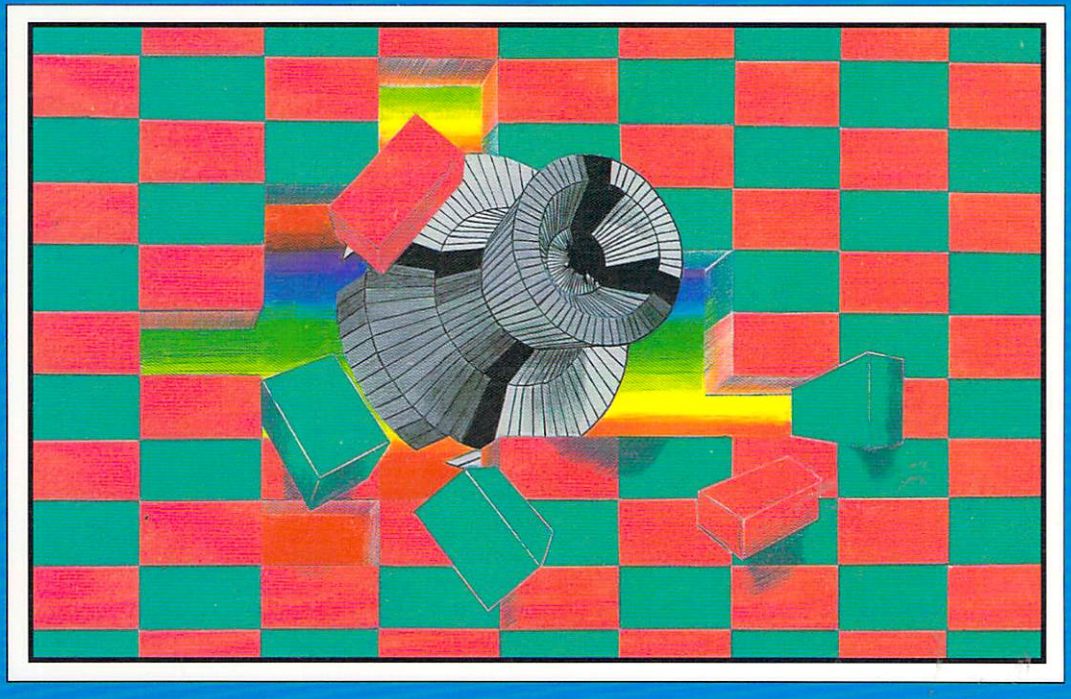
Amiga® Graphics Inside & Out

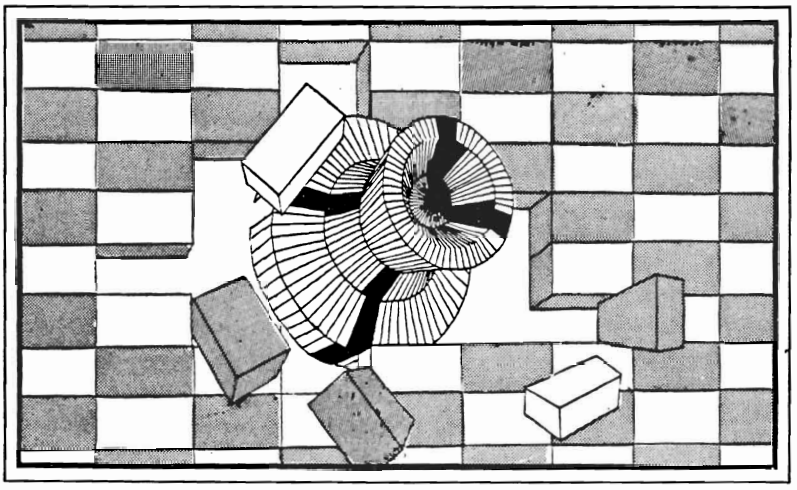
A comprehensive book for understanding and using Amiga graphics



Abacus
A Data Becker Book

Amiga Graphics Inside & Out

Weltner
Trapp
Jennrich



A Data Becker Book
Published by
Abacus

First Printing, July 1989
Printed in U.S.A.

Data Becker, GmbH
Merowingerstrasse 30
4000 Deusseldorf, West Germany

Abacus
5370 52nd Street SE
Grand Rapids, MI 49512

ISBN 0-55755-052-2

Table of Contents

Introduction		ix
1.	Amiga graphics	1
1.1	Seting pixels with PSET	1
1.1.1	Using the mouse and PSET: a simple drawing board	1
1.1.2	Rosette and garland patterns	2
1.1.3	Erasing pixels	3
1.1.4	Adding color to your screen	4
1.1.5	More about PSET and PRESET	6
1.1.6	The "opposite" of PSET: the POINT statement	7
1.1.7	Relative addressing	8
1.2	The LINE statement	10
1.2.1	The power of LINE	10
1.2.2	The Moire effect	10
1.2.3	Quix, lines galore	11
1.2.4	Function plotter	12
1.2.4.1	Function plotter modes and menu controls	15
1.2.4.2	Undefined function values	16
1.2.4.3	Scaling	16
1.2.5	Drawing rectangles	17
1.2.6	Relative addressing with LINE statement	18
1.3	The CIRCLE statement	20
1.3.1	The aspect ratio	20
1.3.1.1	Animation using CIRCLE	21
1.3.1.2	The aspect ratio in circle formulas	23
1.3.2	Angle parameters with CIRCLE	24
1.3.3	Relative addressing with CIRCLE	25
1.3.4	Pie charts	25
1.3.5	Pixels and lines with CIRCLE	27
1.4	Area fill	30
1.4.1	The PAINT statement	30
1.4.2	Another solution: AREA and AREAFILL	31
1.4.2.1	Different modes of AREAFILL	33
1.4.3	Patterns	34
1.4.3.1	Pattern creation	34
1.4.3.2	Patterned areas	35
1.4.3.3	Pattern design in a program	36
1.4.3.4	Patterns and the cursor	38
1.4.3.5	Bringing it together	39
1.5	Medley of colors	41
1.5.1	The complete palette	42
1.5.2	Changing RGB colors	42

1.5.3	The opposite of PALETTE	43
1.5.4	Animation using color	44
1.6	All about PUT and GET	46
1.6.1	Using PUT and GET	46
1.6.2	Saving to disk	48
1.6.3	Alternate uses for PUT	50
1.6.3.1	Default mode of PUT	50
1.6.3.2	A direct method using PSET	52
1.6.3.3	Inverting graphics	52
1.6.3.4	AND or OR	53
1.7	Animation in BASIC	55
1.7.1	Sprites and bobs	55
1.7.2	The OBJECT.SHAPE statement	56
1.7.3	Designing an object	56
1.7.4	The complete object editor: Eddi II	56
1.7.4.1	The screen	68
1.7.4.2	Object size	68
1.7.4.3	Depth	69
1.7.4.4	Colors	69
1.7.4.5	Drawing	69
1.7.4.6	Loading and saving	71
1.7.4.7	Testing objects	71
1.7.4.8	Exiting Eddi II	71
1.7.4.9	Loading objects into your programs	72
1.7.5	Flags	72
1.7.5.1	The SaveBack flag	72
1.7.5.2	SaveBob	75
1.7.5.3	Overlay	75
1.7.5.4	The Shadow mask	75
1.7.5.5	Collisions	76
1.7.5.6	Animated bit-planes	80
1.7.6	The alternative: sprites	85
1.7.6.1	The small difference	85
1.7.6.2	Color and sprites	85
2.	The Amiga operating system	87
3.	Intuition - the user interface	93
3.1	Intuition windows	93
3.2	The window data structure	96
3.3	Functions of the Intuition library	106
3.3.1	A personalized mouse pointer	106
3.3.2	Moving windows made easy	109
3.3.3	Setting window limits	110
3.3.4	Sizing windows	111
3.3.5	WindowToFront and WindowToBack	113

3.4	The Intuition screen	114
3.4.1	Screen structure	115
3.4.2	The Intuition functions for screen handling	118
3.5	Intuition and the rest of the world	120
3.6	The RastPort	122
3.6.1	RastPort data structure	123
3.7	Graphic primitives	129
3.7.1	Multicolor patterns	129
3.7.2	Shadows using cursor positioning	140
3.7.3	Outline print - a special flair	143
3.7.4	Text styles	144
3.8	The RastPort and the graphic system	148
3.9	The Bit-map structure	149
4.	The ViewPort	151
4.1	The ViewPort data structure	153
4.2	The graphic modes of the Amiga	155
4.2.1	The halfbrite mode	156
4.2.2	The hold-and-modify mode: 4096 colors	161
4.3	The ViewPort in the system	166
4.4	View: the graphic brain	168
4.5	Copper programming	175
4.5.1	Double buffering for lightening fast graphics	175
4.5.2	Programming the Copper yourself	180
4.5.3	Programming 400 simultaneous colors	185
4.6	The layers: soul of the windows	188
4.6.1	The Layer data structure	191
4.7	The different layer types	196
4.7.1	Simple layers: your own requester	197
4.7.2	The superlayer - 1024 x 1024 pixels!	199
4.7.3	Permanently deactivating layers	205
4.7.4	Using the BASIC commands within a layer	206
4.8	Layers in the system	212
5.	The Amiga fonts	213
5.1	The Amiga character generator	214
5.2	Opening your first font	218
5.3	Accessing the disk fonts	221
5.4	The font menu	228
5.5	Designing your own fonts	234
5.5.1	Reading the font generator	236
5.5.2	Big Text: enlarging text	240
5.5.3	A fixed width font generator	243
5.5.4	A proportional font	252

6.	Graphic hardcopy	259
6.1	A simple hardcopy routine	262
6.2	Hardcopies: enlarging and shrinking	265
6.3	Printing selected windows	268
6.4	ScreenDump - the complete screen	271
6.5	Multitasking hardcopy	274
7.	The IFF-ILBM Standard	279
8.	A 1024x1024 paint program	293
8.1	Paint-1024 program instructions	311
9.	Graphic programming in C	315
9.1	The Amiga libraries	316
9.2	The boss: the View	317
9.3	The foreman: ViewPort	317
9.4	The worker: the Bit-map	319
9.5	The messenger: RasInfo	322
9.6	The laborer: RastPort	323
9.7	Finish the day	325
10.	Lines and pixels	327
10.1	Pixels set with WritePixel	327
10.2	Drawing lines with Move and Draw	332
11.	Color: drawing pens	339
11.1	The Drawmodes	340
11.2	The foreground pen	341
11.3	The background pen	341
12.	Intuition and graphics	343
12.1	The individual screen	344
12.2	The window	345
12.3	Exiting Intuition	345
13.	Filling areas in C	347
13.1	A Flood function	347
13.2	Filling rectangles?	349
13.3	Polygon filling: Area, makes it possible	356
14.	The Colormap functions	381
14.1	Setting a new color palette	382
14.2	Changing one color	382
14.3	Available colors	383
14.4	The pixel's color	386

15.	Text output	387
15.1	The text length	389
15.2	Fonts with the Amiga	389
15.3	Opening fonts	390
15.4	Closing the font	391
15.5	Software controlled text styles	392
15.6	Fonts a la carte	393
16.	The Blitter functions	399
16.1	Clearing a memory area	399
16.2	Copying data with the Blitter	400
16.2.1	The ClipBlit function	402
16.3	Reading data with the Blitter	407
17.	Amiga resolution modes	413
17.1	The resolution modes	414
17.2	The color modes	418
17.2.1	The EXTRA_HALFBRITE mode	418
17.2.2	Hold-and-modify (HAM) for 4096 colors	419
17.3	The special modes	422
17.3.1	Dual playfield	422
17.3.2	Double buffering	423
17.4	Other modes	435
17.4.1	VP_HIDE	435
17.4.2	Sprites	435
17.4.2.1	The hardware sprites	435
17.4.2.2	Hardware sprites in 15 colors	439
17.4.2.3	Sprite collisions	440
18.	The Amiga animation system	451
18.1	Vsprites	451
18.1.1	Vsprites and collisions	455
18.2	Another GEL: the bob	465
18.2.1	Bobs in buffered bit-maps	476
18.3	AnimObs and AnimComps	479
18.3.1	Collisions with AnimObs	484
19.	Copper programming in C	495
Appendices		
A:	Structures and include files	503
B:	The library-functions	527
C:	The hardware registers	557
Index		593

Introduction

The graphic powers of the Amiga are fantastic. You have probably seen many examples of Amiga graphics that overwhelm competing computers. How can you take command of this wonderful machine? How do you create your own graphic programs and understand what is going on inside your Amiga?

The answers to these and hundreds of other questions are in this book. We will present the material so that it will be interesting for both beginners and experienced programmers.

In Chapter 1, we gently introduce the beginner to the amazing graphic world of the Amiga. Here you will learn first hand how to create graphics. Heavily documented BASIC programs demonstrate the use of the BASIC graphic statements and how to design and use movable objects, etc.

Experienced programmers will find in Chapters 2 thru 8 a step by step explanation of the Amiga's multitasking graphics environment. We'll introduce you to the multitude of system graphic routines in the Kickstart-ROM. We discuss in detail, with tables, all data structures (window, screen, ViewPort, and more.) At the end of this section you will be able to use graphics in 64 color, halfbrite, or 4096 color HAM mode. You will also be capable of programming a coprocessor and other tasks that were not thought possible with BASIC.

In Chapters 9 thru 19, we will examine the use of windows and screens using the C language, unveiling many system secrets. Last we enter the exciting world of automated animation, bobs, sprites and our own virtual sprite machine. Much of the information covered in this section is not even in the ROM manuals from Commodore!

Whether you use this book as an introduction to graphics or for solving a particular problem, you will find the contents of this book to be a wealth of graphics information.

Weltner, Trapp, Jennrich

1. Amiga graphics

In many ways the Amiga is a very unusual computer. Its excellent features have convinced not only graphics users, but many others that it is the computer for them.

The Amiga repeatedly surprises and fascinates users by the speed of its graphics, even graphics done in BASIC. Whether your graphics are very complex or only a small part of your BASIC program, there are several easy statements for drawing pixels, lines and circles.

A major difference between the Amiga and other computers is that you display text and hi-resolution graphics on the same screen. This means that you can easily mix text and graphics without opening another screen. You can start experimenting with the graphic statements immediately.

1.1 Setting pixels with PSET

The smallest unit in graphics is the pixel. We create all computer graphics, from a full screen to single lines and circles, by joining many small screen pixels. The statement you use to set pixels is short and simple:

```
PSET (10,20)
```

This example sets a pixel at a point 11 (10+1) from the left and 21 (20+1) down from the top of the window. Please note that pixel addressing, when referring to screen positions, starts with the value zero (0), not one (1).

1.1.1 Using the mouse and PSET: a simple drawing board

Using the statement PSET you can set any desired point within the limits of the specified output window. The following program is a good example of this. Each time you click the left mouse button you will set a pixel on the screen where the point of the mouse pointer is

located (a pixel at the tip of the pointer). This is a very primitive drawing program, but we will use this principle to create a complete paint program.

```
REM 1.1.1 Drawing with the Mouse
PRINT "Now You can draw with the Mouse"
WHILE INKEY$=""
IF MOUSE(0)<>0 THEN
  x=MOUSE(1)
  y=MOUSE(2)
  PSET (x,y)
END IF
WEND
```

As you can see, you need very few program lines for this small drawing program. The `MOUSE` function provides control of the mouse. When you click the left mouse button `MOUSE(0)` is not equal to zero. Now we can determine the mouse coordinates using `MOUSE(1)` for the X position and `MOUSE(2)` for the Y position.

1.1.2 Rosette and garland patterns

You can create beautiful graphics using only the `PSET` statement. As you have just seen, you can draw them yourself, or you can let the computer create them for you. To have the computer generate your graphics, you have to construct your statements so that the computer will understand what you want. To do this, you put your ideas into mathematical formulas. The following program generates different rosette patterns on the screen using formulas.

```
REM 1.1.2 Rosette
pi=3.1415296#
f=.5  'Specifies the relation of height and width
INPUT "How many Edges : ",edges
CLS
IF edges<> -1 THEN edges=edges+1
REM Predetermined Values
radius=100 'Radius of Maximum Circle
inside=3   'Count of "inside" lines
outside=3  'Count of "outside" lines
FOR t=-inside/10 TO outside/10 STEP .1
  FOR angle=0 TO 2*pi STEP .01
    x=radius*COS(angle)+t*radius*COS(angle*edges)
    y=radius*SIN(angle)+t*radius*SIN(angle*edges)
    PSET (300+x,y*f+100)
  NEXT angle
NEXT t
```

You can determine the shape of the rosette by entering different values for edges. Values between two and twenty work best, however negative numbers will also generate quite interesting images.

You can also modify the figures you created by experimenting with the program variables.

What you eventually see on the screen is a design formed from many ellipsoids that are connected together. Ellipsoid simply means a series of curves generated by rolling an ellipse around one of its axis points. We do not draw the ellipse itself, only the outermost path of the curve. You can see this path as we plot it on the screen. The complete ellipsoidal formula can be found in any mathematics reference guide. We have changed it slightly to make it more user-friendly.

1.1.3 Erasing pixels

Since you can set pixels, you must also be able to erase them. In AmigaBASIC this procedure is very similar to setting the pixels, in fact the statement sounds almost the same. This statement has the same syntax as `PSET` and is:

```
PRESET (x,y)
```

As you can see, the only difference between the statements is two letters. In a program it looks like this:

```
REM 1.1.2 Demo for PRESET
a=200
b=400
c=1
loop:
  FOR x=a TO b STEP c
    PSET (x,100)
    PRESET (x-40*c,100)
  NEXT x
  SWAP a,b
  c=-c
GOTO loop
```

In this program, a 40 pixels long line appears to be moving back and forth on the screen. What actually happens is that we add a new pixel to the beginning of the line and erase one from the end. To exit the program, press and hold the <CONTROL> key, then press the <C> key.

1.1.4 Adding color to your screen

The previous programs have not really been a challenge for the Amiga. We could easily transfer these programs to other computers, although this does not apply to all programs. For example, when it comes to the use of colors, the Amiga is unique. There are very few computers that can claim a palette of 4096 colors. We will begin by showing you how to use a 32 color palette and later advance to 64 and even 4096 colors. You can set colored pixels in the same way you previously set pixels, with the PSET statement. The only difference is that you must add, at the end of the statement, the value of the color for a color register:

```
PSET (10,20),2
```

In this example, we determine the color of the pixel from the second color register. The default color always starts with black, so this pixel will be black.

If you now attempted to access the rest of the 32 color registers (numbered 0-31) you would have problems. By the fifth, or higher register value, the computer would display an error message. Why? Because we do not use all 32 colors for every screen, the Amiga automatically saves memory by not assigning memory for these colors. The more colors you use, the more memory you require (more on this later).

First, we have to open a new screen (one that exists at the same time as the workbench screen and is either behind it or covered by it). When you open a new screen you can specify parameters such as width, height, mode and color depth. For depth you can use values between one (1) and five (5), certain screen modes only allow a maximum of four. The following formula calculates the number of colors: Two to the power of depth (2^{depth}). When you select a mode, you set the resolution of the screen. There are four possible modes on NTSC video systems:

Modes	Resolution
1	320*200
2	640*200
3	320*400
4	640*400

PAL systems, available in Europe, contain 256 screen lines. In most of our programs we use mode one. Please note that the width and height

of a screen cannot be greater than the values of the resolution mode selected.

Nothing can be sent directly to a screen. Once we open a screen we must also open a window for it. This window is where all text and graphics will appear.

```
REM 1.1.4A 32-Color DEMO

REM Open a Screen
SCREEN 1,320,200,5,1
REM Open a Window
WINDOW 2,"Colorpot",(0,0)-(311,185),16,1

FOR y= 0 TO 186
  FOR x= 0 TO 311
    PSET (x,y), (x+y) MOD 32
  NEXT x
NEXT y

WHILE INKEY$="": WEND

REM Close Window first, then Screen
WINDOW CLOSE 2
SCREEN CLOSE 1
```

On the screen you will see all 32 possible colors. At the end of the program we close the window and screen, since they are no longer required. We can also display some interesting patterns using the color demo. For example:

```
REM 1.1.4B Color Demo
SCREEN 1,320,200,5,1
WINDOW 2,"ColorDemo",(0,0)-(62,62),16,1
FOR m=0 TO 31
  FOR x=-m TO m
    FOR y=-m TO m
      PSET (31+x,31+y),((ABS(x) AND ABS(y))+32-m) MOD 32
    NEXT y
  NEXT x
NEXT m
WHILE INKEY$="": WEND
WINDOW CLOSE 2
SCREEN CLOSE 1
```

or

```
REM 1.1.4C Pyramid
SCREEN 1,320,200,5,1
WINDOW 2,,(0,0)-(20,10),16,1
FOR y=0 TO 19
  FOR x=0 TO 19
    f1=ABS(x -10)
    f2=ABS(y -10)
```

```
    IF f1<f2 THEN SWAP f1,f2
    PSET (x,y),31-f1
  NEXT x
NEXT y
WHILE INKEY$="" : WEND
WINDOW CLOSE 2
SCREEN CLOSE 1
```

These two programs demonstrate different methods of color manipulation. In the first program, we joined the X and Y values using AND. You can also create nice patterns by multiplying the values or by using XOR.

In the second program, we drew a square that had a dark outer edge and became progressively lighter towards the center. It creates the effect of looking at a pyramid from the top. To do this, we used only color registers 21 thru 31. These registers contain varying levels of gray shades useful in generating 3-D effects.

1.1.5 More about PSET and PRESET

We mentioned previously that PRESET erases the pixels that PSET set. This is only true as long as you do not specify a color value with PRESET:

```
PSET (100,100)
PRESET (100,100)
```

The above example demonstrates what we mean. We set a pixel and then erase it.

