mousecounters. Then you jump to the routine in screen update.

When you want to read the position, you can:

```
lea.l       mousexy, a1
move.w      (a1), d1        (X-Pos)
move.w      2(a1), d2       (Y-pos)
```

It is also possible to set minimum and maximum values at the top of the routine (rm_xmin, rm_xmax, rm_ymin, rm_ymax).

In the next routine we will look at are GETWB. This routine retrieves information about the Workbench screen (address, height, etc.) and returns the address of this setup in A0:

OFFSET	LENGTH	CONTENT
0	WORD	Screen Widths in Bytes
2	WORD	Screen height in pixels (lines)
4	WORD	Number BITPLANEs
6	WORD	Not used
8	LONG	Address BITPLANE 1
12	LONG	Address BITPLANE 2
16	LONG	Address BITPLANE 3
20	LONG	Address BITPLANE 4

```
Example:
move.w      4(a0), d0       (D0 = No. BITPLANEs)
...
move.l      8(a0), d1       (D1 = Address BITPLANE 1)
...
```

This routine should be easy enough so that no more explanation is needed. It may be useful to have the address of the Workbench screen when you do not "bother" to configure your own screen to test something.

The next example program, named "INITSCREEN" sets up a Lores, 320 * 256 pixels screen with one BITPLANE. You can use this program as a starting point when you need to do / test anything new. The example requires no special explanation.

The last program in this paragraph is more for testing: a "sqr" routine. This routine calculates a square root of numbers in D0. The result is also returned in D0.
Note that this routine only gives integer response.

EPILOG

This was the last chapter in our course on how to program the Amiga in machine code. As you have seen not all topics could be covered, but we tried to offer you as much as possible. From here it should not be too difficult to continue and improve on your own.

We recommend to obtain the so-called "Amiga Bibles" - as we mentioned in the introduction in this chapter, they are rather expensive, but you'll get far with these four books (Amiga Rom Kernel Reference Manuals). We used the HARDWARE REFERENCE MANUAL when we took our first faltering steps into the unknown "wilderness" called Amiga. It therefore should be also a good tool for you.

Should anyone have difficulties obtaining these books, you can contact us on 49 18 00 77 and we will try to help you to get your copies. When you have completed this course (and of course know everything by heart), then you're probably want to get a proof that you have passed this course. And as we mentioned long ago, you will definitely get it. After having reviewed this chapter, you can send us a small sample of your programming skills (at no extra cost) which will be reviewed by us.

After your sample of skills has passed the review we will send you a diploma as a "proof" of your programming skills. We have written the word PROOF in quotation marks. The reason is that we cannot verify that you alone have created the sample. This test gives you a hint where your strengths and weaknesses are.

If you one day are looking for a job within the software development sector, then this diploma and possibly a DEMO is a nice way to present yourself and your knowledge.

We use this opportunity to thank you for have attended this course. Keep your skills up to date, and when we publish more depth/special courses within machine code programming, we'll maybe be in contact again.

We wish you luck in the future. Yours sincerely,

DATA SCHOOL
Carsten Nordenhof