The example below has a completely different action, the pixel is drawn in white and not erased:

```
PSET (100,100),1
PRESET (100,100),1
```

The moment you specify a color register both statements will have the same result. It becomes a matter of personal preference which statement you use.

An alternate method to using PRESET (x,y) to erase pixels is to use PSET (x,y),0. Why are there two statements which seem to do the same thing? The default color (the color used if you do not specify a color with PSET or PRESET) is the key difference. With PSET the default is the foreground color and with PRESET it is the background color. This also has an effect on changes to the foreground and

background colors. You do not have to concern yourself with these values when using PSET to erase pixels.

Using both statements in a program makes it much easier to understand what is going on. You can see right away where pixels are being drawn or erased.

```
REM 1.1.5 Demo for P(RE)SET
RANDOMIZE TIMER
SCREEN 1,320,200,5,1
WINDOW 2,,(0,0)-(311,185),16,1
l=.8
DIM y(l*100)
WHILE INKEY$=""
COLOR INT(RND*32),INT(RND*32)
CLS
FOR j=1 TO 4
  y(j)=50+RND*80:s(j)=RND*50
NEXT j
FOR x=0 TO 6.2+l STEP .04
  FOR j=1 TO 4 STEP 2
    PRESET ((x-l)*50,y(j)-s(j)*SIN(x-l))
    PRESET ((x-l)*50,y(j+1)-s(j+1)*COS((x-l)*j))
    PSET (x*50,y(j)-s(j)*SIN(x)),INT(RND*32)
    PSET (x*50,y(j+1)-s(j+1)*COS(x*j))
  NEXT j
NEXT x
WEND
WINDOW CLOSE 2
SCREEN CLOSE 1
```

In this program, lines of different lengths move across the screen. As in the first PRESET demo program, we set a pixel at the front of a line and erase one at the end. What is special about this program is the changing of the foreground and background colors. As you can see, erasing using PRESET is quite easy.

1.1.6 The "opposite" of PSET: the POINT statement

By opposite we mean instead of setting a pixel, we can read a pixel to see if it is set and what color it is. To do this, we use the POINT statement:

```
PRINT POINT(x,y)
```

The result will be either the color or a value indicating that the pixel (x,y) coordinates are not in the output window. In the first case, the

number of the color register is printed, in the second a value of minus one (-1) is returned.

Our next program creates a few graphic lines, then the POINT statement is used to determine if a pixel has been set. The resulting picture will look as if it was created by using randomly set pixels.

```
REM 1.1.6 Patterns
SCREEN 1,320,200,5,1
WINDOW 2,,(0,0)-(80,80),16,1
RANDOMIZE TIMER
x=40
y=40
Direction:
dx=INT (RND*5)-2
dy=INT (RND*5)-2
IF dx=0 AND dy=0 THEN Direction
IF ABS(dx)>ABS(dy) THEN
  st=ABS(dx)
ELSE
  st=ABS(dy)
END IF
WHILE INKEY$=""
  IF POINT (x+dx,y+dy)=-1 THEN Direction
  FOR i= 1 TO st
    x=x+dx/st
    y=y+dy/st
      PSET (x,y),POINT(x,y) MOD 31+1
  NEXT i
WEND
WINDOW CLOSE 2
SCREEN CLOSE 1
```

The arrangement used to build this screen is quite simple. One pixel wanders around the screen. (With each move we check the pixel positions color register and increment it by one.) When the moving pixel hits a window border, we change directions.

End the program by pressing any key.

1.1.7 Relative addressing

With PSET, and almost all other graphic statements in AmigaBASIC, there are two methods of addressing. We can easily compare the first method to home addresses and ZIP codes. When you send a letter it always arrives at the address to which you sent it. This is the method of addressing we have been using in our programs:

```
PSET (20,30)
```

The values 20 and 30 are the absolute coordinates of the pixel we want to set in its row and column. This method of addressing is named "*absolute*". The second method of addressing is called "*relative*". Relative address coordinates do not refer directly to a row and column position, instead they point to a position relative to your current position. This is like telling the computer: from your current position move three pixels to the right and then two down. The key to this method is the current position of the graphic cursor. The indicator for relative addressing is the word `STEP` which is positioned before the parenthesis:

```
PSET STEP (3,2)
```

Although we cannot simply ask where the graphic cursor is, we can move it easily without doing any drawing.

```
v=POINT (x,y)
```

We move the graphic cursor automatically when we set or erase points. By using this form of the `POINT` statement, we can relocate it anywhere we want. The variable `'v'` can be any variable not already used in your program.

When you start a program, the graphic cursor will always be in the center of the output window.

If you use relative addressing with constant coordinates, you plot pixels with the same offset between them. This is not something you would necessarily want, but our next sample program demonstrates how this works. This program is very similar to our first drawing program but instead of setting one pixel when you press a mouse button it will set several pixels giving a brush effect.

```
REM 1.1.7 Relative Addressing
WHILE INKEY$=""
IF MOUSE(0)<>0 THEN
  x=MOUSE(1)
  y=MOUSE(2)
  PSET (x,y)
  PSET STEP (10,10)
  PSET STEP(-10,10)
  PSET STEP (-10,-10)
  PSET STEP (10,0)
END IF
WEND
```

The first pixel we set using an absolute address. The other four pixels we set with a relative address, which made the distance between them the same.

1.2 The LINE statement

The LINE statement has two uses that look similar, but produce completely different results. First, as the name implies, you can draw straight lines. You can also use LINE to draw boxes. We will discuss the second form of the LINE statement in more detail later on.

```
LINE (20,10)-(200,100),2
```

The above statement draws a black (color register 2) line from point (20,10) to point (200,100).

1.2.1 The power of LINE

With the PSET statement, you can perform almost anything that is possible in the computer graphics world. You can create any desired graphics or lines you want. The problem with using PSET by itself is the time involved. The time saved using the LINE statement is enormous and easily demonstrated:

```
REM 1.2.1 Benchmark for PSET and LINE
REM Straight Line with PSET
FOR i= 0 TO 180
  PSET(i,i)
NEXT i
REM Straight Line with LINE
LINE (180,0)-(0,180)
```

The LINE statement opens up more possibilities, probably a lot more than the PSET statement. In the same way you form lines by setting pixels one after another, you can draw a line by simply specifying the start and end. In this example, the start and end coordinates for the LINE statement were in the same row.

1.2.2 The Moire effect

Moire patterns becomes visible whenever you draw lines close to or intersecting each other. Using this effect you can design astounding pictures. Watch the graphics created by the program below.

```
REM Moire-Lattice
a=182              'size of the rectangle
FOR s=1 TO 10
  CLS
  FOR i=0 TO a STEP s
    LINE (140,1)-(140+2*a,i),2
    LINE (140,1)-(140+2*i,a),2
    LINE (140+2*a,a)-(140,i),2
    LINE (140+2*a,a)-(140+2*i,1),2
  NEXT i
  WHILE INKEY$="": WEND
NEXT s
```

We draw lines from two corners of the rectangle to the opposite side. The Moire effect shows up at the connecting points of both corners because that is where we have drawn most of the lines.

We can also achieve this effect by drawing many lines from a single point and increase the effect by alternating between two different colors.

```
REM 1.2.2B Moire Demo 2
xmax=618   'size of the output window
ymax=186
COLOR 1,2  'background black
Start:
CLS
xm=INT(RND*xmax)  'Coordinates of Center Point
ym=INT(RND*ymax)
FOR i=0 TO ymax
  LINE (xm,ym)-(0,i),i MOD 2+1
  LINE (xm,ym)-(xmax,i),i MOD 2+1
NEXT i
FOR i=0 TO xmax
  LINE (xm,ym)-(i,0),i MOD 2+1
  LINE (xm,ym)-(i,ymax),i MOD 2+1
NEXT i
WHILE INKEY$="": WEND
GOTO Start
```

When you observe the graphics created by this program, it is hard to believe we created them so easily.

1.2.3 Quix, lines galore

Again we come back to the speed of the LINE statement. In the following program, we let quix race around the screen. Quix is composed of many lines that are more or less parallel. The movement of quix is caused by erasing lines at one end and drawing them at the opposite end. It becomes difficult to tell where a line starts or ends,

since the portion added uses the same end coordinates as the line it is being added to. We simply modify the coordinates slightly to make the existing line longer.

```
REM 1.2.3 Quix
DEFINT a-z
SCREEN 1,320,200,5,1
WINDOW 2,"Quix",(0,0)-(297,185),31,1
RANDOMIZE TIMER
a=20
DIM x(1,a),y(1,a)
x(0,0)=150
y(0,0)=100
x(1,0)=170
y(1,0)=100
WHILE INKEY$=""
  FOR z=0 TO a
    LINE (x(0,z),y(0,z))-(x(1,z),y(1,z)),0
    FOR i= 0 TO 1
newx: x(i,z)=ABS(x(i,alt)+RND*20-10)
      IF x(i,z)>WINDOW(2) THEN newx
newy: y(i,z)=ABS(y(i,alt)+RND*20-10)
      IF y(i,z)>WINDOW(3) THEN newy
    NEXT i
    f1=f1 MOD 31 +1
    LINE (x(0,z),y(0,z))-(x(1,z),y(1,z)),f1
    alt=z
  NEXT z
WEND
WINDOW CLOSE 2
SCREEN CLOSE 1
```

While quix is moving around you can stop it by clicking and holding on the window size gadget and shrinking the window. The coordinates used by quix will never be greater than the current window size. You can determine the actual size of the output window by using WINDOW(2) (width) and WINDOW(3) (height).

1.2.4 Function plotter

A function plotter is one of many practical mathematical applications requiring graphics. It can be a great help in the visual examination of functions. Experimenting with parameters and functions can also be a lot of fun while looking for interesting curves. This function plotter is very comprehensive and user friendly. For example, the program will compute your coordinates so that the function plotted will always be within the window limits. The ¶ characters in the following program

listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
REM 1.2.4 Function Plotter¶
DEFDBL x,y,m,f¶
DIM y(618)     'maximum Number of Function Values¶
x1=-10: x2=10 'Value Limits¶
func=1         'First Function¶
coordinates=1 'Coordinates Intersect at¶
MENU 1,0,1,"Activity          "¶
MENU 1,1,1,"Draw Function     "¶
MENU 1,2,1,"Enter Coordinates" ¶
MENU 1,3,1,"Crosshairs off    "¶
MENU 1,4,1,"Quit              "¶
MENU ON¶
MENU 2,0,1,"Function      "¶
DEF FNy1(x)=SIN(x)/(x^2+1)¶
a$(1)="y=sinx/(x^2+1)      "¶
MENU 2,1,1,a$(1)¶
DEF FNy2(x)=SIN(x)*10-1/x+x^2¶
a$(2)="y=sin(x)*10-1/x+x^2"¶
MENU 2,2,1,a$(2)¶
DEF FNy3(x)=SIN(1/x)/x¶
a$(3)="y=sin (1/x) /x      "¶
MENU 2,3,1,a$(3)¶
DEF FNy4(x)=(EXP(x)-1)/(EXP(x)+1)¶
a$(4)="y=(e^x-1)/(e^x+1)   "¶
MENU 2,4,1,a$(4)¶
WINDOW 1, a$(1),,23¶
GOSUB Calculate¶
Select:        SLEEP¶
               ON MENU GOSUB Branch¶
               GOTO Select¶
Selection:     ON MENU(1) GOSUB Calculate, Entry,
Crosshair, Quit¶
               RETURN¶
Branch:        ON MENU(0) GOTO Selection¶
               func=MENU(1)¶
               WINDOW 1, a$(func)¶
Entry:         WINDOW 2,"Coordinate Entry",(0,0)-
(250,9),16¶
               INPUT "Start Value : ";x1¶
               INPUT "End Value   : ";x2¶
               IF x2<x1 THEN SWAP x1,x2¶
               WINDOW CLOSE 2¶
Calculate:     IF x1=x2 THEN Entry¶
               fwidth=WINDOW(2)¶
               WINDOW 2," Calculating Function - Please
Wait!",(0,0)-(300,0),0¶
               min=0¶
               max=0¶
               ON ERROR GOTO Errorcond¶
               FOR i=0 TO fwidth¶
                 fvalue=x1+(x2-x1)*i/fwidth¶
                 ON func GOSUB F1,F2,F3,F4¶
```

```
                    IF y(min)>y(i) THEN min=i¶
                    IF y(max)<y(i)OR y(max)=9999 THEN max=i¶
Proceed:          NEXT i¶
                  ON ERROR GOTO 0¶
                  min=y(min)¶
                  max=y(max)¶
                  WINDOW CLOSE 2¶
                  GOSUB Title¶
Drawplot:         CLS¶
                  fheight = WINDOW(3)-8¶
                  IF coordinates=1 THEN¶
                    IF min<=0 AND max=>0 THEN ¶
                      h=fheight+min*fheight/(max-min)¶
                      LINE (0,h)-(fwidth,h),2¶
                    END IF¶
                    IF x1<=0 AND x2=>0 THEN¶
                      b=-x1*fwidth/(x2-x1)¶
                      LINE (b,0)-(b,fheight),2¶
                    END IF¶
                  END IF    ¶
                  IF min=max THEN    ' When min=max then draw
a straight line '¶
                    IF max=9999 THEN  ' Function value not
defined '¶
                      CLS¶
                    ELSE ¶
                      LINE (0,fheight/2)-(fwidth,fheight/2)¶
                    END IF¶
                  END IF¶
                  j=0¶
                  WHILE (y(j)=9999 AND j<618) ' find first
defined Function Value '¶
                    j=j+1¶
                  WEND¶
                  IF j=618 THEN RETURN¶
                  PSET (j,fheight-(y(j)-min)*fheight/(max-
min))¶
                  FOR i=j+1 TO fwidth¶
                    IF y(i)<>9999 THEN ¶
                      IF flag THEN¶
                        PSET (i,fheight-(y(i)-
min)*fheight/(max-min))¶
                        flag=0¶
                      ELSE  ¶
                        LINE -(i,fheight-(y(i)-
min)*fheight/(max-min))¶
                      END IF¶
                    ELSE¶
                      flag=1¶
                    END IF¶
                  NEXT i¶
                  RETURN¶
Errorcond:        y(i)=9999¶
                  RESUME Proceed¶
F1:               y(i)=FNy1(fvalue)¶
                  RETURN¶
```

```
F2:             y(i)=FNy2(fvalue)¶
                RETURN¶
F3:             y(i)=FNy3(fvalue)¶
                RETURN¶
F4:             y(i)=FNy4(fvalue)¶
                RETURN ¶
Title:          MENU 3,0,1,MID$(STR$(x1),1,5)+
"<x<"+MID$(STR$(x2),1,5)¶
                MENU 4,0,1,MID$(STR$(min),1,5)+
"<y<"+MID$(STR$(max),1,5)¶
                RETURN¶
Crosshair:      IF coordinates=1 THEN¶
                  coordinates=0¶
                  MENU 1,3,1,"Crosshairs On    "¶
                ELSE¶
                  coordinates=1¶
                  MENU 1,3,1,"Crosshairs Off   "  ¶
                END IF¶
                GOTO Drawplot¶
Quit:           WINDOW 1,"Function Plotter",(0,0)-
(617,184),31¶
                MENU RESET¶
                END¶
```

You are probably asking yourself what does a function plotter have to do with the LINE statement since we form a function by plotting points. Our program works by plotting a point for each curve. Between these points are large gaps and if we only plotted the calculated points our function would not look like a curve. We make the connection between all of these points by using the LINE statement.

1.2.4.1 Function plotter modes and menu controls

This program has two standard and two pseudo menus. The pseudo menus have no submenus and only provide you with information. The first pseudo menu displays the limit values of the curve in the X direction. The second pseudo menu lists the maximum and minimum Y values of the function. Both of these menus are only visible after your function has been plotted and changed with each function.

The activity menu has four submenus. Submenu one redraws the current function and is used after you have changed the size of the output window. The other three submenus are for entering new coordinates, making the crosshair visible or invisible and exiting the program. With the second menu you can choose from one of four preset mathematical functions. Unfortunately with BASIC it is not possible for you to input the function formulas. If you want to test other functions, you can do so by changing those at the beginning of

the program. You will also have to change the text for `a$(n)` since it lists the functions for menu two.

1.2.4.2 Undefined function values

This program allows you to view mathematical functions on your screen without spending a lot of time figuring out values. For example, lets say you want to compute the value of a curve 1/x and you input a value of x=0. Since division by zero creates an error, and we used the `ON-ERROR-GOTO` assignment before calculating the function values, BASIC branches on the error. Simply put, instead of the program halting on the error it goes to a specified error handling routine which processes the problem.

In this program, the function value on specific errors will default to 9999. If the drawing routine receives this value, it then knows the pixel should not be plotted.

In addition to division by zero, overflow errors are also handled. There is, of course, the danger of other unforeseen errors happening at which time the program will stop.

For the program to properly interpret errors that occur after the function calculations, the `ON-ERROR-GOTO` assignment is switched off again.

1.2.4.3 Scaling

You can directly affect the size and shape of a curve by enlarging or shrinking it. When you change the size of the output window there is a proportional effect on the displayed curve by either stretching or compressing it.

By scaling we mean that a curve is always displayed to fill the entire window. However, the values of the X and Y axis are dependent on the height and width of the window. Stretching a curve can sometimes produce confusing results. On the other hand, compressing it can cause a smooth straight curve to appear more sloped.

The advantage to this method of presentation is that you do not have to worry much about the course of the curves. As mentioned previously, you can obtain more information about a curve from the menus.

1.2.5 Drawing rectangles

Earlier we discussed using the LINE statement not only to draw lines, but also rectangles. Now we will show you how to draw a rectangle with a single LINE statement. The difference between lines and rectangles is a ,B at the end of the statement.

```
LINE (20,10)-(200,100),2,B
```

This statement draws an unfilled black rectangle on the screen. The first coordinate pair is the upper left hand corner and the second pair is the lower right hand corner. You can see it is very easy to draw rectangles. Although the sides will always be parallel to the output window, this type of rectangle is very useful.

You can also draw filled rectangles instead of unfilled by using ,BF instead of ,B at the end of the statement.

```
LINE (30,10)-(300,100),3,BF
```

If you leave out the color parameter, the rectangle is drawn in the foreground color.

With this statement we can also perform a speed test. Our test program is similar to the boxes demo of the workbench. We will draw filled rectangles with random colors and coordinates.

```
REM 1.2.5 Speed Test of
REM       the LINE Statement
WHILE INKEY$=""
  x=WINDOW(2)
  y=WINDOW(3)
  LINE (RND*x,RND*y)-(RND*x,RND*y),INT(RND*4),BF
WEND
```

Even though this is a very simple program, it seems amazing if you are not familiar with the Amiga. The speed with which the Amiga draws and fills rectangles is very fast, so fast in fact that you do not have time to count them. Keep in mind that these are not all small rectangles. Some of them have thousands of pixels and you still cannot see the difference.

1.2.6 Relative addressing with LINE statement

You can use relative coordinates with the LINE statement in either mode in exactly the same way as you used them with PSET.

The first coordinate pair is relative to the graphic cursor position and the second pair is relative to the starting point of the line.

Both coordinates do not have to be relative. The following statement draws a line from (30,20) to (20,120).

```
LINE (30,20)- STEP(-10,100)
```

If you want to continue drawing from the current position of the graphic cursor, you just leave the first coordinate pair out. We will demonstrate this method in the following program in which we will draw multiple rectangles inside of each other:

```
REM 1.2.6A Boxes Within Boxes
SCREEN 1,320,200,2,1
WINDOW 2,"Rotated Rectangle",(0,0)-(311,185),16 ,1
COLOR 2,1
CLS
d=5                'Distance (1-10)
b=60               'Width of first Rectangle
FOR x=0 TO 311 STEP b
  FOR y=0 TO 185 STEP b
    x1=x: x2=x+b
    y1=y: y2=y
    FOR a= 0 TO .7 STEP ATN(d/b)   'a< PI/4
      IF ((x+y)/b)MOD 2=0 THEN     'Rotate Direction
        LINE (x1,y1)-(x2,y2)
        LINE -(2*x+b-x1,2*y+b-y1)
        LINE -(2*x+b-x2,2*y+b-y2)
        LINE -(x1,y1)
      ELSE
        LINE (2*x+b-x1,y1)-(2*x+b-x2,y2)
        LINE -(x1,2*y+b-y1)
        LINE -(x2,2*y+b-y2)
        LINE -(2*x+b-x1,y1)
      END IF
      x1=x1+COS(a)*d  'Calculate next rectangle
      x2=x2-SIN(a)*d
      y1=y1+SIN(a)*d
      y2=y2+COS(a)*d
    NEXT a
  NEXT y
NEXT x
WHILE INKEY$="": WEND
WINDOW CLOSE 2
SCREEN CLOSE 1
```

When the program is finished you can hardly distinguish the individual rectangles. They have disappeared into the overall pattern. This effect is increased by neighboring rectangles being rotated in the opposite direction.

Both coordinate pairs are required for only the first line of a rectangle or another geometric figure.

The second coordinate pair of the LINE statement determines where the graphic cursor will be after drawing the line. If you plan ahead, you can save yourself some work: Using LINE to draw boxes, you can swap the X and Y coordinates and determine what corner the graphic cursor will be at after the statement. This is very useful when using relative addressing.

The following program draws a random bar graph of the type popular for statistics. Our bar graph is to be three dimensional so we have created a shadow composed of lines.

```
REM 1.2.6B Bargraph
RANDOMIZE TIMER
SCREEN 1,320,200,4,1
WINDOW 2,"Bargraph",,31,1
FOR i=0 TO 7
  x=30+i*37
  y=INT(RND*160)+1
  LINE (x,180-y)-STEP(6,-6),i+1
  LINE -STEP (0,y),i+1
  LINE -STEP (-6,6),i+1
  LINE -STEP (-20,-y),i+1,bf
  LINE -STEP (6,-6),i+1
  LINE -STEP (20,0),i+1
NEXT i
WHILE INKEY$="": WEND
WINDOW CLOSE 2
SCREEN CLOSE 1
```

Except for the first coordinate pair of each bar all coordinates are relative, including those of the boxes. Using this method can save you some time because you have to calculate only a few coordinates.

1.3 The CIRCLE statement

For a computer that is designed to handle professional graphics, we would expect BASIC to include a 'CIRCLE' statement.

In school you learned that you require a center and a radius to draw a circle. This is the same for AmigaBASIC:

```
CIRCLE (200,100),100
```

We can draw colored circles in the same way. The value for the color register is simply added at the end of the statement:

```
CIRCLE (200,100),100,2
```

With both circles there are two notable points:

1. Both figures are more ellipse shaped than round.
2. The radius is smaller than the height in pixels of the window, but with a radius of 100 the circle should not fit in the window.

Both points are dependent upon each other and the aspect ratio.

1.3.1 The aspect ratio

You have probably experimented with the adjustments on the back of your monitor. One of these adjustments changes the vertical height of your screen. Using this adjustment you could expand your screen vertically until our last circle appears round. The problem with doing this is that you would cut off the top and bottom of the screen. If you change to a different screen resolution, your circle would be egg shaped.

We have a much better solution. You can use a parameter with the CIRCLE statement that adjusts the pixel height and width ratios to round the image. This is how you do it:

```
CIRCLE (100,100),100,,,,2
```

There are four commas between the radius and the aspect ratio because we have left out three parameters: the color, which we already selected, and two angle parameters that we will discuss later.

Drawing perfect circles is not the only way you can use the aspect ratio. In our next program, which draws many ellipses and circles, you can create your own shapes. The shapes we achieve with this program range from circles to diamonds to four pointed stars. All of these figures are created using many arcs:

```
REM 1.2.1 Ellipses
SCREEN 1,320,200,2,1
WINDOW 2,,(0,0)-(311,185),16,1
FOR g= 0 TO 80 STEP 5
  CLS
  FOR f= .0001 TO 1 STEP .1
    CIRCLE (100,100),(80-g*f),,,,f
    CIRCLE (100,100),(80-g*f),,,,1/f
  NEXT f
  WHILE INKEY$="": WEND
NEXT g
WINDOW CLOSE 2
SCREEN CLOSE 1
```

We have used the lower resolution of 320*200 for this program because the aspect ratio is equal to one. This is the reason the horizontal and vertical ellipses are so similar. The difference is that once we use an aspect ratio of `f` which is smaller than one and the second time we use `1/f` which is larger than one. Thus the ellipses are basically the same, except one is tall and the other wide.

All of the figures are dependent on the values of the aspect ratio and the radius. The larger we make the value of `f`, the rounder the circle and, at the same time, the smaller the radius. How small the radius becomes is different for each figure.

1.3.1.1 Animation using CIRCLE

The next program contains simple animated graphics. It simulates a bouncing ball on the screen. With each bounce the ball will compress slightly as if made out of rubber. This action gives it the next bounce.

```
REM 1.3.1.1A Springing Ball
SCREEN 1,320,200,2,1
WINDOW 2,,(0,0)-(100,100),16,1
WHILE INKEY$=""
  FOR i=0 TO 3.14 STEP .08
    f=1
    y=70*SIN(i)
    IF y<10 THEN f=.5+y/20
    CLS
    CIRCLE (50,100-y),20,1,,,f
```

```
  NEXT
WEND
WINDOW CLOSE 2
SCREEN CLOSE 1
```

The aspect ratio is dependent on the height of the ball, which we calculate with the sine function. When the ball is lower than the radius of the circle, we change the value of f (the aspect ratio) and the ball compresses.

The principle of our animation is simple: We draw a circle and calculate the position of the next circle. Before we draw the new circle we clear the screen. Because you don't have to calculate the circle, it is simpler and faster to clear the screen instead of erasing the existing circle with the background color. When we use CLS, the Blitter, a special graphic coprocessor, takes care of most of the work. Clearing the screen is a small task for the Blitter, which can fill areas of up to a million pixels in only one second.

In the next program, we show you a wire model of a chess piece. When you press a key the figure will rotate. You might recognize this figure immediately; it's the queen.

The moire effect, which you saw with the LINE statement, also appears with the CIRCLE statement. It will always appear when many lines, or curves in this case, are close together.

```
REM 1.3.1.1B Chess Piece
FOR f=0 TO .5 STEP .05
CLS
  READ l
  FOR i= 1 TO l
    READ a
    CIRCLE (320,150-i*3*2*(.5-f)),a*2,2,,,f
  NEXT i
  RESTORE
  WHILE INKEY$="": WEND
NEXT f
REM Queen
DATA 39,31,29,29,31,26,23
DATA 21,27,22,19,16
DATA 14,13,13,12,12,12,11,11,11,11,22
DATA 16,16,20,16,16
DATA 17,18,19,21,23,26,29
DATA 27,25,10,10,8
```

We use only the CIRCLE statement to rotate this figure. In this program the variable f sets the aspect ratio and the height for the center point of each circle. At first you will see the queen from the side. As you rotate the figure to a top (or bottom) view, the circles will get rounder and the center points closer together.

1.3.1.2 The aspect ration in circle formulas

The default aspect ratio is 0.44 at a screen resolution of 640*200. Depending on the adjustment of your monitor, this value will draw circles or ellipses. The best way to use the CIRCLE statement is with a variable for the aspect ratio that you set at the start of your program. You only have to change the variable once to change the program for different monitors.

You can use any value between 0 and 200 for the aspect ratio. Using zero will result in a horizontal line and using 200 causes a similar action, only vertically.

When you set the aspect ratio smaller than one, the horizontal distance from the center to the side will be equal to the radius. A value greater than one makes the horizontal distance smaller than the radius and the vertical distance equal to the radius. An aspect ratio of one will make the height and width distances equal to the radius. You can calculate what the differences of height and width will be for other aspect ratio values. To do this we must understand how the CIRCLE statement computes its path. The following program uses everything we have learned so far about CIRCLE. Although a bit slow in BASIC, this program will help you understand the aspect ratio:

```
REM 1.3.1.2A Simulation of the Circle statement
CIRCLE (130,100),100,2,,,.2
CALL Circlefunc (130!,100!,100!,1!,.2)
END
SUB Circlefunc (mx,my,radius,colr,f) STATIC
FOR w=0 TO 2*3.1415296# STEP .01
  IF f<1 THEN
    x=COS(w)*radius
    y=-f*SIN(w)*radius
  ELSE
    x=COS(w)*radius/f
    y=-SIN(w)*radius
  END IF
  PSET (mx+x,my+y),colr
NEXT w
END SUB
```

The above program shows you where to use the aspect ratio in a formula. It is the factor that you multiply with X (for f smaller than one) and divide with Y (for f greater than one). Knowing this we can calculate exactly what distance a specific pixel will be from the center. The following program does this job for us by drawing in the radiuses.

```
REM 1.3.1.2B Draw Radiuses
CIRCLE (200,100),100,2,,,.2
x1=200
y1=100
FOR angle= 0 TO 6 STEP .5
  x=x1
  y=y1
  CALL Coordinates (x,y,100!,angle,.2)
  LINE (x1,y1)-(x,y)
NEXT angle
END
SUB Coordinates (mx,my,radius,w,f) STATIC
IF f<1 THEN
    mx=mx+COS(w)*radius
    my=my-f*SIN(w)*radius
  ELSE
    mx=mx+COS(w)*radius/f
    my=my-SIN(w)*radius
  END IF
END SUB
```

The sub program calculates the intersections and returns the results to the main program. You can use the same subroutine for all of the circles because we pass all required parameters to it. The parameter that determines what point you reach is the angle. In the above program, we calculate twelve angles between zero and six that set the circumference for the circles. The sub program then calculates the new values for the variables based on the circles center point.

1.3.2 Angle parameters with CIRCLE

The angle parameters are the last two parameters for the CIRCLE statement. These parameters allow you to draw only a piece of a circle. The first angle parameter sets the starting point of an arc and the second angle sets the end point. Arcs are plotted counter clockwise. The CIRCLE statement looks like this with all of the parameters:

```
CIRCLE (mx,my),radius,color,start,end,Aspect
```

If you use two CIRCLE statements with the same start and end angles, but swapped, you would draw a complete circle.

You can specify any start and end angle between -2*PI and 2*PI. You may also know from your math classes that in curves 2*PI equals a full circle. Here you might get the idea that using both maximum parameters (-2*PI and 2*PI) would draw two circles. That is mathematically logical, but false in this case. The negative sign in this instance has no mathematical meaning, but rather a technical one. If a

start angle has a minus sign in front, a line is drawn connecting the circles center to the curves starting point. This also applies to the end angle value. The negative sign means smaller than zero. Even if you use a minus sign in front of a zero, it is not negative. To start a curve with an angle of zero and draw a line, you must use a value of -0.0001. This value does not have much effect on the angle, but does tell the computer what you want to do.

1.3.3 Relative addressing with CIRCLE

We can also use relative addressing with the CIRCLE statement. When you draw circles using relative addressing, the graphic cursor represents the center point. Unlike PSET and LINE the graphic cursor does not move to the last pixel drawn, but remains at the center of the circle.

1.3.4 Pie charts

You have probably seen pie charts. For example, we could represent election results using a pie chart to show who won how many seats. The pie sections symbolize the total possible seats. The seats of one party are represented by pieces of the pie in the party's color. An election is only one of the many uses of the pie chart.

The Amiga can also display pie charts. Our program displays them almost the same as those you have seen on television, in color and three dimensional.

```
REM 1.3.4 Pie Charts
SCREEN 1,320,200,5,1
WINDOW 2,"Pie Charts",,,1
f=.3
pi=3.141529
start:
CLS
sum=0
INPUT "How many Values";n

IF n<2 THEN
  CLS
  PRINT "Demo Program"
  n=INT(RND(1)*10)+3
  DIM value(n),colr(n)

  FOR i=1 TO n
```

```
    value(i)=RND(1)*20+1
    colr(i)=INT(RND(1)*31)
    sum=sum+value(i)
  NEXT i
ELSE
  DIM value(n),colr(n)
  FOR i=1 TO n
    value(i)=1
    PRINT i". Value";
    INPUT value(i)
    INPUT "Color";colr(i)
    colr(i)=colr(i) MOD 32
      sum=sum+value(i)
   NEXT i
   CLS
END IF
REM Draw Pie Chart
mx=WINDOW(2)/2
my=WINDOW(3)/2
w1=0
radius=mx-10
CIRCLE (mx,my+20),radius,1,pi,2*pi,f
LINE (mx-radius,my)-(mx-radius,my+20)
LINE (mx+radius,my)-(mx+radius,my+20)
LINE (mx,my)-(mx+radius,my)
FOR i=1 TO n
  w2=w1+2*pi*value(i)/sum
  CIRCLE (mx,my),radius,1,-w1,-w2 ,f
  REM Color Segment
  x=COS(w1+(w2-w1)/2)*radius/2
  y=-f*SIN(w1+(w2-w1)/2)*radius/2
  PAINT STEP(x,y),colr(i),1
  IF w2>pi THEN
    REM Draw Side Line
    x=COS(w2)*radius
    y=-f*SIN(w2)*radius
    LINE (mx+x,my+y)-(mx+x,my+y+20)
    REM Color Side
    IF w2-.1>pi THEN
      x=COS(w2-.1)*radius
      y=-f*SIN(w2-.1)*radius
      PAINT (mx+x,my+y+18),colr(i),1
    END IF
  END IF
  w1=w2
NEXT i
INPUT "New Graphic";a$
ERASE value,colr
IF a$<>"n" THEN start
WINDOW CLOSE 2
SCREEN CLOSE 1
```

This program gives you the option of selecting your own values or using randomly generated values. The demo program will start if you

enter a zero after the value query. If you enter your own values, you can select any of 32 colors for each value.

Each segment of the pie is drawn separately. Earlier we discussed how you can draw a segment. All you need is a starting and ending angle. You place a minus sign in front of each angle to draw the connecting lines from the center to the ends of the arc. To bring the color into each segment we need another statement. All you need is a framed surface and the coordinates of a pixel within this frame to use the PAINT statement. The first requirement was fulfilled when we drew the circle segment. To complete the second requirement we have to use the circle formula that we used before to demonstrate the aspect ratio. You calculate the center point of a segment with the following:

```
X=COS(W1+(W2-W1)/2)*RADIUS/2
Y=-f*SIN(W1+(W2-W1)/2)*RADIUS/2
```

In this formula W1 is the start angle, W2 the end angle, and f is the aspect ratio (here it is always smaller than one). The X and Y variables are not absolute window coordinates, but relative to the circle center point. This allows us to fill the segment with the following statement.

```
PAINT STEP (X,Y),colr(i),1
```

The variable colr(i) contains the color with which we want to fill the segment. The trailing one means we want the segment outlined in white.

We use a modified version of the formula that calculates center points to separate and color the borders of the pie.

1.3.5 Pixels and lines with CIRCLE

As silly as it may seem, you can also plot pixels, not just ellipses and circles, using the CIRCLE statement. We are not referring to the lines created by setting the aspect ratio to zero as you saw above. Nor do we mean the pixel set by using a radius of zero, neither of these actions have a practical use. To draw useful lines and pixels we have to manipulate the angle parameters.

We set pixels by using start and end angle values that are equal. You draw a line by making one of the equal angles negative. The distance of a pixel or length of the line are set by the radius and the aspect ratio.

In certain instances, drawing lines with this method is very useful. You can draw a line for a specific length and angle from one point. To do this with a LINE statement, you would need two known points and would have to calculate the angle.

We can demonstrate the advantages of this unusual use of CIRCLE with the following program. This is an analog clock that we draw using only the CIRCLE statement and no other graphic statements. You save a lot of time and calculation effort because the program does not have to compute the coordinates of the hands and minute divisions.

```
REM 1.3.5 Analog Clock
pi=3.1415926#
f=.5            ' Aspect Ratio '
REM Draw Circle
CIRCLE (100,100),100,1,,,f
REM Minute Marks
FOR i=.0001 TO 2*pi STEP pi/30
CIRCLE (100,100),97,1,i,i,f
NEXT i
REM Hours Marks
FOR i=.0001 TO 2*pi STEP pi/6
CIRCLE (100,100),93,1,i,i,f
CIRCLE (100,100),90,1,i,i,f
NEXT i
st:  INPUT "Hour    ";hour
     IF hour >12 GOTO st
INPUT "Minutes ";minutes
hangle=-((12-hour)*60-minutes)*pi/360-pi/2.0001
mangle=-(60-minutes)*pi/30-pi/2.0001
IF hangle<-2*pi THEN hangle=hangle+2*pi
IF mangle<-2*pi THEN mangle=mangle+2*pi
REM Hands
CIRCLE (100,100),85,2,mangle,-mangle,f
CIRCLE (100,100),70,3,hangle,-hangle,f
ON TIMER(60) GOSUB Timeout
TIMER ON
WHILE 1:WEND
Timeout:
REM Delete Old Hands
CIRCLE (100,100),70,0,hangle,-hangle,f
CIRCLE (100,100),85,0,mangle,-mangle,f
hangle=hangle+pi/360
mangle=mangle+pi/30
IF hangle>0 THEN hangle=hangle-2*pi
IF mangle>0 THEN mangle=mangle-2*pi
REM New Hands
CIRCLE (100,100),85,2,mangle,-mangle,f
CIRCLE (100,100),70,3,hangle,-hangle,f
RETURN
```

The TIMER statement makes sure that we call the subroutine every minute that updates the clock. This statement lets you bypass interrupt

programming. which is something you may not be familiar with. This method is much more comfortable to use than the usual way of interrupt programming in AmigaBASIC.

Instead of the endless loop, `WHILE 1:W END`, you could substitute your main program. In this case, it would be useful to place the clock within its own window.

1.4 Area fill

So far you have learned several ways to draw different shapes. You used the LINE statement to quickly draw colored rectangles on the screen.

The CIRCLE statement does not have a built in fill function so we stepped ahead a bit and used the PAINT statement. This is the statement we are now going to look at in detail.

1.4.1 The PAINT statement

To paint simply means to fill an area with color. The PAINT statement does this with amazing speed. To do this, first specify the location of a pixel inside the shape you want to paint. If the shape and fill colors are not the same, add the shape color at the end of the statement. The shape color tells the computer where to stop painting. You can specify the location using either relative or absolute coordinates. Using a relative address after the CIRCLE statement is easy because the graphic cursor is still located in the center of the circle. You can see how easily we fill circles with the next program.

```
REM 1.4.1 Fill Demo
SCREEN 1,320,200,5,1
WINDOW 2,,(0,0)-(311,185),16,1
RANDOMIZE TIMER
WHILE INKEY$=""
  f=INT(RND*32)
  CIRCLE (RND*311,RND*185),RND*100,f
  PAINT STEP (0,0),f
WEND
WINDOW CLOSE 2
SCREEN CLOSE 1
```

Compared to the rectangle program using LINE, this program fills in the color a little slower. The CIRCLE statement is one reason why this happens. However, the PAINT statement fills in color slower than the LINE statement with a box fill function. This is because the PAINT statement has to check every pixel for the shape border before painting.

You should be very careful when using PAINT. If the border of the area you want filled has even one hole in it, the entire screen can be painted. The same thing will happen if you specify a window type smaller than 16 (see PAINT in the BASIC handbook). In this case, if the shape

passes outside the right window border, the entire window is filled. To test this, change the 16 in the last program to a zero.

1.4.2 Another solution: AREA and AREAFILL

There is another way to fill an area besides using LINE and PAINT. Unlike with PAINT, we do not need a border shape, instead we need single pixels as corner points.

There are two statements used in this method. The first statement, AREA, sets all the corners. This statement is easier to use than PSET because no colors are needed.

```
AREA (10,30)
AREA (199,140)
AREA STEP (200,-30)
```

As you can see, the AREA statement is used relatively.

The second statement, AREAFILL, tells the computer to paint the defined area.

```
AREAFILL
```

After entering the four statements, you should see a white triangle on your screen.

Because the power of these statements is their ability to create figures with many corners, the triangle does not completely demonstrate this power.

The sequence of the corners you specify will determine what your figure will look like. Although the order of three points does not make much difference, with four points it is possible to have three different figures using the same coordinates.

```
REM 1.4.2A Three Possible
AREA (10,10)
AREA (30,140)
AREA (60,100)
AREA (50,20)
AREAFILL
WHILE INKEY$="": WEND
CLS
AREA (10,10)
AREA (60,100)
AREA (50,20)
AREA (30,140)
```

```
AREAFILL
WHILE INKEY$="":WEND
CLS
AREA (10,10)
AREA(60,100)
AREA(30,140)
AREA(50,20)
AREAFILL
```

The larger the number of corners, the greater the number of possible connecting lines. A pentagram, which has five points and can be drawn by hand in one motion, is a good example of this.

```
REM 1.4.2B Pentagram
AREA (100,20)
AREA (140,100)
AREA (20 ,45)
AREA (180,40)
AREA (40,110)
AREAFILL
```

You will notice that in the completed star only the points are white but the center is still blue. To explain this, we will use the same program again, only slightly changed:

```
REM 1.4.2C Framing by Drawing Twice
FOR i=0 TO 3
 AREA (10,30)
 AREA (199,140)
 AREA STEP (200,-30)
NEXT
AREAFILL
```

By changing the program we have set each corner twice and filled each area twice with AREAFILL. You might think that drawing twice works better, but this is not true. Instead of a filled area, we see only the border.

The pentagram is much the same. Like the triangle above, the pentagram's center is outlined twice and filled twice. Painting the same area a second time has the same result as erasing it.

A figure with only five corners isn't very complicated. But if we create a figure using 19 corners, it is difficult to detect all the corner points. By using random values, you can create figures that resemble modern art.

```
REM 1.4.2D 19 Corners
WINDOW 1, "Modern Art", (0,0)-(615,185),15
Start:
CLS
RANDOMIZE TIMER
```

```
FOR i= 0 TO 18
AREA (RND*611,RND*185)
NEXT i
AREAFILL
WHILE INKEY$="" :WEND
GOTO Start:
```

Since AREAFILL can only work on a maximum of 19 corners, if you try to set more than this, nothing will be displayed on the screen. After an AREAFILL statement, all the corner points are removed from memory and you can specify new ones.

1.4.2.1 Different modes of AREAFILL

AREAFILL has two different modes. The first mode, which you already know because we have been using it, always fills an area with the default foreground color. In our example we used white but you can change this by using the COLOR statement.

```
COLOR 2
```

Once this statement is executed, the next area is filled with black. This is the only way you can have a direct effect on the color of an area. Since it is the default mode, this mode does not require a lot of explanation. You can specify this mode with a trailing zero:

```
AREAFILL 0
```

The second mode of AREAFILL inverts the area instead of filling the area.

```
REM 1.4.2.1A Invert Demo
WINDOW 1, "Invert Demo", (0,0)-(615,185),15
PRINT "This is a Test!!"
PRINT "All Points inside"
PRINT "the Triangle will be"
PRINT "inverted!!"
CIRCLE (100,100),90,2
PAINT STEP (0,0),3,2
AREA (20 ,0)
AREA (180,45)
AREA (40,100)
AREAFILL 1
```

To indicate this mode enter a one after the AREAFILL statement. For each pixel on the screen there is a series of bits in memory that specify its color register. When we invert a pixel, each bit value changes to its opposite value. If a bit was equal to one it changes to zero and if it was

zero it changes to one. You can calculate the color register change to a pixel like this:

```
newcolor=(2^Depth-1)=oldcolor
```

The following program demonstrates how fast these color changes can take place:

```
REM 1.4.2.1B Speed
WINDOW 1,"Speed Test", (0,0)-(611,185),15
LOCATE 10,4
PRINT "Speed is not a Magic Trick!"
WHILE INKEY$=""
FOR i= 0 TO 2
  AREA (RND*611,RND*185)
NEXT i
AREAFILL 1
WEND
```

This program uses random corner points for a triangle and inverts everything inside it. The loop repeats and creates many triangles on top of each other in blue and orange. The result is a screen full of blue and orange checks. The longer the program runs, the more difficult it is to identify the individual shapes.

1.4.3 Patterns

When a computer fills an area, it sets one pixel after another until the area is filled with the selected color. Besides performing this type of area fill, the Amiga can also fill an area with a pattern designed by you.

Patterns can improve a graphic image or make part of the picture stand out. You can also create shadows and build textured walls easily with patterns.

Besides patterned areas, you can also draw patterned lines.

Both patterns and patterned lines require the same statement. Since the format is much the same for both, we will discuss patterned lines first.

1.4.3.1 Pattern creation

To create a pattern for a line, we use a 16 bit mask composed of 16 binary numbers (ones and zeros). Each value of one in the mask equals

a set pixel in the pattern and each zero equals an empty position. After 16 pixels, the mask pattern repeats. Therefore you can only draw where a mask bit is equal to one. This is similar to using a drawing template because you can only draw where there is a hole.

Since AmigaBASIC cannot manipulate binary numbers, we have to convert the binary mask value to hexadecimal. You could also convert the mask to decimal but this is more complicated. To convert to hexadecimal we have to work with four bits at a time. Our mask could look like this:

```
1011 0010 0000 1111
```

We convert each four bit block separately. To do this, we take the first bit and multiply it by eight. Then we multiply the second bit by four and add it to the first. We then multiply the third bit by two and add, and then add the last bit. The result of this operation is a number between 0 and 15. In hexadecimal, the numbers 10 thru 15 are converted to the letters A thru F. With our sample mask the conversion looks like this:

```
1*8 + 0*4 + 1*2 + 1 = B (11)
0*8 + 0*4 + 1*2 + 0 = 2
0*8 + 0*4 + 0*2 + 0 = 0
1+8 + 1+4 + 1+2 + 1 = F (15)
```

By vertically reading the end of the lines above we have B20F. We can try this out in our next program:

```
REM 1.4.3.1 Patterned Lines
PATTERN &HB20F
LINE (0,0)-(614,185)
LINE (20,30)-(104,105),2,B
```

As demonstrated, our pattern works not only on lines, but also on rectangles. The label "&H" in front of the mask value identifies the number as a hexadecimal value.

1.4.3.2 Patterned areas

The pattern format for areas is similar to the format for line patterns. The main difference is that areas are two dimensional. Because of this, you have to stack many 16 bit masks, one on top of the other, to build an area pattern. These masks are assigned using an array variable with the PATTERN statement.

The pattern for the lines is the first parameter. The pattern for the areas is the second parameter and defined in an integer array variable.

```
REM 1.4.3.2 Pattern Maker
DEFINT a
OPTION BASE 1
DIM a(8)
FOR i=1 TO 8
READ a(i)
NEXT i
PATTERN ,a
COLOR 3,1
LINE (0,0)-(614,185),,bf
WHILE INKEY$="": WEND
COLOR 1,0
CLS
DATA &h0,&h7FFF,&h7FFF,&h7FFF
DATA &h0,&hFF7F,&hFF7F,&hFF7F
```

The first thing you should do is make your array values all short integers. This means only numbers between -32768 and 32767, or in hexadecimal 0 to FFFF. If the variable type is wrong, you can create an error condition. To prevent this, we declare the array as short integer using `DEFINT` at the beginning of the program.

The `DIM` instruction tells `PATTERN` how many rows are in the pattern. You must dimension the array even if there are less than ten elements. The number of array elements also has to equal a power of two (1, 2, 4, 8, ...). Should there be one more or one less, you will get an "illegal function call" error. With normal arrays, you have to use the default array starting value of zero, such as 0-3, 0-7, etc. The ending index value is always smaller than the power of two. Or you can use:

```
OPTION BASE 1
```

to set the default starting value of all arrays to one.

1.4.3.3 Pattern design in a program

As you saw above, calculating the pattern values yourself is a lot of work. But when you have a computer that can do the work for you, why should you do it?

All you need is a small program that understands the binary pattern mask and converts it to hexadecimal or decimal. Since AmigaBASIC does not recognize binary, we have written a small subroutine. This subroutine converts character strings into short integers (numbers that

memory sees as two bytes). In the character string, you represent each zero with a space and any other character equals a one. We selected the left Amiga button as the shape for our pattern.

```
REM 1.4.3.3 Design im Listing
WINDOW 1, "Design patterns",(0,0)-(617,185),15
OPTION BASE 1
a=8
DIM f$(a)
REM    0123456789ABCDEF
f$(1)="             ***
f$(2)="            ****
f$(3)="          *  ***
f$(4)="        *    ***
f$(5)="       *********"
f$(6)="     **       ***"
f$(7)="  *****     *****"
f$(8)=""
REM    0123456789ABCDEF
CALL changeformat(f$(),a)
CIRCLE (400,140),100
PAINT STEP(0,0),2,1
AREA (150,160)
AREA (500,100)
AREA (570,170)
AREAFILL 1
MOUSE ON
WHILE INKEY$=""
IF MOUSE(0)<>0 THEN
  b=MOUSE(1)
  c=MOUSE(2)
  IF b>0 AND b<600 AND c>0 AND c<172 THEN
    LINE (b,c)-(b+4,c+4),1,bf
  END IF
END IF
WEND
SUB changeformat (fd$(1),g) STATIC
DIM fd%(g)
FOR i=1 TO g
  fd$(i)=fd$(i)+SPACE$(16)
  FOR j=0 TO 3
    h=0
    FOR k=0 TO 3
      IF MID$(fd$(i),j*4+k+1,1)<>" " THEN h=h+2^(3-k)
    NEXT k
    fd%(i)=fd%(i)+VAL("&h"+HEX$(h*2^(4*(3-j))))
  NEXT j
  PRINT i, HEX$(fd%(i)),fd%(i)
NEXT i
PATTERN ,fd%
END SUB
```

First we convert the character strings from spaces and asterisks into a pattern. Then we use this pattern to fill a circle and a triangle. The program also allows you to draw with the mouse, using the defined

pattern. When you press the mouse button you can draw a 4*4 pixel sized rectangle filled with the pattern.

Two parameters are passed into the subroutine that converts the character strings into hexadecimal values. The first parameter is the array where the mask will be stored and the second parameter is the number of elements in the array. We perform the conversion using the principles we explained earlier. The subroutine then passes the pattern to the PATTERN statement.

Since calculating a pattern mask every time you run a program can be troublesome, the above program is very useful as a pattern editor. For example, change the Amiga-A to any pattern you want to use. If you want a different pattern, change the strings again. The calculated pattern values are printed on the screen in decimal and hexadecimal. When the pattern generated is the one you want, write down the values and insert them in your own program.

1.4.3.4 Patterns and the cursor

Patterns are supposed to be used with fill functions and should only affect fill functions. However a side effect occurs when defining a pattern. The cursor is also changed. For example, in the previous program, the cursor appeared after the program printed the pattern values in the output window, but only as a small point. If you press the space bar you would see a dotted line instead of the normal pattern. These dots come from the Amiga-A pattern. If this changed cursor is disturbing, you can recover the normal cursor by adding the following lines to the pattern program:

```
REM Cursor reset
DIM norm%(2)
norm%(1)=&HFFFF
norm%(2)=&HFFFF
PATTERN ,norm%
```

It is very important that you use the statement "OPTION BASE 1" before running these program lines. If you do not use OPTION BASE, you must decrement all array values by one.

Of course you could do the opposite and change the cursor on purpose. You can make the cursor disappear when using INPUT or change it to a dashed line. To create a dashed line make the value of norm%(2) in the last program equal to zero. A half block cursor requires an array with eight elements. You set the first four values to zero and the last

four to &HFFFF for a bottom half block cursor. To make a top block cursor, just switch the four first and last values.

1.4.3.5 Bringing it together

You can write some fantastic programs using pattern filled areas. The below program is a good example of this.

```
REM 1.4.3.5A Stars and Stripes
DEFINT a-z
OPTION BASE 1
DIM a(16)
SCREEN 1,320,200,5,1
WINDOW 2,,(0,0)-(311,185),16,1
FOR i= 1 TO 16
  READ a(i)
NEXT i
PAINT (0,0),2
LINE (16,16)-(260,146),9,bf
FOR i= 0 TO 5
  LINE(16,26+i*20)-(260,36+i*20),1,bf
NEXT i
PATTERN ,a
LINE (16,16)-(111,80),1,bf
DATA 0,1536,3840,-16,16320,8064
DATA 6528,0,0,96,240,4095,1020
DATA 504,408,0
WHILE INKEY$="": WEND
WINDOW CLOSE 2
SCREEN CLOSE 1
```

An American flag appears shortly after starting this program. We defined the stars in a pattern containing two stars, one above the other. The top star is centered and the other is slightly offset so the rows are not vertically aligned.

As we hinted above, another possible use for patterns is to create 3-D effects. We have a small demonstration program to illustrate this:

```
REM 1.4.3.5B 3-D
DEFINT a-z
REM 3-D Cubes
OPTION BASE 1
DIM c(4)
SCREEN 1,320,200,3,1
WINDOW 2,,(0,0)-(311,185),16,1
COLOR 0,1
CLS
REM Pattern
c(1)=&H1010
```

```
c(2)=&H4040
c(3)=&H101
c(4)=&H404
PATTERN ,c
h=94
x=68
y=x/2
REM Large Cube
AREA (60,44)
AREA STEP (x,-y)
AREA STEP (x,y)
AREA STEP (0,h)
AREA STEP (-x,y)
AREA STEP (0,-h)
AREAFILL
SWAP c(1),c(4)
SWAP c(2),c(3)
PATTERN ,c
AREA STEP (0,0)
AREA STEP (-x,-y)
AREA STEP (0,h)
AREA STEP (x,y)
AREAFILL
REM Small Cube
COLOR 4
AREA STEP (0,-h/2)
AREA STEP (x/2,-y/2)
AREA STEP (0,-h/2)
AREA STEP (-x/2,-y/2)
AREA STEP (-x/2,y/2)
AREA STEP (0,h/2)
AREAFILL
SWAP c(1),c(4)
SWAP c(2),c(3)
PATTERN ,c
AREA STEP (0,0)
AREA STEP (x/2,-y/2)
AREA STEP (0,-h/2)
AREA STEP (-x/2,y/2)
AREAFILL
WHILE INKEY$="": WEND
WINDOW CLOSE 2
SCREEN CLOSE 1
```

You will see a 3-D cube standing on one of its corners. It appears that either a small cube is missing from the top front corner or that there is another cube stacked on top.

The patterns that we used to create the intersecting pieces are almost the same. The only difference is the order we assign the values. To change from one pattern to another, just reverse the order of the variables. In this program we used two SWAP-assignments to take care of this.

1.5 Medley of colors

The Amiga provides a palette of 4096 colors. If you use pure BASIC you can only use 32 colors at one time (we'll show you how to use more later). The specific colors are stored in the color registers. When you use a statement, you are not selecting what color to use but which color register to get the color from.

You cannot use all 32 colors in any type of screen. The number of possible colors depends on the depth of your screen. A normal Workbench screen has a depth of two, which allows you 2^2 colors or four colors. For every pixel there are two bits in memory which determine those four colors.

00	Color register 0
01	Color register 1
10	Color register 2
11	Color register 3

To display 32 colors you need five bits per pixel, but first you have to open a new screen with a depth of five:

```
SCREEN 1,320,200,5,1
WINDOW 2,"Title",(0,0)-(311,185),16,1
```

Now you could use all the statements you have learned so far with 32 colors.

Of the 32 colors available, you can select one for the foreground and one for the background.

```
COLOR 1,0
```

These are the two default values. The first sets the foreground color and the second sets the background color. All the graphic statements will use these two colors. As long as you leave the color parameter off, even PSET will use the default foreground color, and PRESET will use the default background color.

Changing the foreground and/or background, color does not instantly change the screen colors. Using the CLS statement makes the new colors take effect.

```
COLOR 2,3
CLS
```

The above statement gives you an orange screen with a black foreground color.

1.5.1 The complete palette

Most of the time you will probably want a completely different set of colors in your programs instead of the 32 default colors. To do this, you have to change the colors in the color registers.

Every color is composed of three values which set the red, green and blue of the color (RGB). Each of these values has a range from one to 16 or 16^3, which equals 4096 colors.

The BASIC statement used to change colors is PALETTE. You must specify the RGB (red, green and blue) values and which color register to change. The RGB color division must be a number between zero and one.

```
PALETTE 0,.75,1,0
```

This line sets a neon yellow background. This color is a mix of red and yellow; blue is not used. To set the normal blue background:

```
PALETTE 0,0,.3,.6
```

Setting the red range to one (1) and the other ranges to zero gives you red. This procedure works the same for the other two ranges.

Making all three color ranges equal produces a grey color. The lower you make the range value, the darker the color. Setting all three ranges to one (1) produces white.

1.5.2 Changing RGB colors

You can find any color you want by experimenting with the three color ranges. However, finding a specific color with this method could be tiring. The following program allows you to search for and find your target color. You can change all three ranges while the program is

running and quickly find your desired combination. The results are visible on the screen at the same time.

```
REM 1.5.2 ColorConstructor
SCREEN 1,320,200,2,1
WINDOW 2,"ColorConstructor",(0,0)-(297,185),31,1
LINE (200,20)-(300,150),3,bf
LOCATE 5
PRINT "> '4'   '5'   '6'"
LOCATE 16
PRINT "< '1'   '2'   '3'"
LOCATE 10
PRINT "   R     G     B"
r=1:g=.5:b=0 'orange
WHILE 1
 LOCATE 11
 PRINT USING "  #.##";r;g;b
 PALETTE 3,r,g,b
 WHILE a$=""
  a$=INKEY$
 WEND
  IF a$<>"" THEN
   IF a$="1" AND r>=.0666 THEN r=r-.06666
   IF a$="4" AND r<=.9333 THEN r=r+.06666
   IF a$="2" AND g>=.0666 THEN g=g-.06666
   IF a$="5" AND g<=.9333 THEN g=g+.06666
   IF a$="3" AND b>=.0666 THEN b=b-.06666
   IF a$="6" AND b<=.9333 THEN b=b+.06666
   a$=""
  END IF
WEND
```

You change the color ranges by using the 1-6 keys. The numeric keypad is perfectly arranged for changing color ranges. The 4 key raises the red range and the 1 key lowers it. The 5 and 2 keys raise and lower the green range, and the 6 and 3 keys raise and lower the blue range.

The color range will increase and decrease by a value of 0.0666 or 1/15. This insures a color change with every keypress.

You can also change the mouse pointer colors with PALETTE. The effected registers are 17, 18 and 19. Try it out.

1.5.3 The opposite of PALETTE

In many programs it is very important to know what colors are in the palette. However, there is no statement in BASIC that performs the opposite of PALETTE. Again we will step ahead a little and get our

data directly from memory. We will explain later, when we leave pure BASIC, how to determine the addresses of the color table.

Since the functions assigned at the beginning of the program take care of any addresses or PEEKs, you do not have to concern yourself with these. You just have to provide the color register number to the function. There are three functions; one for each RGB color range.

```
REM 1,5,3 Palette-Opposite
SCREEN 1,320,200,5,1
WINDOW 2,,,16,1
DEF FNcolortab=PEEKL(PEEKL(PEEKL(WINDOW(7)+46)+48)+4)
DEF FNred(f)=(PEEKW(FNcolortab+2*f) AND 3840)/3840
DEF FNgreen(f)=(PEEKW(FNcolortab+2*f) AND 240)/240
DEF FNblue(f)=(PEEKW(FNcolortab+2*f) AND 15)/15
PRINT "RGB Color Values:"
FOR i = 0 TO 31
  LOCATE 5+i MOD 16,1+INT (i/16)*20
  COLOR i
  PRINT USING "##";i,
  COLOR 1
  PRINT USING " #.##";FNred(i);FNgreen(i);FNblue(i)
NEXT i
WHILE INKEY$="":WEND
WINDOW CLOSE 2
SCREEN CLOSE 1
```

This program prints the color values for every color. The color register and color are listed in front of the three ranges.

1.5.4 Animation using color

Now we'll show you a way to make the functions from the last program useful. You can easily animate a picture by repeatedly swapping several colors. This color cycling trick may be familiar to you because many paint programs, such as GraphiCraft® and Deluxe Paint®, use it. You can magically display moving rivers or similar actions with this method. With some programming, this color swapping is possible in BASIC. However, you should keep the number of colors being swapped as small as possible because of BASIC's slowness. If you use too many colors, the cycling will be slower, causing a jerky and ragged look.

```
REM 1.5.4 Palette-Opposite Expanded
SCREEN 1,320,200,5,1
WINDOW 2,,,16,1
PALETTE 0,0,.5,0
PALETTE 28,0,.35,.72
```

```
PALETTE 29,0,.35,1
PALETTE 30,0,.5,1
PALETTE 31,0,.6 ,1
DEF FNcolortab=PEEKL(PEEKL(PEEKL(WINDOW(7)+46)+48)+4)
DEF FNred(f)=(PEEKW(FNcolortab+2*f) AND 3840)/3840
DEF FNgreen(f)=(PEEKW(FNcolortab+2*f) AND 240)/240
DEF FNblue(f)=(PEEKW(FNcolortab+2*f) AND 15)/15
f1=28:f2=31
FOR j=0 TO 311 STEP 4
FOR i=0 TO 3  STEP .5
 LINE (j+i,120+j/8)-(j+i-4,140+j/8),28+i
 LINE (j+i+1,120+j/8)-(j+i-3,140+j/8),28+i
NEXT i
NEXT j
rotation:
r1=FNred(f2)-.03
g1=FNgreen(f2)-.03
b1=FNblue(f2)-.03
FOR i=f1 TO f2
  r=FNred(i)-.03
  g=FNgreen(i)-.03
  b=FNblue(i)-.03
  PALETTE i,r1,g1,b1
  r1=r
  g1=g
  b1=b
NEXT i
GOTO rotation
```

This program draws an abstract river. We set the last eight color registers to blue for the river.

In the program routine "rotation" we subtract .03 from the received color range before using the value with PALETTE. This corrects a calculation error for the floating point value, that is between 0 and 15, which we received from memory.

1.6 All about PUT and GET

The PUT and GET statements have many uses, which include saving your graphics on disk and working with animation. However, all the PUT and GET capabilities have one thing in common: the management of screen information.

1.6.1 Using PUT and GET

With GET you can read a graphic from a specific region and with PUT you can draw it. We store the data in an array variable defined as integer. By using integer values between -32768 and 32767, building the data array is easier.

1. Array position = width
2. Array position = height
3. Array position = depth
4. Array position = bit-planes

Starting with the fourth array element are the bit-planes. The depth of a graphic determines how many bits are used for each pixel. The first bit-plane contains all the first bits, the second all the second bits, etc. This is also why the depth sets the number of bit-planes. The number of elements in array will differ greatly depending on the graphics mode, width values and height values.

To store data in integer form, we combine 16 bits of one bit-plane and one screen line. Since we cannot store more than this in a short integer, we have to divide the width by 16 when assigning array space. If there is a remainder (the width did not divide evenly by 16), we have to round off. The last pixel of a line is cut off if you do not round off. Now multiply this value with the height and depth to get the number of array elements. You will always have the three elements for width, height and depth. The following formula will calculate the number of array elements required:

```
Arrayspace=3+Height*Depth*INT((Width+15)/16)
```

The first program in this section simply stores the screen contents with GET, clears the screen and restores the old contents.

```
REM 1.6.1 Demo for GET and PUT
OPTION BASE 1
DEFINT f
CIRCLE (170,60),110
PAINT STEP (0,0),2,1
COLOR 1,2
LOCATE 5,10
PRINT "This demo shows"
PRINT TAB(10)"how to store"
PRINT TAB(10)"graphics to memory"
PRINT TAB(10)"and also how to"
PRINT TAB(10)"display them again"
PRINT TAB(10)"on the screen !!!"
AREA (140,20)
AREA (80,60)
AREA (300,80)
AREAFILL 1
COLOR 1,0
REM Save Graphic
x1=40 :y1=10
x2=300:y2=120
DIM feld(3+2*(y2-y1+1)*INT((x2-x1+16)/16))
GET (x1,y1)-(x2,y2),feld
REM Redisplay Graphic
FOR i=0 TO 140
  CLS
  PUT (i*3,i),feld
NEXT i
```

The first 20 lines of the above program draw the graphics. We used the top left and lower right corners of this image to calculate how much array space to dimension.

We can easily check whether or not our calculation assigned enough space. Instead of adding the three required values in the formula, try adding only two. As soon as the program starts, an "illegal function call" error will appear on the screen. This error will always appear when you fail to reserve enough space for GET. As long as you have plenty of memory, you can never assign too much space.

This demonstration shows you more than just how these statements work; you can also see how fast the graphics move around the screen. The motion is so smooth you can even read the moving text. As you have witnessed, moving graphics and animation is easy when you use PUT and GET.

1.6.2 Saving to disk

You have seen how to store screen information in memory using arrays. But we cannot just leave the data in memory, because when you turn the computer off the data will be lost. In order to permanently keep your graphic, you must save it on disk. The following program contains subroutines for saving and loading and can be used in other programs.

This program is also good for painting. When the program starts, you have a single pixel brush that has four colors. You can save your drawing to disk and then load it again to use as a brush.

```
REM 1.6.2 Paint Program
OPTION BASE 1
DEFINT a-z
PRINT "Draw with the mouse."
PRINT "When you want to save part"
PRINT "of your graphic, press the"
PRINT "'s' key. To load a graphic"
PRINT "back in, press the 'l' key."
PRINT "Press the 'e' key to exit."
WHILE a$<>"e"
  a$=INKEY$
  WHILE a$=""
    IF MOUSE(0)<>0 THEN
      PSET (MOUSE(1),MOUSE(2))
    END IF
    a$=INKEY$
  WEND
  IF a$="l" THEN GOSUB picload
  IF a$="s" THEN GOSUB picsave
  IF a$>="0" AND a$<"4" THEN COLOR VAL(a$)
WEND
END

picload:
    DIM fd(10000)
    WINDOW 2,,(0,0)-(600,0),16
    INPUT "Load File :";b$
    IF b$<>"" THEN
      CALL loader(b$,fd())
    END IF
    WINDOW CLOSE 2
    WHILE INKEY$=""
      IF MOUSE (0)<>0 THEN
        PUT(MOUSE(1),MOUSE(2)),fd
      END IF
    WEND
    ERASE fd
RETURN
```

```
picsave:
    WINDOW 2,,(0,0)-(600,0),16
    INPUT "Save File :";b$
    IF b$<>"" THEN
      PRINT "Position the corner points";
      PRINT "with the mouse";
1     IF MOUSE(0)=0 THEN 1
      ax=MOUSE(1):ay=MOUSE(2)
2     IF MOUSE(0)<>0 THEN 2
3     IF MOUSE(0)=0 THEN 3
      bx=MOUSE(1):by=MOUSE(2)
      WINDOW CLOSE 2
      CALL saver (b$,ax,ay,bx,by,2)
    ELSE
      WINDOW CLOSE 2
    END IF
RETURN

' The following SUBroutines work '
' without the main program.      '

SUB saver (n$,x1,y1,x2,y2,dp) STATIC
  e=3+(y2-y1+1)*dp*INT((x2-x1+16)/16)
  DIM graphic%(e)
  GET (x1,y1)-(x2,y2),graphic%
  OPEN n$ FOR OUTPUT AS #1
  FOR i=1 TO e
    WRITE #1, graphic%(i)
  NEXT i
  CLOSE #1
  ERASE graphic%
END SUB

SUB loader (filename$,graphic%(1)) STATIC
  OPEN filename$ FOR INPUT AS 1
  INPUT #1,graphic%(1),graphic%(2),graphic%(3)
  e=3+graphic%(3)*graphic%(2)*INT((graphic%(1)+15)/16)
  FOR i=4 TO e
    INPUT #1, graphic%(i)
  NEXT i
  CLOSE 1
END SUB
```

If you want to save the entire screen, you won't have enough memory assigned. To avoid this problem, use the following statement once before starting the program:

```
CLEAR ,40000
```

To be able to load a full screen again you also have to change a variable dimension. At the program label `picload` raise the `DIM` statement for `fd` to 15000.

There are also a few things you must do when you use the load and save subroutines in your own programs.

For the save subroutine, you have to pass the following variables: data name, upper left and lower right hand corner coordinates and the depth.

For the load subroutine you need the data name and an array that is large enough to hold the data. To prevent errors, it is much safer to assign an array larger than you need.

You can switch between, and use, load and save routines as often as you like. The ERASE statement at the end of the save subroutine handles any variable problems. Because variables retain their values between subroutine calls, a "duplicate definition" error would appear if we had not used ERASE.

We manage the loading and saving on disk with the standard data file handling statements OPEN, INPUT, WRITE and CLOSE. You use OPEN and CLOSE to open and close a disk data file. When you open a file, you also indicate if you are going to read or write data. To write data into a sequential file, use the WRITE statement. Each data value is sent to the file one after another, like entering characters on the keyboard.

When you are loading, load the width, height and depth before loading the bit-planes. Then calculate the size of the array and determine how many elements are loaded.

1.6.3 Alternate uses for PUT

In the last program you probably noticed an interesting effect caused by the PUT statement. Passing over an existing graphic on the screen caused a blending of the image. Maybe you also discovered that when you set the same graphic twice in the same place, the graphic is erased.

1.6.3.1 Default mode of PUT

Both of these effects are side effects of the PUT statement. Instead of painting over what is on the screen, PUT links the new pixel with the existing one by using XOR. The logic table for XOR looks like this:

```
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0
```

When two pixels line up exactly, the pixel is erased. A pixel is drawn only when a set and unset pixel are on top of each other. Since this is not always the best use of PUT, you can modify this effect. You can use additional statements, called *action-verbs* after PUT for specific needs.

The default "action-verb" for PUT is XOR as we demonstrated in the last program. Written out the statement would have the following syntax:

```
PUT (x,y),feld,XOR
```

You can easily move an object, without changing it, across a background and create quality animation. The ball in the next program does this.

```
REM 1.6.3.1 Moving Pictures with PUT
DEFINT g
COLOR 2,0
LOCATE 10,10
PRINT "This program moves a ball"
LOCATE 11,10
PRINT "around the screen without"
LOCATE 12,10
PRINT "changing the text !!"
DEFINT g
DIM g(250)
CIRCLE (20,20),20,1,,,.5
PAINT (20,20),3,1
GET (0,10)-(40,30),g
PUT (0,10), g, XOR
WHILE INKEY$=""
  FOR i=0 TO 180
    PUT (2*i,i), g ,XOR
    PUT (2*i,i),g,XOR
  NEXT i
WEND
```

When you use the PUT statement twice in the same location you get your old picture back.

This simple animation can be compared to the effects created by using sprites on a Commodore 64, but these effects are more difficult to create. You can achieve many other sprite type effects using different modes of the PUT statement.

The secret of the different modes and speed of the PUT statement is a graphic coprocessor called the *Blitter*. The Blitter can move data around in memory with incredible speed.

Not only does the Blitter move data fast, it can also perform operations like XOR at the same time.

1.6.3.2 A direct method using PSET

The Blitter can also skip any special operations and put the data directly on the screen. To achieve this direct, but not necessarily faster, method use the PSET mode.

Remove one of the PUT statements (`Put (2*i,i),g,XOR`) in the last program. Then replace the XOR in the next PUT line with PSET. Now you can see the difference. The graphic, which is drawn over the whole screen, erases nothing and draws over anything in its path. We have not only the ball, but the entire rectangle that we saved with the GET statement. Even the white locations change on the screen using PUT-PSET.

1.6.3.3 Inverting graphics

Similar to using PSET and PRESET for setting pixels, there is an opposite for PUT-PSET called PRESET. By using the PRESET mode you can effortlessly invert graphics. You only need two lines for this:

```
GET (X1,Y1)-(X2,Y2),g
PUT (X1,Y1),g, PRESET
```

The variable g is an integer that you must define before entering these lines so that enough memory will be assigned.

Inverting a graphic means inverting each bit of the graphic. All ones become zeros and all zeros become ones. This also causes the colors to come from different color registers. You can calculate the new color register like this:

```
New register=2 ^ depth of graphic - old register
```

How the colors change is demonstrated in this next small program:

```
REM 1.6.3.3A Color Change with PRESET
DEFINT f
DIM f(400)
SCREEN 1,320,200,5,1
WINDOW 2,,(0,0)-(287,30),16,1
FOR i=0 TO 31
LINE (i*9,0)-(i*9+8,40),i,bf
NEXT i
WHILE INKEY$=""
FOR i=31 TO 0 STEP -1
  GET (i*9,0)-(i*9+8,40),f
  PUT (i*9,0),f,PRESET
NEXT i
WEND
WINDOW CLOSE 2
SCREEN CLOSE 1
```

There are two different ways to restore a picture that you have inverted using PUT-PRESET. The first is to invert the same area again.

We use the method in the following program. A rectangle will wander around the screen. Every time it appears we invert the area it is in. Before the rectangle moves again we invert the area a second time so that the old background is visible again:

```
REM 1.6.3.3B Invert demo
DEFINT f
DIM f(100)
CIRCLE (160,80),100,1
PAINT STEP(0,0),2,1
LINE(120,50)-(180,160),1,b
PAINT (121,51),3,1
PAINT (179,159),1,1
WHILE INKEY$=""
  FOR i=0 TO 160
    GET (i,i)-(i+20,i+20),f
    PUT (i,i),f,PRESET
    FOR t=0 TO 200: NEXT
    GET (i,i)-(i+20,i+20),f
    PUT (i,i),f,PRESET
  NEXT i
WEND
```

The second method is much shorter and easier. After inverting, draw the graphic in the same position again using PUT-PSET.

1.6.3.4 AND or OR

The logic functions AND and OR are two additional modes for switching with PUT. Like XOR, they operate on the individual bits of a pixel,

which determine its color, instead of the whole pixel. If both the screen pixel and the pixel stored with GET are equal to one, then AND will have a value of one. The following combinations are possible with a depth of two and a maximum of four colors:

0 AND 0 =0
0 AND 1 =0
1 AND 0 =0
1 AND 1 =1

The following colors result:

If a pixel is blue (00), the second color has no effect; it stays blue (00).
Orange (11) and another color change the pixel to the other color.
When both colors are the same there is no change.
White (01) and black (10) changes to blue (00).

The following is a short program that will demonstrate these combinations:

```
REM 1.6.3.4 AND Switch
DEFINT f
DIM f(100)
CIRCLE (160,80),100,1
PAINT STEP(0,0),2,1
LINE(120,50)-(180,160),1,b
PAINT (121,51),3,1
PAINT (179,159),1,1
PRINT "test"
GET (0,0)-(39,8),f
FOR i=20 TO 160 STEP 8
  PUT (i,i),f,AND
NEXT i
```

With GET, we store the word "test" as a graphic. Then we PUT this graphic in various locations on a four colored screen. As in the table above, our white text is only visible with the orange and white backgrounds.

When we substitute OR for the AND we achieve an opposite effect. The text is visible in the blue and black backgrounds and the blue background of the text is not visible at all. The logic table for OR is:

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

1.7 Animation in BASIC

As an owner of an Amiga, you should be very proud of its highly praised animation capability. You can easily run animations even in BASIC by using the many built-in statements that start with OBJECT.

To create unusual effects you have to understand how these statements work and how to use them. The BASIC handbook lists all of the statements but does not show you everything you can do with them. For example, how you create a bob is left to the included program and the other statement explanations are too brief. Some very interesting things, like the COLLISION mask, are not even mentioned.

1.7.1 Sprites and bobs

The animation statements of BASIC are used for both sprites and bobs and are not limited to one or the other. Since sprites and bobs could be something new to you, here is a short definition.

Both sprites and bobs are movable graphics. Many of us have seen sprites on the Commodore 64 and other home computers. On the Amiga, a sprite can be 16 pixels wide, as high as the screen, and have up to three different colors. Sprites can move very quickly and require little programming effort to control.

Bobs are similar to sprites, but are designed for other uses. A bob can be any size you want and is limited only by the amount of available memory. Depending on the depth of the screen, a bob can have up to 32 colors. Bobs move slower than sprites and are displayed on the screen using different technology.

Another difference between bobs and sprites, which we'll discuss later, is a set or unset bit in a register. Using more than four sprites can get complicated, but the number of bobs you can use is limited only by the amount of memory.

1.7.2 The OBJECT.SHAPE statement

Unlike with the Comodore 64, you do not have to put the bob and sprite data into memory. The OBJECT-SHAPE statement does this for you. After entering all the required data using a character string, pass it to the OBJECT-SHAPE statement. This is the only method you can use in BASIC to design an object.

Because a character string that contains the data for a sprite or bob has a special format, the OBJECT-SHAPE statement will not understand just any character string. The following detailed explanation shows you how to define your own objects.

1.7.3 Designing an object

There are programs included with BASIC, like the OBJEDIT program, that make object editing easy for you. Although OBJEDIT is adequate for most uses, it cannot handle self-defined collision masks or shadow pictures. (We will explain later what these are and how you use them).

Why do you have all these fantastic possibilities but lack the software to use them? Read on; we can solve this problem.

1.7.4 The complete object editor: Eddi II

Eddi II, which has more options that OBJEDIT, is a program that lets you design any type of object. With this program you can create objects with a width of up to 309 pixels and a height of up to 183 pixels. You can also change the screen depth, which affects different parameters of the object and test your object while in the program. There are also many drawing statements that you can use for painting. When you are satisfied with your design, you can save it on disk as an object or as a graphic to be loaded with the PUT statement.

The program itself is more convincing than anything we can say about it. Because of the length of the program, you need a minimum of 512K bytes of memory to use it. The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
REM *****************¶
REM 1.7.4     Eddi II¶
REM *****************¶
REM¶
REM Jens Trapp   4/87¶
REM¶
CLEAR ,42000&     'Increase Work Buffer¶
OPTION BASE 1¶
DEFINT a-r,t-z¶
DEF FNe(b,h,t)=(3+t*h*INT((b+15)/16))¶
REM Sprite Colors¶
DIM sr(3),sg(3),sb(3)¶
sr(1)=1:sg(1)=1  :sb(1)=1    'white¶
sr(2)=0:sg(2)=0  :sb(2)=0    'black¶
sr(3)=1:sg(3)=.53:sb(3)=0    'orange¶
REM Change Menus¶
MENU 1,0,1,"Window"¶
MENU 1,1,1,"Load  "¶
MENU 1,2,1,"Save  "¶
MENU 1,3,1,"Size  "¶
MENU 1,4,1,"Test  "¶
MENU 1,5,1,"Delete"¶
MENU 1,6,1,"Quit  "¶
MENU 2,0,1,"Tools"¶
MENU 2,1,2,"  Points 0"¶
MENU 2,2,1,"  Lines   "¶
MENU 2,3,1,"  Frames  "¶
MENU 2,4,1,"  Boxes   "¶
MENU 2,5,1,"  Circles "¶
MENU 2,6,1,"  Fill    "¶
MENU 3,0,1,"Depth"¶
MENU 3,1,1,"  1  "¶
MENU 3,2,1,"  2  "  ¶
MENU 3,3,1,"  3  "¶
MENU 3,4,1,"  4  "¶
MENU 3,5,2,"  5  "¶
MENU 4,0,1,"Flags"¶
MENU 4,1,1,"  Sprite   "¶
MENU 4,2,1,"  Collision"¶
MENU 4,3,1,"  Shadow   "¶
MENU 4,4,2,"  Saveback "¶
MENU 4,5,2,"  Overlay  "¶
MENU 4,6,1,"  Savebob  "¶
MENU 4,7,1,"PlanePick  "¶
MENU 4,8,1,"PlaneOnOff "¶
MENU 4,9,0,"SpriteColor"¶
ON MENU GOSUB domenu   'Activate Menus¶
MENU ON¶
MOUSE ON¶
REM Default Values¶
sdepth    = 5          'Screendepth¶
gdep      = sdepth     'Depth of Graphic¶
wd        = 100        'width¶
ht        = 100        'height¶
thick     = 0          'brush thickness¶
DIM fd (FNe(wd,ht,gdep))¶
```

```
pcolor    = 1          'Color = white¶
fcolor    = 1          'Framecolor = white¶
tool=1                 'Default Tool = "point"¶
REM flags for standard objects¶
planepick = 2^gdep-1¶
saveback  = 1¶
overlay   = 1¶
REM Build Screen¶
SCREEN 1,320,200,sdepth,1¶
WINDOW 2,,,16,1¶
GOSUB makescreen¶
MainLoop:¶
    a$=INKEY$¶
    WHILE a$=""¶
      IF MOUSE(0)<>0 THEN GOSUB droutine¶
      a$=INKEY$¶
    WEND¶
REM Change Format¶
IF ASC(a$)=28 AND ht>1 THEN¶
  LINE (0,ht+1)-(wd+1,ht+1),8¶
  ht=ht-1¶
  fd(2)=ht¶
  LINE (0,0)-(wd+1,ht+1),fcolor,b¶
END IF¶
IF ASC(a$)=29 AND ht<184 THEN¶
  LINE (1,ht+1)-(wd,ht+1),0¶
  ht=ht+1¶
  LINE (0,0)-(wd+1,ht+1),fcolor,b¶
END IF  ¶
IF fvsprite=0 THEN¶
  IF ASC(a$)=31 AND wd>1 THEN¶
    LINE (wd+1,0)-(wd+1,ht+1),8¶
    wd=wd-1¶
    fd(1)=wd¶
    LINE (0,0)-(wd+1,ht+1),fcolor,b¶
  END IF  ¶
  IF ASC(a$)=30 AND wd<WINDOW(2) THEN¶
    LINE (wd+1,1)-(wd+1,ht+1),0¶
    wd=wd+1¶
    LINE (0,0)-(wd+1,ht+1),fcolor,b¶
  END IF    ¶
  IF ASC(a$)=139 THEN ¶
    ERASE fd¶
    DIM fd(FNe(wd,ht,sdepth))¶
    GET (1,1)-(wd,ht),fd¶
    LOCATE 24,1¶
    PRINT "Set the size using the mouse";¶
    WHILE MOUSE(0)=0 :WEND¶
    wd=MOUSE(1):IF wd>WINDOW(2) THEN wd=WINDOW(2)¶
    ht=MOUSE(2):IF ht>184 THEN ht=184¶
    GOSUB makescreen¶
  END IF¶
END IF¶
IF ASC(a$)>47 AND ASC(a$)<58 THEN thick=ASC(a$)-48: MENU
2,1,1-(tool=1),"  Points"+STR$(ASC(a$)-48)¶
IF a$<>"q" THEN MainLoop¶
```

```
endprog:¶
     WINDOW CLOSE 2¶
     SCREEN CLOSE 1¶
     MENU RESET¶
END¶
makescreen:¶
     COLOR 1,0:CLS¶
     fd(3)=gdep¶
     PUT (1,1),fd,PSET¶
sh2: LINE (0,0)-(wd+1,ht+1),1,b  'Graphic Frame¶
     LINE (wd+2,0)-(WINDOW(2),WINDOW(3)),8,bf¶
     LINE (0,ht+2)-(wd+2,WINDOW(3)),8,bf¶
GOTO tf                                    'Color Palette¶
domenu:¶
    title=MENU(0)¶
    pnt=MENU(1)¶
ON title GOTO wndow,tools,gdepth,flags1¶
RETURN¶
gdepth:¶
     MENU 3,gdep,1¶
     planepick=pnt^2¶
     IF pnt<gdep THEN¶
      ERASE fd¶
      DIM fd(FNe(wd,ht,sdepth))¶
      GET (1,1)-(wd,ht),fd¶
      gdep=pnt¶
      GOTO makescreen¶
    END IF¶
    gdep=pnt¶
tf: MENU 3,gdep,2¶
    LINE (0,186)-(WINDOW(2),WINDOW(3)),8,bf¶
    FOR i=0 TO 2^gdep-1¶
      LINE(30+i*8,186)-(37+i*8,195),i,bf¶
    NEXT i¶
    IF pcolor>2^gdep-1 THEN pcolor=1¶
    IF fcolor>2^gdep-1 THEN fcolor=1¶
GOTO colors2¶
wndow:¶
ON pnt GOTO loader,saver,size,test,del,endprog¶
RETURN¶
tools:¶
    MENU 2,tool,1¶
    tool=pnt¶
    MENU 2,tool,2¶
    IF tool=6 THEN¶
      fcolor=pcolor¶
      LINE (0,0)-(wd+1,ht+1),fcolor,b¶
      GOTO colors2¶
    END IF¶
RETURN¶
flags1:  ¶
ON pnt GOTO fvsprite1,coll1,shadmask1,saveback1,overlay1
,savebob1,planepick1,planeonoff1,sprcolor¶
RETURN¶
fvsprite1:¶
    fvsprite=(fvsprite+1)MOD 2¶
```

```
fvsprite2:¶
    IF fvsprite=1 THEN¶
      MENU 4,1,2¶
      MENU 4,9,1¶
      FOR i= 1 TO 3              ¶
        PALETTE i,sr(i),sg(i),sb(i)¶
      NEXT i  ¶
      FOR i=2 TO 8¶
        MENU 4,i,0¶
      NEXT i¶
      gdep=2¶
      IF wd<16 THEN¶
        f=0¶
      ELSE¶
        f=8¶
      END IF  ¶
      LINE (18,0)-(wd+1,ht+1),f,bf¶
      wd=16¶
      LINE (0,0)-(wd+1,ht+1),fcolor,b¶
      FOR i=1 TO 5¶
        MENU 3,i,0¶
      NEXT i¶
      MENU 3,2,2¶
      collmask=0¶
      shadmask=0¶
      saveback=0¶
      overlay=0¶
      savebob=0  ¶
      planeonoff=0¶
    ELSE¶
      MENU 4,9,0 ¶
      MENU 4,1,1¶
      PALETTE 1,1,1,1¶
      PALETTE 2,0,0,0¶
      PALETTE 3,1,.53,0¶
      FOR i=2 TO 8¶
        MENU 4,i,1¶
      NEXT i¶
      gdep=5¶
      MENU 3,0,1¶
      FOR i=1 TO 4¶
        MENU 3,i,1¶
      NEXT  ¶
      MENU 3,5,2  ¶
    END IF¶
    planepick=2^gdep-1¶
GOTO tf¶
coll1:¶
    collmask=(collmask+1)MOD 2¶
coll2:¶
    IF collmask=1 THEN ¶
      b=0¶
      CALL filename("CollisionMask",coll$,b)¶
      IF coll$="" THEN collmask=0¶
    END IF¶
    MENU 4,2,1+collmask¶
```

```
RETURN¶
shadmask1:¶
    shadmask=(shadmask +1)MOD 2¶
shad2:¶
    IF shadmask=1 THEN¶
      b=0¶
      CALL filename("ShadowMask",shad$,b)¶
      IF shad$="" THEN shadmask=0¶
    END IF¶
    MENU 4,3,1+shadmask¶
RETURN      ¶
saveback1:¶
    saveback=(saveback+1)MOD 2¶
    MENU 4,4,1+saveback¶
RETURN¶
overlay1:¶
    overlay=(overlay+1)MOD 2¶
    MENU 4,5,1+overlay¶
RETURN¶
savebob1:¶
    savebob=(savebob+1)MOD 2¶
    MENU 4,6,1+savebob¶
RETURN¶
planepick1:¶
    CALL planes("Planepick",planepick)¶
RETURN¶
planeonoff1:¶
    CALL planes("PlaneOnOff",planeonoff)¶
RETURN¶
sprcolor:¶
WINDOW 3,"Sprite Color",(0,0)-(200,40),16,1¶
INPUT "Color 1,2 or 3 ";a¶
IF a>0 AND a<4 THEN ¶
  INPUT "Red range   : ";sr(a)¶
  INPUT "Green range : ";sg(a)¶
  INPUT "Blue range  : ";sb(a)¶
  PALETTE a,sr(a),sg(a),sb(a)¶
END IF¶
WINDOW CLOSE 3¶
RETURN¶
droutine:¶
    x=MOUSE(1)¶
    y=MOUSE(2)¶
    xalt=MOUSE(3)¶
    yalt=MOUSE(4)¶
    IF y>185 THEN colors¶
    IF x>wd OR x<1 OR y>ht OR y<1 THEN RETURN¶
    ON tool GOTO pnte,w,w,w,w,fillroutine¶
w:  IF MOUSE(0)<>0 THEN¶
      f1=POINT(xalt,yalt)¶
      PSET (xalt,yalt),-(f1=0)¶
      f2=POINT(x,y)¶
      PSET (x,y),-(f2=0)¶
      PSET (x,y),f2¶
      PSET (xalt,yalt),f1¶
    ELSE¶
```

```
      ON tool GOTO droutine,linetool,frametool,
boxtool,circletool¶
    END IF¶
GOTO droutine¶
del:¶
    LINE (1,1)-(wd,ht),0,bf¶
RETURN¶
pnte:¶
    x1=x+thick:IF x1>wd THEN x1=wd¶
    y1=y+thick:IF y1>ht THEN y1=ht¶
    LINE (x,y)-(x1,y1),pcolor,bf¶
RETURN¶
linetool:¶
    LINE(x,y)-(xalt,yalt),pcolor¶
RETURN¶
frametool:¶
    LINE(x,y)-(xalt,yalt),pcolor,b¶
RETURN¶
boxtool:¶
    LINE (x,y)-(xalt,yalt),pcolor,bf¶
RETURN¶
circletool:¶
    IF y<>yalt AND x<>xalt THEN¶
      r=ABS(x-xalt):v=ABS(y-yalt)¶
      IF v<r THEN¶
        CIRCLE(xalt,yalt),r,pcolor,,,v/r¶
      ELSE¶
        CIRCLE(xalt,yalt),v,pcolor,,,v/r¶
      END IF¶
    END IF¶
GOTO sh2¶
fillroutine:¶
    PAINT(x,y),pcolor,fcolor¶
RETURN  ¶
colors:¶
    IF POINT(x,y)>=0 THEN pcolor=POINT(x,y)¶
colors2:¶
    LINE (0,186)-(25,195),pcolor ,bf¶
    LINE (0,186)-(25,195),fcolor,b¶
RETURN¶
size:¶
    WINDOW 3,"Size",(0,0)-(200,30),16,1¶
    PRINT "Width  = ";wd¶
    PRINT "Height = ";ht¶
    PRINT "Press a Key";¶
    WHILE INKEY$="":WEND¶
    WINDOW CLOSE 3¶
RETURN¶
test:¶
    ERASE fd¶
    DIM fd(FNe(wd,ht,sdepth))¶
    GET (1,1)-(wd,ht),fd¶
    CLS¶
    LOCATE 10,5¶
    PRINT "Working"¶
    GOSUB BFormat¶
```

```
    OBJECT.SHAPE 1,a$        ¶
    ON COLLISION GOSUB initobj¶
    COLLISION ON¶
    GOSUB initobj¶
    OBJECT.ON¶
    FOR i=0 TO 100¶
      COLOR INT(RND*32)¶
      LOCATE INT(RND*22)+1,RND*50 +1¶
      PRINT "EDDI II"¶
    NEXT i¶
    WHILE INKEY$="":WEND¶
    OBJECT.OFF¶
    OBJECT.STOP¶
    COLLISION OFF¶
GOTO makescreen¶
initobj:¶
    OBJECT.X 1,10¶
    OBJECT.Y 1,10¶
    OBJECT.VX 1,20¶
    OBJECT.VY 1,15 ¶
    OBJECT.START 1¶
RETURN¶
loader:  b=1¶
         MENU 3,gdep,1¶
         CALL filename("Load",n$,b)¶
         IF n$="" OR b=2 THEN RETURN¶
         OPEN n$ FOR INPUT AS #1¶
         IF b=0 THEN¶
           INPUT #1,wd,ht,gdep¶
           ERASE fd¶
           DIM fd(FNe(wd,ht,sdepth))¶
           FOR i=4 TO FNe(wd,ht,gdep)¶
             INPUT #1,fd(i)¶
           NEXT i¶
           fd(1)=wd¶
           fd(2)=ht¶
           fd(3)=gdep¶
           planepick=2^gdep-1¶
         ELSE¶
           ColorSet=CVL(INPUT$(4,1))   'These variables
are not¶
           DataSet=CVL(INPUT$(4,1))    'used by this
program.¶
           gdep=CVL(INPUT$(4,1))¶
           wd=CVL(INPUT$(4,1))¶
           ht=CVL(INPUT$(4,1))¶
           flags=CVI(INPUT$(2,1))¶
           planepick=CVI(INPUT$(2,1))¶
           planeonoff=CVI(INPUT$(2,1))¶
           ERASE fd¶
           DIM fd(FNe(wd,ht,gdep))¶
           fd(1)=wd¶
           fd(2)=ht¶
           fd(3)=gdep¶
           FOR i=4 TO FNe(wd,ht,gdep)¶
             fd(i)=CVI(INPUT$(2,1))¶
```

```
        NEXT i¶
      END IF¶
    IF flags AND 1 THEN¶
      fvsprite=1¶
      FOR i=1 TO 3¶
        a=CVI(INPUT$(2,1))¶
        sr(i)=(a AND 3840)/3840¶
        sg(i)=(a AND 240)/240¶
        sb(i)=(a AND 15)/15¶
      NEXT i  ¶
      GOSUB fvsprite2¶
    ELSE¶
      collmask=(flags AND 2)/2¶
      shadmask=(flags AND 4)/4¶
      saveback=(flags AND 8)/8¶
      overlay=(flags AND 16)/16¶
      savebob=(flags AND 32)/32¶
      MENU 4,1,1¶
      MENU 4,2,1+collmask¶
      MENU 4,3,1+shadmask¶
      MENU 4,4,1+saveback¶
      MENU 4,5,1+overlay¶
      MENU 4,6,1+savebob¶
      IF shadmask=1 THEN¶
      b=0:shad$=""¶
      CALL filename("ShadowMask",shad$,b)¶
      IF shad$<>"" THEN¶
        OPEN shad$ FOR OUTPUT AS 2¶
          PRINT #2,wd;ht;1;¶
      END IF¶
      FOR i=4 TO FNe(wd,ht,1)¶
        a=CVI(INPUT$(2,1))¶
         IF shad$<>"" THEN PRINT #2,a;¶
       NEXT i¶
       IF shad$<>"" THEN CLOSE 2¶
      END IF    ¶
      IF collmask=1 THEN¶
          b=0:coll$=""¶
          CALL filename("CollisionMask",shad$,b)¶
          IF coll$<>"" THEN¶
            OPEN coll$ FOR OUTPUT AS 2¶
            PRINT #2,wd;ht;1;¶
            FOR i=4 TO FNe(wd,ht,1)¶
             PRINT #2,CVI(INPUT$(2,1));¶
          NEXT i¶
          CLOSE 2¶
        END IF¶
      END IF    ¶
    END IF¶
    CLOSE 1¶
GOTO makescreen¶
saver:¶
     b=1¶
     ERASE fd¶
     DIM fd(FNe(wd,ht,sdepth))¶
     GET (1,1)-(wd,ht),fd¶
```

```
      CALL filename("Save",n$,b)¶
      IF n$="" OR b=2 THEN RETURN¶
      OPEN n$ FOR OUTPUT AS 2¶
        IF b=1 THEN¶
        GOSUB BFormat¶
        PRINT #2,a$;¶
        ELSE  ¶
          PRINT #2,wd,ht,gdep¶
          FOR i=4 TO FNe(wd,ht,gdep)¶
            PRINT #2,fd(i)¶
          NEXT i¶
        END IF¶
        CLOSE 2  ¶
RETURN¶
BFormat:¶
    a$=MKL$(0)+MKL$(0)¶
    a$=a$+MKI$(0)+MKI$(gdep)¶
    a$=a$+MKI$(0)+MKI$(wd)¶
    a$=a$+MKI$(0)+MKI$(ht)¶
    flags=fvsprite+2*collmask+4*shadmask+8*saveback¶
    flags=flags+16*overlay+32*savebob¶
    a$=a$+MKI$(flags)¶
    a$=a$+MKI$(planepick)¶
    a$=a$+MKI$(planeonoff)¶
    FOR i=4 TO FNe(wd,ht,gdep)¶
      a$=a$+ MKI$(fd(i))¶
    NEXT i¶
    IF shadmask THEN¶
      IF shad$="" THEN GOSUB shad2¶
      OPEN shad$ FOR INPUT AS 1¶
      INPUT #1,b,h,t¶
      IF b<>wd OR h<>ht THEN¶
        LOCATE 10,4¶
        PRINT "Not in ShadowMask Format!"¶
        CLOSE 1¶
        WHILE INKEY$="":WEND¶
        GOTO makescreen¶
      END IF¶
      FOR i=4 TO FNe(wd,ht,1)¶
        INPUT #1,a¶
        a$=a$+MKI$(a)¶
      NEXT i¶
      CLOSE 1¶
    END IF¶
    IF collmask THEN¶
      IF coll$="" THEN coll2¶
      OPEN coll$ FOR INPUT AS 1¶
      INPUT #1,b,h,t¶
      IF b<>wd OR h<>ht THEN¶
        LOCATE 10,4¶
        PRINT "Not in CollisionMask Format!"¶
        CLOSE 1¶
        WHILE INKEY$="":WEND¶
        GOTO makescreen¶
      END IF¶
      FOR i=4 TO FNe(wd,ht,1)¶
```

```
        INPUT #1,a¶
        a$=a$+MKI$(a)¶
      NEXT i¶
      CLOSE 1¶
    END IF¶
    IF fvsprite THEN¶
      a$=a$+MKI$(INT(sr(1)*15)*256+INT(sg(1)*15)
*16+sb(1)*15) 'SprColor 1¶
      a$=a$+MKI$(INT(sr(2)*15)*256+INT(sg(2)*15)
*16+sb(2)*15) 'SprColor 2¶
      a$=a$+MKI$(INT(sr(3)*15)*256+INT(sg(3)*15)
*16+sb(3)*15) 'SprColor 3¶
    END IF¶
RETURN¶
SUB planes (b$,p)  STATIC¶
    WINDOW 3,b$,(0,0)-(150,24),16,1¶
    PRINT "End with <RETURN>"¶
pl: FOR i=0 TO 4¶
      LOCATE 2,2+i*2¶
      PRINT i;¶
      LOCATE 3,2+i*2¶
      PRINT (2^i AND p)/2^i ;¶
    NEXT i¶
    a$=""¶
    WHILE a$=""¶
      a$=INKEY$¶
    WEND¶
    IF ASC(a$)>=48 AND ASC(a$)<53 THEN p=p XOR
2^(ASC(a$)-48)¶
    IF ASC(a$)<>13 THEN pl¶
    WINDOW CLOSE 3¶
END SUB¶
SUB filename (b$,c$,d) STATIC¶
    WINDOW 3,b$,(0,0)-(300,40),16,1¶
    IF c$<>"" THEN¶
      PRINT "Old "b$"Filename :"¶
      PRINT c$¶
      PRINT "<RETURN> for same name"¶
    END IF¶
    INPUT "New Filename :",d$¶
    IF d$<>"" THEN c$=d$¶
    IF d=1 THEN¶
      PRINT "Bob or Put Format"¶
      INPUT "(b/p)";d$¶
      IF d$="b" THEN ¶
        d=1 ¶
      ELSE¶
        IF d$="p" THEN¶
          d=0 ¶
        ELSE¶
          d=2¶
        END IF¶
      END IF¶
    END IF¶
    WINDOW CLOSE 3¶
END SUB¶
```

The most important variables:

`fd` Short integer array. This field stores the screen information and is easily used with `PUT` and `GET` to display the graphic. `Fd(1)` is the width, `fd(2)` the height, and `fd(3)` the depth. The size of the array is self-defined using a function.

`wd` Width of the graphic. Since our graphic always starts with (1,1) we can increase to the maximum value while drawing.

`ht` Height and last Y value of the graphic.

`depth` Actual depth of the graphic. This value has no effect on the screen depth.

`sdepth` Actual depth of the screen. This value stays the same.

`x` X coordinate when drawing.

`y` Y coordinate when drawing.

`xalt` Starting coordinate for drawing lines, etc.

`yalt` Y coordinate identical use as `xalt`.

`pcolor` Actual drawing color.

`fcolor` Color of the frame when filling. The fill function uses the frame color.

`tool` The number of the actual drawing statement.

`title` Number of the active menu.

`pnt` Number of the active submenu.

`b` Help variable for format decisions.

`thick` Brush width for "point".

`n$` Data name for loading and saving.

`coll$` Data name for the `COLLISION` mask (see below).

`Shad$` Data name for the shadowmask (see below).

All object variables and flags will be explained in the following sections.

1.7.4.1 The screen

When you start the program your design board will appear on the screen (Eddi's own window and screen). In this window you will see two things:

1.) all 32 colors in small boxes at the bottom of the screen and
2.) a large window framed in white. This window is where you can design all your objects.

1.7.4.2 Object size

The first thing you should do is set the size of your object. You can change this size later, but you should have an idea of the form and size you want before you start.

To find out the current size of your object, activate the menu item `Size` in the `Window` menu, which displays the width and height. You can change these values in two different ways by using the keyboard.

1. The cursor keys. The size of the object is increased or decreased.

2. First press the help key and then use the mouse pointer to indicate the new lower right hand corner of the object.

The top left hand corner of the object is always in the top left hand corner of the screen and cannot be changed. You can shrink both height and width all the way down to one. You can even define an object as small as one pixel, which is the smallest possible object.

The maximum size of 312*184, if you have enough memory, is large enough for most uses.

1.7.4.3 Depth

Depth is also important to the format of your object because it determines the maximum number of colors the object can have. For example, with a depth of five you can have up to 32 colors. Depth is processed in a special way in this program and has its own menu that allows you to check or change the current depth.

A checkmark in the depth menu indicates the actual depth. You can change this the same way you select items in the Workbench menus.

When you change the depth, the number of available colors and the length of the color bar also changes. Any full color object that you currently have in the design window is also affected.

1.7.4.4 Colors

Look closely at the row of available colors and you can see that the first square is slightly larger than the rest. This square indicates the current drawing color. To select a different drawing color just click with the mouse on the color you want.

1.7.4.5 Drawing

Finally we have reached the main part of the program: drawing. There are six drawing modes located in the "`Tools`" menu. This menu contains just about everything you will need to draw wonderful pictures. To find the graphic statements we have already discussed, look through the program listing. All of the following statements use the current selected drawing color:

`Points N` `POINT` is a drawing statement you learned in some of the demonstrations. When you click the left mouse button, you will set a pixel where the mouse pointer is located. This statement also performs a special function. Instead of just setting pixels, you can set blocks of pixels up to 9*9 in size. The number after "point" determines the brush size and can be changed from the keyboard using the number keys (0-9).

`Lines` This menu item works exactly like the BASIC `LINE` statement. You click and hold the left mouse button wherever you want to start a line, then move to where you want the line to end and release the button. While you hold the button, the start and end points of the line will blink. This allows you to easily determine the direction and level of the line.

`Frames` This statement draws an unfilled rectangle. You set the upper left and lower right hand corners in the same way you set the ends of the line above.

`Boxes` Boxes performs the opposite of frames by drawing a filled rectangle. You accomplish the sizing the same way.

`Circles` The circle menu item lets you set your parameters using the mouse which is much easier than the BASIC method. Sizing is similar to the line statement, but with circle, your start point will be the center of the circle. The point where you release the mouse button sets the maximum height and width of the circle. However, this release point is never part of the circle. The X and Y difference in relation to the release point determines the horizontal and vertical radius. Therefore, the two points you set with the mouse mark exactly a quarter of the circle.

`Fill` The last option fills any framed area. Unlike `OBJEDIT`, this fill statement allows you to fill an area with a different color than the frame of the area. When you select fill, the program automatically selects the current drawing color as the frame and fill color. If you select a different color while "fill" is active, the frame color does not change. The program indicates the current color by a frame around the drawing color and around the object.

Use the `Delete` option from the `Window` menu to clear the screen when you are not satisfied with an object design.

1.7.4.6 Loading and saving

In order to keep your masterpiece you need load and save options. The BASIC statements GET and PUT are perfect for this.

Actually, you can save and load your data in two different formats. The first format uses the PUT statement to save data in short integer form. As you saw in previous demonstrations, the data is easily loaded using the GET and PUT statements and is displayed in original form on your screen.

The second format is designed for bobs and is used to create files for bobs and sprites. Besides the raw picture information, these files contain many other values for objects and are displayed using the OBJECT-SHAPE statement.

The program remembers the last filename used for a load or save. To load or save using the same filename you only have to press the <Return> key.

1.7.4.7 Testing objects

The Test option in the Window menu enables you to test your new objects without having to exit the editor. You can make quick changes easily if the object is not exactly what you want.

You do not have to save an object to disk before testing it. We take the data directly from the screen and convert it to the OBJECT-SHAPE statement format. Therefore, you can test it out before you save it. To exit the test mode, press any key.

1.7.4.8 Exiting Eddi II

Even exiting the program has two options. You can select Quit from the Window menu, or simply press the <q> key. Remember that exiting the program clears all data so be sure you have saved your object first.

Upon exiting, the window and screen for the program are closed and the normal BASIC menus are activated again.

1.7.4.9 Loading objects into your programs

To load the objects as bobs you must save them in bob format. Then it is quite easy to load them again using the OBJECT.SHAPE statement.

```
OPEN "Filename" FOR INPUT as 1
OBJECT.SHAPE 1,INPUT$(LOF(1),1)
CLOSE 1
```

1.7.5 Flags

The flags determine the appearance of an object on the screen. Each flag has two possible settings. It is either set and equals one or not set and equals zero. Several flags are combined into one byte and all flags are passed to the computer in the character string that is used with the OBJECT-SHAPE statement. Once set, flags cannot be modified from BASIC.

Our editor has its own menu for the flags. Set flags are indicated by a small check in the menu.

There are two other functions in the Flag menu besides the flags: PlanePick and PlaneOnOff. We will explain a bit later what these two do.

1.7.5.1 The SaveBack flag

We will discuss the SaveBack flag first because it is the simplest one. All flags have names that help you remember what they do. For example, SaveBack is an abbreviation for "save the background".

When you draw a bob it becomes part of the existing screen. What was on the screen is painted over. To save and restore what the bob covers when it is moved, you have to set the SaveBack flag.

When the SaveBack flag is not set, and you move the object, the original object image will still be on the screen. Test this out with one of your objects in the editor.

When this flag is set and you are moving large objects, there will be a flickering on the screen. Also, screens with a lot of depth will have the same effect. Since there is no way to counter this flicker while using BASIC, you should set this flag only if you really need it.

At this point we need another sample program. To keep our program short, we have used smaller objects. The number of data statements required for bigger objects would make this demonstration program too large.

```
REM 1.7.5.1 Airplane
DEFINT a
SCREEN 1,320,200,2,1
WINDOW 2,,,16,1
PRINT "Reading in DATA..."
FOR i=1 TO 313
  READ a
  a$=a$+MKI$(a)
NEXT i
LOCATE 10,1
PRINT "In this Object demo you will see an"
PRINT "airplane that flies across the"
PRINT "screen, without having any effect"
PRINT "on this text."
OBJECT.SHAPE 1,a$
ShowOBJ:
OBJECT.X 1,1
OBJECT.Y 1,80
OBJECT.VX 1,3
OBJECT.VY 1,20
OBJECT.AX 1,4
OBJECT.AY 1,-2
OBJECT.ON
OBJECT.START
WHILE INKEY$="":WEND
GOTO ShowOBJ
REM Data for the airplane
DATA 0,0,0,0,0,2,0,88,0,25
DATA 8 :REM This is the value for the flags
DATA 3,0,8160,0,0,0,0,0,16368,0,0,0,0
DATA 0,16376,0,0,0,0,0,16380,0,0,896,0
DATA 0,16382,0,0,-1,-32768&,0,16382,0,1,-28668,-16384
DATA 0,16383,0,3,4100,24576,0,16383,0,6,4100,12288
DATA 0,16383,-32768,12,4100,6144,0,16383,-
16384,24,4100,3072
DATA 0,16383,-16384,48,4100,2044,0,16383,-2048,127,-1,-1
DATA 0,16383,-1,-1,-1,-14337,0,16383,-1,-1,-2048,12543
DATA 0,16383,-1,-4,1023,-385,0,4095,-1,-31,-1,-129
DATA -32768,4095,-1,-497,-1,-129,-32768,1023,-1,-257,-1,-
129
```

```
DATA -32768,1023,-1,-257,-1,-129,-32768,1023,-1,-129,-1,-
385
DATA -32768,511,-1,-249,-1,-257,0,7,-1,-32,0,504
DATA 0,0,0,16383,-1,-32,0,0,0,0,2044,0
DATA 0,0,0,0,0,0,0,0,0,0,0,0
DATA 0,0,0,0,0,0,0,0,0,0,0,0
DATA 0,0,0,0,0,0,0,8176,0,0,0,0
DATA 0,1536,0,0,28667,0,1024,1648,0,0,-4101,-32768
DATA 1024,1736,0,0,-4101,-32768,1024,1728,0,0,28667,8192
DATA 1024,1728,0,0,28667,-32768,1024,1728,0,0,28667,-
32768
DATA 1024,1728,0,0,0,0,1024,1736,0,0,0,0
DATA 2048,1648,0,0,0,0,2048,0,0,0,0,0
DATA 14336,0,0,0,0,127,-2048,0,0,0,0,127
DATA -2048,3,-1,-512,0,0,26624,1023,-1,-512,0,0
DATA 2048,0,0,256,0,0,3072,0,0,0,0,0
DATA 1024,0,0,0,0,0,1024,0,0,0,0,0
DATA 1024,0,0,0,0,0,1024,0,0,0,0,0,1024
```

We have included the OBJECT statements in this program. You have to use most of these statements for any object. Here are the individual explanations:

`OBJECT.SHAPE` The character string `a$` contains all the necessary information you have to pass to the computer. When you want to use an object you just specify the number assigned to it with `OBJECT.SHAPE`. In our program that number was 1. Previously we demonstrated how you load the object data from disk.

`OBJECT.X/Y` With these statements you set the start position. The first parameter sets the object number.

`OBJECT.VX/Y` This statement sets a speed (velocity) for the selected object.

`OBJECT.AX/Y` The velocity of an object does not have to be constant. You can set an acceleration factor with this statement.

`OBJECT.ON` Makes the specified object visible on the screen.

`OBJECT.START` Even though you have set a velocity and acceleration for your object, it will not move until you execute this statement.

This program is a very good demonstration of the `SaveBack` flag. The text remains visible even though the airplane flies right over the text.

If you want to see what happens when the SaveBack flag is not set, change the 11th value (the only value in the second DATA line) in the data list. Change the eight to zero. Now when you run the program one corner of the airplane will leave a trail on the screen.

1.7.5.2 SaveBob

The SaveBob flag is almost the opposite of SaveBack flag. When you set SaveBob, the object remains on the screen. You might be familiar with this effect from some of the better paint programs. This method allows you to draw while an object is on the screen.

1.7.5.3 Overlay

In our last sample program you probably noticed that when the airplane flew over the text it covered a much larger area than the actual size of the airplane. This effect can be removed with the Overlay flag. When the Overlay flag is set, all the unset pixels of the object become transparent. You can then see the background through the object wherever there are no set pixels. To test this out on our airplane, set the overlay flag by adding 16 to the eight of the SaveBack flag. The new flag becomes 24 and both flags are active.

This is only one of many possibilities you have with Overlay. The other uses are connected to the Shadow mask.

1.7.5.4 The Shadow mask

The Shadow mask is one of two masks that you can define for each object. With the Shadow mask you can specify which pixels the background covers and which pixels only change the color of the background. Also, if you have set the Overlay flag, the Shadow mask determines which pixels of the object are visible.

The mask must have the same width and height as the object you use with it. In format, a mask and object are the same, except the mask depth is independent of the object depth. Each bit in the mask is equal to one pixel of the object. If a bit in the mask is set, the corresponding

pixel in the object is on top of the background. The Overlay flag handles unset pixels. If the Overlay flag is set, the object pixel has no effect on the background. If the Overlay flag is unset, the color of the background pixel is changed.

If the Overlay flag is set the computer will create a default Shadow mask (if you have not created one). In this mask, all bits are set to match the set pixels of the object. This makes all the unset pixels of the object transparent.

When you want to create your own Shadow mask with the editor you should use the following steps:

1. First, you must already have an object for the mask. The best way to do this is to load your object first which gives you the correct format for the mask.
2. Set the depth to one.
3. Draw the mask. Any remaining pixels from the object can assist you with this.
4. Save the mask to disk in P format.
5. Load the object that the mask is for again.
6. Now set the Shadow mask flag on, in the Flags menu and enter the name used to save your mask.
7. Save the object in B format or test it.

Of course if you already have a mask designed you can skip the first five steps of this list.

1.7.5.5 Collisions

The second mask is the Collision mask. This mask, which has nothing to do with the OBJECT.HIT statement which we will discuss later, determines where the computer "senses" a collision. When the Collision mask is empty, the computer cannot see a collision. When you define a Collision mask, the set bits of the mask are the sensitive points.

You can design and install a Collision mask in the same way as the Shadow mask. If you don't define a Collision mask, the computer will use a default Collision mask. The default mask is built by using a logic OR function on all the bit-planes of the bob (exactly like the Shadow mask). The computer then reacts to contact with any set pixel of the object.

In our next program we'll demonstrate other methods, other than using the `Collision` mask, for controlling collisions. We'll test these statements using two squares that move around the screen.

```
REM 1.7.5.5 Collision
DEFINT a
RANDOMIZE TIMER
SCREEN 1,320,200,2,1
WINDOW 2,,,16,1
PRINT "Reading in DATA..."
FOR j= 1 TO 2
FOR i=1 TO 13
  READ a
  a$(j)=a$(j)+MKI$(a)
NEXT i
  a$(j)=a$(j)+MKI$(-1)+MKI$(-1)
  FOR i=1 TO 30
    a$(j)=a$(j)+MKI$(-32768&)+MKI$(1)
  NEXT
  a$(j)=a$(j)+MKI$(-1)+MKI$(-1)
NEXT j
FOR i=1 TO 30
a$(2)=a$(2)+MKI$(0)
NEXT i
a$(2)=a$(2)+MKI$(1)+MKI$(-32768&)
a$(2)=a$(2)+MKI$(1)+MKI$(-32768&)
FOR i=1 TO 30
a$(2)=a$(2)+MKI$(0)
NEXT i
CLS
OBJECT.CLIP (70,30)-(230,190)
LINE (70,30)-(230,190),3,b
ON COLLISION GOSUB collroutine
COLLISION ON
OBJECT.SHAPE 1,a$(1)
OBJECT.SHAPE 2,a$(2)
OBJECT.PRIORITY 1, 1
OBJECT.HIT 1,3,2
OBJECT.HIT 2,2,2
OBJECT.X 1,150
OBJECT.Y 1,80
OBJECT.X 2,155
OBJECT.Y 2,85
OBJECT.VX 1,8
OBJECT.VY 1,4
OBJECT.START 1,2
OBJECT.ON 1,2
WHILE INKEY$=""
LOCATE 2,15
PRINT OBJECT.X(1);TAB(21);OBJECT.Y(1)
PRINT TAB(15);OBJECT.X(2);TAB(21);OBJECT.Y(2)
WEND
WINDOW CLOSE 2
SCREEN CLOSE 1
END
```

```
collroutine:
n=COLLISION(0)
m=COLLISION(n)
more:
IF n=1 AND m<0 THEN
  BEEP
  IF ABS(m) MOD 2=0 THEN
    OBJECT.VX 1,(m+3)*(RND*20+1)
  ELSE
    OBJECT.VY 1,(m+2)*(RND*20+1)
  END IF
END IF
OBJECT.VX 2,OBJECT.VX(1)+3*SGN(OBJECT.X(1)-1-OBJECT.X(2))
OBJECT.VY 2,OBJECT.VY(1)+3*SGN(OBJECT.Y(1)-1-OBJECT.Y(2))
OBJECT.START
n=COLLISION(0)
m=COLLISION(n)
IF m<>0 THEN more
RETURN
DATA 0,0,0,0,0,1,0,32,0,32
DATA 24,1,0
DATA 0,0,0,0,0,1,0,32,0,32
DATA 10,1,0
```

While the two squares dash around the screen, we can display their coordinates by using the `OBJECT.X` function. Or you could also display the objects' velocity using the `OBJECT.VX` function. The same function is available for the Y direction. These functions are very important when you want a program to react to a collision in a certain way.

In order for a collision to occur, you always need two things. In our program we have two objects, both objects are squares, the same size, the same color, and are only a frame. Even though the objects seem similar, they have differences. Since we did not specify a collision mask for the first object, the computer provides a default collision mask using the `OR` function on its bit-planes. This collision mask consists of only the frame of the object.

For the second object we design our own collision mask. In this mask, we set only the four center pixels of the object. The program places the second object inside the first and whenever the second object tries to escape from the first there is a collision.

By using the `OBJECT.CLIP` statement, we confined the first object to a specific screen area. Because of this, the first object cannot move just anywhere on the screen.

Without the `COLLISION` functions, we would not be able to react to collision. The three `COLLISION` functions are the only object related statements that do not start with `OBJECT`. The first statement, `ON`

COLLISION GOSUB, tells the computer where to branch. However, this statement will not function without the second statement, COLLISION ON, which tells the computer that, if there is a collision, then branch. If COLLISION ON has not been used, the object will simply stop moving at the border of the specified area.

The third function helps us determine which two objects have collided. You can read the first object number with COLLISION(0) and the second object number with COLLISION(n). The n must be the number of the object that was involved in the collision. To determine the other object involved in the collision, substitute COLLISION(0) for the (n) in the COLLISION function. The value that is returned is the number of the second object. A collision with the border will return a value smaller than one.

COLLISION(COLLISION(0))	Border
-1	top
-2	left
-3	bottom
-4	right

Sometimes it is possible for the second bob to escape from the first bob. When the second object's speed is very fast, the collision mask can jump over the other object's mask without touching it. Because the border does not affect it, the second bob can leave the defined screen area and enclose the first object. To determine which collision will cause this breakout, use the OBJECT.HIT statement. Specify two 16 bit masks. The first is named the MeMask and the second is named the HitMask. When two objects collide with another, the MeMask of one is compared with the HitMask of the other. If both masks have a one in the same position, a program interrupt occurs. An interrupt will also happen when an object's HitMask is set equal to one and it collides with a window or area border set with OBJECT.CLIP.

Instead of using a bit pattern, you set these masks using a number between -32768 and 32767. The default value of these masks is -1 which means a full mask. The end result is an interrupt on any collision.

The final new statement in this program does not have a direct connection to collisions. This statement lets you specify the sequence in which to draw the objects on the screen. This is mostly important for deciding which objects are to appear first. The statement is OBJECT.PRIORITY and uses a default value of zero. The higher the priority number, the earlier an object is drawn.

1.7.5.6 Animated bit-planes

We have mentioned many times that the graphic information of an object is divided among different bit-planes. We mean that every screen pixel is made up of many bits in memory that determine the pixel's color. The depth is determined by how many bits are available for use per pixel. The first bit of all pixels builds the first bit-plane, the second bit builds the second bit-plane, etc.

All bit-planes are stored one after the other in memory. This gives you a special advantage, particularly when using objects, because you can easily add or delete a bit-plane without affecting your objects. You can also define objects that have less depth than the screen. For example, you could use a screen depth of five and an object with a depth of two. With this depth, your object could only have four colors but these colors can be any four of the screen's 32.

There are two statements that enable you to play with the object color combinations that are possible with a palette of 32 colors. By using the same object from our last program we can demonstrate this.

```
REM 1.7.5.6 OBJECT.PLANES statement
SCREEN 1,320,200,5,1
WINDOW 2,"Planes-Demo",,31,1
FOR i=1 TO 61
  READ g
  p$=p$+MKI$(g)
NEXT i
OBJECT.SHAPE 1,p$
OBJECT.ON
x=50
y=5
FOR i=0 TO 3
 FOR j=i+1 TO 4
 IF j<>i THEN
   FOR k=0 TO 31
     IF (k AND 2^i OR k AND 2^j)=0 THEN
       OBJECT.X 1,x
       OBJECT.Y 1,y
       OBJECT.PLANES 1,2^i+2^j,k
       x=x+20
       IF x>244 THEN
         x=50
         y=y+20
       END IF
     END IF
   NEXT
 END IF
 NEXT j
```

```
NEXT i
LOCATE 22,5
PRINT "80 different color combinations"
WHILE INKEY$=""
WEND
WINDOW CLOSE 2
SCREEN CLOSE 1
DATA 0, 0, 0, 0, 0, 2, 0, 16, 0, 16, 52
DATA 0, 0, -8180,-8082,-8081,-8081,-8177
DATA -1,-1,-1,-1,-1,-1,-1025,-1105,-681
DATA -2049,-1,-4,-2,-1,-1,-1,-1,-1,-1,-1
DATA -1,-16381,-16381,-16381,-16381
DATA -16381,-16381,-4,-2,-1,-1,-1,-1,-1
DATA -1,-1,-1,-1,-1025,-1105,-681,-2049
DATA -1
```

As you can see, there are 80 possible combinations. We will explain why there is only 80 later on. First we want to show you how to achieve these combinations. There are two values in the object structure that make this possible. The first value tells us what bit-plane is being written to. The bit-plane determines the order of the five bits of a pixel and also the color in which it appears. This value is named `PlanePick` and you will find it in our editor under the `Flags` menu.

There are ten possible combinations for a two depth bob, in a screen with a depth of five:

1.	11000	0, 1, 2, 3
2.	10100	0, 1, 4, 5
3.	10010	0, 1, 8, 9
4.	10001	0, 1, 16, 17
5.	01100	0, 2, 4, 5
6.	01010	0, 2, 8, 9
7.	01001	0, 2, 16, 17
8.	00110	0, 4, 8, 9
9.	00101	0, 4, 16, 17
10.	00011	0, 8, 16, 17

After each of the ten combinations we have listed the color registers that the computer uses when you write to other bit-planes. In the number series made up of ones and zeros, a one means that a plane will be written in that bit-plane. The first plane of an object will be the first selected plane of the screen. `PlanePick` is not limited to spreading two planes over five bit-planes. It works just as well with fewer or more planes.

As explained above, our editor determines the selected plane through a series of ones and zeros. When you checked the menus for `PlanePick` you probably got an idea about the second method for changing colors. Directly under `PlanePick` in the menus is the

statement PlaneOnOff. This statement sets a value in the object structure that determines what happens in the unselected screen planes. All planes not used by PlanePick are considered unused. As the name indicates, you use PlaneOnOff to turn planes on and off. You already know about the off status since this is the normal condition for an unused plane. When you switch on a bob in an unused plane a shadow mask is written to that plane. In other words, when a bit-plane is set to on, and the shadow mask contains a matching bit for a pixel, that pixel's color will change.

The construction of PlaneOnOff is the same as PlanePick. Unlike the flags, the value assigned using the editor can be changed later, otherwise our color combination program would be more complicated. We use the OBJECT.PLANES to set both of these values. This is the sequence you pass the values: object number, PlanePick, and PlaneOnOff. Again you cannot simply enter the binary number sequence, you have to calculate the values. The following formula does this:

```
PlanePick=b1+2*b2+4*b3+8*b4+16*b5
```

All you have to do is replace b1 thru b5 with the converted binary sequence. The formula for PlaneOnOff is the same.

PlanePick, and especially PlaneOnOff, open up unexpected methods for animation. From one bob you can create many different figures. Using this method we have created an Olympic torch on the screen. The flickering flame and different colors are created by changing PlanePick and PlaneOnOff.

```
REM 1.7.5.6B Fire
DEF FNplanes=CINT(RND)*4+CINT(RND)*8+CINT(RND)*16
FOR i= 1 TO 274
READ a
a$=a$+MKI$(a)
NEXT i
SCREEN 1,320,200,5,1
WINDOW 2,,,,1
PALETTE 0,0,0,0
PALETTE 4,1,.5,0
PALETTE 8,1,0,0
PALETTE 12,1,.26,0
PALETTE 16,1,.4,0
PALETTE 20,.8,0,0
PALETTE 24,.95,.5,0
PALETTE 28,1,.9,0
LOCATE 2,5
PRINT "The Eternal Flame"
LINE (96,120)-(110,180),26,bf
CIRCLE (103,70),60,27,4.02,5.4
LINE (66,108)-(140,108),27
PAINT (100,110),27
OBJECT.SHAPE 1,a$
```

```
CLOSE 1
OBJECT.X 1,79
OBJECT.Y 1,79
OBJECT.ON 1
fire:
a=FNplanes
IF a=28 THEN a=24
OBJECT.PLANES 1,a
FOR i=0 TO 10:NEXT
OBJECT.PLANES 1,,FNplanes
FOR i=0 TO 10: NEXT
GOTO fire
REM Flame Data
DATA 0,0,0,0,0,2,0,48,0,29
DATA 12,24,0,0,0,0,0,0,0,0
DATA 0,0,0,0
DATA 0,0,0,0,0,0,0,0,0,0
DATA 0,0,0,0,0,0,0,96,0,0
DATA 0,0,0,1152,16384,0,1665,-16384,32,1792
DATA -16384,16,1671,-32768,48,902,0,18,5661,-32768
DATA 26,-30194,128,12,-20869,256,7,-14609,-13824,3
DATA 15999,-15872,0,-257,-17664,0,-1554,23040,0,-7434
DATA -8704,0,-18441,5120,0,27399,18432,0,30230,16384
DATA 0,29704,16384,0,-32752,0,0,0,0,0
DATA 0,0,0,0
DATA 0,0,192,0,0,320,0,0,320,128
DATA 0,480,1920,0,416,7680,256,1888,16128,896
DATA 1664,-1024,896,16353,-9216,992,-16607,-208,976,-20702
DATA -2044,504,32727,26652,404,8013,26728,252,4991,29040
DATA 210,15420,31728,122,-21928,-3088,77,-21544,16128,46
DATA 8360,8896,50,56,960,10,-32176,8384,10,-16384
DATA -32640,6,12,-24320,5,271,8192,2,-31244,-32768
DATA 0,-32657,-26624,0,-32745,12288,0,0,8192,0
DATA 2,0,0,6
DATA 0,0,6,0,0,7,0,0,3,0
DATA 0,9,0,0,11,256,0,10,768,1
DATA 28,3328,1,26,4608,2,62,8704,2,206
DATA 8704,6,460,30208,6,460,28160,15,510,17920
DATA 31,3486,-31744,4,-27235,19968,20,21022,19456,5
DATA 13132,7168,21,-26168,12288,7,-25524,-4096,1,22590
DATA 28672,2,-10628,20480,3,-3279,0,1,29903,-20480
DATA 0,19525,12288,0,11577,0,0,1033,0,0,11777,0
```

Our flame uses only eight different colors that come from registers 0, 4, 8, 12, 16, 20, 24 and 28. These are the only colors changed, all others are free for use.

It is possible for the flame to have multiple combinations of the two planes and the shadow mask. `PlanePick` could select both planes at the same time. It is also possible that only one, or none, of the planes will be drawn. `PlaneOnOff` can also set unused planes on.

All of this is arranged with very little effort. These methods can be applied to many different types of objects. For example, an exploding spaceship, the winking of an eye or lip movement, and much more can be packed into an object.

1.7.6 The alternative: sprites

We have not said much about sprites so far in our discussions of animation. Sprites are controlled in a similar way that bobs are controlled. The same BASIC statements that apply to bobs also apply to sprites.

1.7.6.1 The small difference

To design a sprite with our editor all you have to do is set the sprite flag in the `Flag` menu. When you do this all the other menu items will become ghost selections. You will not be able to use them because they have no effect on sprites. Sprite characteristics are determined by the settings of the `Overlay` and `SaveBack` flags.

The properties of sprites are very specific. A sprite can never be more than 16 pixels wide and can only have three colors. The height is not limited. Sprites can move much faster than bobs.

1.7.6.2 Color and sprites

In contrast with bobs, a sprites' color has no direct connection to the screen depth. Sprites carry their own color information which is not stored in any registers (there are no more than 35 possible colors). The colors carried by a sprite can affect the screen colors that are covered by it. When a sprite is on top of or behind a pixel, the color register for that pixel will cause a color change.

You cannot choose which registers a sprite stores its colors. However, you can choose which colors the sprite uses. When you have the sprite flag set you can select "`Spr.Color`" from the menu to pick three colors for the sprite. The values are assigned the same way as with PALETTE.

2. The Amiga operating system

So far our programs have worked well using only the everyday BASIC statements. However, at this point BASIC alone is not enough. Our next series of projects require capabilities that BASIC cannot provide. These capabilities include a graphic hardcopy routine, new character sets, 1024x1024 pixel superbitmap graphics, and much more.

With most computers you would have to write machine language programs in order to perform additional functions and statements. This is not required with the Amiga because the solutions for our next projects are resident in the Amiga operating system and the Amiga operating system *libraries*. Each library consists of hundreds of small machine language routines. These routines are organized by functions and can handle just about any operation. So, with the Amiga, you do not have to create new commands, you only have to learn how to use the libraries.

The procedure for accessing libraries is built into AmigaBASIC, the statements "LIBRARY" and "DECLARE FUNCTION (...) LIBRARY" are used to access the libraries. In order to access these libraries we use a .bmap file. For every system library, you create a matching .bmap file which contains the names of all the machine language library routines and other necessary parameters. By using the ConvertFD program on the Extras disk, you can quickly create all the required .bmap files. See the aboutBmaps program on your Extras disk for more information on creating the .bmap files.

Before you read any further, you should generate the following ".bmap" files:

.bmap files
graphics.bmap
exec.bmap
layers.bmap
intuition.bmap
diskfont.bmap
dos.bmap

You should copy these files to your LIBS directory on the Workbench disk. You can also use these files by having them and your program files in one directory on the same disk.

Now that we have finished the groundwork, we can move on to programming. The LIBRARY and DECLARE FUNCTION (...) LIBRARY statements are our tools. Before you can use one (or more) of the system libraries, you have to open them. We use the LIBRARY statement to do this:

```
LIBRARY "graphics.library"
```

When AmigaBASIC receives this command, BASIC looks for the definition file graphics .bmap. If this file is not found in the current directory, BASIC will look for it on the Workbench disk in the LIBS directory. When BASIC doesn't find the .bmap file there, you will get a "file not found" error. As you can see, in order to run your program, it is important that the .bmap file is available.

When BASIC finds the .bmap file, it opens the appropriate system library. From this point on, BASIC will compare the program function calls with the graphics.bmap function list. The requested routines are executed from the library when they are called.

The next program is a small demonstration of how this works. The function we want to test is in the graphic.library and is called *text*. This routine prints a text string of any length on the screen. There are three required parameters:

1.) the address of the RastPort (to be discussed shortly),
2.) the address of the text,
3.) the length of the text.

```
REM 2A Hello World
LIBRARY "graphics.library"
a$="Hello World!"
length%=LEN(a$)
rastport&=WINDOW(8)
CALL Text(rastport&,SADD(a$),length%)
LIBRARY CLOSE
```

Enter this program very carefully because system libraries are very sensitive to errors. When the program starts, it will look for the graphics.bmap file. In a few moments the message, "Hello World!" will appear in the upper left hand corner of the screen. The statement LIBRARY CLOSE closes the library, but it is not required because BASIC will automatically close any open libraries any time you use RUN or NEW.

We like to explain the statement SADD(a$) and the variable rastport& used in the above program. The statement SADD passes the memory address of the text string a$. The Text routine then uses this address to read the text. Rastport indicates which drawing plane

to use. We will explain this in more detail later. What you need to understand now is that the RastPort address tells the computer the window in which to display the text. The actual window address is in the variable WINDOW(8).

The above program only demonstrated how to use a library. You could have easily accomplished the same thing with a normal PRINT statement. The next program has a SUB routine named "P" that uses the Text routine and is simply a faster replacement for PRINT. To call it:

```
P "text",mode%
        mode%       0 = PRINT "text";
                    1 = PRINT "text"
```

```
'########################################
'#
'# Program:  Quick Print
'# Date:     04/13/87
'# Author:   tob
'# Version:  1.0
'#
'########################################
PRINT "Searching for .bmap file..."
'GRAPHICS-Library
'Text()
LIBRARY "graphics.library"
demo:    demo$=STRING$(80,"*")
         CLS
        '* Quick Print
         PRINT "QUICK PRINT:"
         FOR loop1%=0 TO 10
           P demo$,1
         NEXT loop1%
        '* normal PRINT
         PRINT "AmigaBASIC's normal PRINT:"
         FOR loop2%=0 TO 10
           PRINT demo$
         NEXT loop2%
         LIBRARY CLOSE
SUB P(was$,mode%) STATIC
      CALL Text(WINDOW(8),SADD(was$),LEN(was$))
      IF mode%=1 THEN PRINT
END SUB
```

This new routine is three times as fast as PRINT. The text appears immediately, not line by line. If you add an LPRINT to the SUB P routine, much more is possible. You could PRINT and LPRINT all text without using an LPRINT after each PRINT in your program.

We are finished with our first useful application of the system libraries. Hopefully, you are now curious about the many other possibilities.

Our introduction is not quite complete because the DECLARE FUNCTION (...) LIBRARY statement was not included in our demonstration program. The Text routine used a CALL statement since the libraries performed the required actions without having to return any results to BASIC; therefore a DECLARE statement was not needed. Functions that have to return a result to BASIC must use the DECLARE statement. For example, one of these functions from the graphic library is ReadPixel. ReadPixel requires several parameters: the address of the RastPort (a pointer to the desired window), and a X and Y coordinate. The value that is returned to the program by ReadPixel is the color of the specified pixel. To use this function, you must use the DECLARE assignment:

```
REM 2C DECLARE
DECLARE FUNCTION ReadPixel% LIBRARY
LIBRARY "graphics.library"
x%=320
y%=125
rastport&=WINDOW(8)
colour%=ReadPixel%(rastport&,x%,y%)
PRINT "Found Color Number";colour%
LIBRARY CLOSE
```

ReadPixel is declared as a function in the first program line because, when called, it returns a value to the program. The "%" character after ReadPixel in the function definition, declares the value returned by ReadPixel to be of type integer. From this point on, the routine is called using the name ReadPixel%.

This completes our explanation of the basic principles involved in using libraries. You know how to open a library and how to access a routine in it. The following pages provide details on what routines the various libraries have and what parameters are required. AmigaBASIC can open and access up to five libraries at the same time. You should always remember that the routines in these libraries are machine language programs. Even though they are a bit more difficult to use, these routines are much faster than pure BASIC. Also remember that these routines do not check for incorrect or invalid parameters, so one small mistake, an invalid parameter or a wrong call, could cause the system to crash. You would then have to reboot from the Workbench and all programs in memory are lost.

Enter all your programs carefully and save them to disk before you test them.

The following program shows you how easy it is to crash the system.

Note: This program has errors and causes a system crash - all programs in memory will be lost!

```
REM 2D_Crashomatic
REM **********************************
REM ATTENTION!!!!
REM THIS PROGRAM WILL CRASH THE AMIGA!
REM **********************************

LIBRARY "graphics.library"
a$="Hello World!"
length%=LEN(a$)
rasport&=WINDOW(8)

'* Above is the error (rasport& instead
'* of rastport& so rastport& = 0).

CALL Text(rastport&,SADD(a$),length%)
LIBRARY CLOSE
```

If you run this program you will receive the following display or the system will lock up completely:

```
Software failure. Click left mouse button to continue.
  Guru Meditation #00000004.00xxxxx
```

Follow the instructions, or perform a warm reset by pressing both <A> Amiga keys and the <Ctrl> key at the same time.

3. Intuition - the user interface

We begin our trip through the graphic world of the Amiga with a system component named *Intuition*, which refers to a library within the operating system (see Chapter 2). Intuition is directly related to the graphic libraries' capabilities and is responsible for windows, screens, requesters, alerts (Guru Meditations, for example) and much more.

3.1 Intuition windows

Unless you are running another operating system, Intuition will manage all windows. This also applies to the standard windows of the BASIC interpreter, LIST and BASIC. A data block used for Intuition windows consists of 124 bytes and contains all data specific to a window. The starting address of the BASIC output window data is always stored in the variable `WINDOW(7)`. You can obtain the start address like this:

```
windo&=WINDOW(8)
```

The data block is stored in the following format:

Data Structure Window/Intuition/124 Bytes

Offset	Type	Definition
+ 000	Long	Pointer to next window
+ 004	Word	X coordinate of upper left corner
+ 006	Word	Y coordinate of upper edge
+ 008	Word	Width of window
+ 010	Word	Height of window
+ 012	Word	Y coordinate of mouse, rel. to window
+ 014	Word	X coordinate of mouse, rel. to window
+ 016	Word	minimum width of window
+ 018	Word	minimum height of window
+ 020	Word	maximum width of window
+ 022	Word	maximum height of window
+ 024	Long	Window modes
		Bit 0: 1=Sizing gadget available
		Bit 1: 1=Dragbar gadget available
		Bit 2: 1=Fore/background gadgets available

Offset	Type	Definition
		Bit 3: 1=Close gadget available
		Bit 4: 1=Sizing gadget is right
		Bit 5: 1=Sizing gadget is bottom
		Bit 6: 1=Simple refresh
		Bit 7: 1=Superbitmap
		Bit 8: 1=Backdrop window
		Bit 9: 1=Report mouse
		Bit 10: 1=GimmeZeroZero
		Bit 11: 1=Borderless
		Bit 12: 1=Activate
		Bit 13: 1=This window is active
		Bit 14: 1=This window is in request mode
		Bit 15: 1=Active window with active menu
+ 028	Long	Pointer to menu header
+ 032	Long	Pointer to title text for this window
+ 036	Long	Pointer to first active requester
+ 040	Long	Pointer to double click requester
+ 044	Word	Number of the window block request
+ 046	Long	Pointer to the screen, of this window
+ 050	Long	Pointer to the RastPort of this window
+ 054	Byte	Left border
+ 055	Byte	Top border
+ 056	Byte	Right border
+ 057	Byte	Bottom border
+ 058	Long	Pointer to border RastPort
+ 062	Long	Pointer to the first gadget
+ 066	Long	Pointer to previous window (father)
+ 070	Long	Pointer to next window (child)
+ 074	Long	Pointer to sprite data for pointer
+ 078	Byte	Height of sprite pointer
+ 079	Byte	Width of sprite pointer
+ 080	Byte	X offset of pointer
+ 081	Byte	Y offset of pointer
+ 082	Long	IDCMP flags
+ 086	Long	User message port
+ 090	Long	Window message port
+ 094	Long	IntuiMessage message key
+ 098	Byte	Detailpen
+ 099	Byte	Blockpen
+ 100	Long	Pointer to menu checkmarks
+ 104	Long	Pointer to screen title text
+ 108	Word	GZZ-mouseX
+ 110	Word	GZZ-mouseY

Offset	Type	Definition
+ 112	Word	GZZ-width
+ 114	Word	GZZ-height
+ 116	Long	Pointer to external data
+ 120	Long	Pointer to user data

Every window has a data block like the one in the above format. To access the different data fields in the data block, first select the desired output window using the `WINDOW OUTPUT` statement. After this command, the data block address will be in the variable `WINDOW(7)`. You add the offset value of the desired data field to this address. There are three types of data fields, byte, word and long. A byte field which consists of exactly one byte, is read with `PEEK` and is changed with `POKE`. A word field is two bytes wide and is read and written with `PEEKW` and `POKEW`. A long field is four bytes wide and is read and written with `PEEKL` and `POKEL`. We will present several examples of each.

3.2 The window data structure

Now you know the method used to access the data structure of your output window. The following is a detailed explanation of each data field to explain what you can do with them.

Offset 0: Pointer to next window

Intuition manages all windows in a type of chain. This offset contains the starting address of the next window data block. You can find the data block of the next and previous window from your current data block. This particular pointer is not important because the chain pointers come later at offsets 66 and 70.

Offset 4 and 6: Position of window in upper left hand corner

These two word fields contain the X and Y coordinates of the upper left hand corner of their window. These coordinates are relative to the upper left hand corner of the screen that contains the window. The lines below provide you with the coordinates:

```
windo&=WINDOW(7)
x%=PEEKW(windo&+4)
y%=PEEKW(windo&+6)
PRINT "Upper left Window Corner is at"
PRINT "Coordinates (";x%;",";y%;")"
END
```

Because the BASIC output window sits in the upper left hand corner of the Workbench screen you would receive coordinates of (0,0). The moment you drag this window to a different position, the coordinates are changed. Later we will use these values to access various Intuition routines.

Offset 8 and 10: Window dimensions

Since we can change the window size with the sizing gadget in the lower right hand corner of the window, the program doesn't always know the current window size. The small program below provides the current window size:

```
windo&=WINDOW(7)
windoWidth%=PEEKW(windo&+8)
windoHeight%=PEEKW(windo&+10)
PRINT "At the moment your Window is ";windoWidth%
PRINT "Pixels wide and ";windoHeight%;" Pixels high."
END
```

Offset 12 and 14: Mouse coordinates

These two word fields contain the actual mouse coordinates relative to the upper left hand corner of the window. The first field contains the Y coordinate and the second field contains the X coordinate.

Using these two values you can easily create a small drawing program. The following lines demonstrate this:

```
'########################################
'#
'# Section: 3.2
'# Program: Mouse Draw I
'# Date:    04/05/87
'# Author:  tob
'# Version: 1.0
'#
'########################################
init:       windo&=WINDOW(7)
           'Address of Window Data Structure
loop:      'wait for key press
            PRINT "Press any key to exit"
            WHILE INKEY$=""
              mouse.y% = PEEKW(windo&+12)
              mouse.x% = PEEKW(windo&+14)
              PSET (mouse.x%,mouse.y%)
            WEND
```

Two things are immediately visible. First, the line drawn starts several pixels below the mouse pointer and second, only a dotted line is being drawn. The mouse position fields contain the mouse coordinates relative to the upper left hand corner of the window. However, `PSET` draws in a plane that uses an offset relative to the upper left hand corner of the drawing plane and not the window (as long as the window is a GimmeZeroZero window; we will have more on this later). To correct the window/plane problem, subtract 11 from the Y value and 4 from the X value. The second problem is caused by BASIC's slow speed. The mouse moves faster than BASIC can draw. Test this by moving the mouse very slowly across the screen. You should see a complete line from one point to another. The next program contains both changes:

```
'########################################
'#
'# Section: 3.2B
```

```
'# Program: Mouse Draw II
'# Date:    04/05/87
'# Author:  tob
'# Version: 1.0
'#
'########################################
init:       windo&=WINDOW(7)
           '* Address of Window Data Structure
            mouse.y% = PEEKW(windo&+12)-11
            mouse.x% = PEEKW(windo&+14)-4
loop:      '* wait for key press
            PRINT "Press any key to exit"
            WHILE INKEY$=""
              oldmouse.y% = mouse.y%
              oldmouse.x% = mouse.x%
              mouse.y%    = PEEKW(windo&+12)-11
              mouse.x%    = PEEKW(windo&+14)-4
              LINE (oldmouse.x%,oldmouse.y%)-
(mouse.x%,mouse.y%)
            WEND
```

When you speed around the screen with this program, you will discover that individual "peaks" appear. This happens when the mouse coordinates are calculated but the Y value has not been changed.

Offset 16, 18, 20 and 22: Window limits

Some windows can be made larger and smaller using the mouse. Every window has its own maximum and minimum size. A window can be any size within these limits in the window data structure.

```
windo&=WINDOW(7)
min.x%=PEEKW(windo&+16)
min.y%=PEEKW(windo&+18)
max.x%=PEEKW(windo&+20)
max.y%=PEEKW(windo&+22)
PRINT "Minimum Size: X=";min.x%;
PRINT "Y=";min.y%
PRINT "Maximum Size: X=";max.x%;
PRINT "Y=";max.y%
END
```

In this example we read the limit values. You can easily set your own window limits by using POKEW to the required addresses:

```
windo&=WINDOW(7)
min.x%=5
min.y%=7
max.x%=640
max.y%=200
```

```
POKEW windo&+16,min.x%
POKEW windo&+18,min.y%
POKEW windo&+20,max.x%
POKEW windo&+22,max.y%
END
```

There are two things you must always be sure of:

a) The minimum size has to be smaller than the maximum size.

b) The maximum size cannot be smaller than the current window you are changing. (Dimensions are available from offset 8 and 10).

Offset 24: Window modes

Intuition has different window types and each window can have various gadgets assigned to them:

a) Sizing gadget
b) Dragbar
c) Fore/background gadget
d) Close gadget

You can also set the refresh mode for the window. This mode determines how a window is restored once another window covers it. *Simple refresh* leaves the restoring up to you. Normally this means that if another window covers your window, the covered part is lost. *Smart refresh* uses Intuition to save the covered portion of window and restores it after the window is uncovered. This method requires more time and can use much memory, depending on how many windows are active. The *Superbit* method keeps a copy of the entire window contents in RAM. Although this expends a large amount of memory, it allows a window to show a piece of a much larger graphic.

Besides the basic attributes of a window, there are also special windows. A *backdrop window* is always behind all the other windows and cannot be put in the front. For example, it works as the background or graphic plane. The visible workbench screen is nothing more than a backdrop window that covers the screen. A *GimmeZeroZero window* has two parts, a border and a drawing plane. This method allows easy drawing because it is impossible to accidentally draw over the border. The BASIC window is an example of a GimmeZeroZero window. The *borderless window* does not have a border frame. An example of this type is the backdrop workbench window because it does not have a border frame and it merges with the background.

To access these modes use the bit pattern from the table in Section 3.1.

Note: Changes to this field only take effect after Intuition refreshes the window (for example, when you drag it to move it).

Offset 28: The menu header

A special property of all Intuition windows is their ability to have menus. When you press the right mouse button, the menus appear in the top window bar.

This field contains the pointer to the Intuition menu system for this window.

Offset 32: Title text for the window

Every window has its own name. This offset contains the starting address for the title text. The following lines demonstrate an easy way to change this:

```
window.name$="Hello World"+CHR$(0)
POKEL WINDOW(7)+32,SADD(window.name$)
```

As soon as you perform an action with the window, such as dragging (causing Intuition to refresh the window), the new name appears. Later we will show you a method for changing a window name that will give you instant results.

The `CHR$(0)` is a null byte added at the end of the text that tells Intuition where the text ends.

Offset 36, 40 and 44: Requester handling

Intuition uses this data field to remember how many (and what kind) of requesters are blocked by this window. Basic programmers can ignore this field for now.

Offset 46: Pointer to screen

Windows with complete freedom of movement do not exist. Every window appears in a screen. The Workbench screen is the first screen and the other screens can be overlaid by using the `SCREEN` statement. This offset contains the address of the Intuition screen data structure. In the same way that each window has its own data structure, each screen has its own data structure. Screen data is arranged somewhat differently. We will discuss that in more detail later.

Offset 50: Pointer to RastPort of window

The RastPort is nothing more than another data structure. It could be defined as an intersection. From the RastPort there are paths to

window, screens and even the most simple graphic components like bit-maps and layers. We will discuss all of these in detail later.

Offset 54, 55, 56 and 57: Window borders

This four byte field contains the dimensions in pixels of the window border. When we were working with the mouse coordinates (offset 12, 14) we subtracted the values 11 and 4. The same values are located here; they are the height of the top border and the width of the left border.

You can calculate the drawing coordinates in a GimmeZeroZero window by adding these values. This determines your relative position to the top left hand corner of the window.

When working with other window types you have to be careful not to draw over the window border. The following rules can help you avoid this:

a) Your X coordinate must be greater than the left window border and smaller than the width of the window (offset 8) minus the width of the right window border.

b) Follow the same rule for the Y coordinates.

Offset 58: The RastPort border

All GimmeZeroZero windows control two independent drawing planes: the window border and the window contents. This offset contains the address pointer for the RastPort of a GimmeZeroZero window. The following program uses the RastPort border and the graphic function `Text` to display a status line in the window header:

```
'########################################
'#
'# Section: 3.2C
'# Program: Status Line
'# Date:    12/17/89
'# Author:  tob
'# Version: 1.0
'#
'########################################

' This program creates a User Status Line in a
' GimmeZeroZero Window. BorderRastPort is the
' length of x. x is independent of the actual
' window size. Error checking prevents any
' Gadgets from being disturbed.

PRINT "Searching for .bmap file..."
```

```
'GRAPHICS-Library
'Text()
'SetAPen()
'SetDrMd()
'Move()

LIBRARY "graphics.library"

main:       CLS
            Status "STATUS: Demo Window. Please press a
Key!",60
            WHILE INKEY$=""
            WEND
            WHILE yn$<>"y"
              Status "STATUS: Please Enter your Name!",60
              CLS
              LOCATE 1,1
              LINE INPUT "--> ";n$
              Status "Name: "+n$+". Correct (y/n) ?",60
              LOCATE 1,1
              PRINT SPACE$(50)
              LOCATE 1,1
              LINE INPUT "--> ";yn$
            WEND
            Status "Test is over. Good Bye! (ANY KEY!)",0
            CLS
            WHILE INKEY$=""
            WEND

endprog:    WINDOW 1,""
            LIBRARY CLOSE
            END

SUB Status(text$,t.width%) STATIC
      borderRast&    = PEEKL(WINDOW(7)+58)
      IF borderRast& = 0 THEN
        BEEP
        PRINT "This is not a GimmeZeroZero type Window."
        ERROR 255
      END IF

      windoWidth% = PEEKW(WINDOW(7)+8)
      maxChar%    = INT((windoWidth%-86)/8)
      TextLen%    = LEN(text$)
      IF t.width% = 0 THEN t.width%=TextLen%
      IF t.width%<maxChar% THEN maxChar%=t.width%
      IF TextLen%<t.width% THEN
        text$ = text$+SPACE$(t.width%-TextLen%)
      END IF
      CALL SetAPen(borderRast&,1)
      CALL Move(borderRast&,32,7)
      CALL text(borderRast&,SADD(text$),maxChar%)
      CALL SetDrMd(borderRast&,0)
      CALL SetAPen(borderRast&,3)
      CALL Move(borderRast&,31,7)
      CALL text(borderRast&,SADD(text$),maxChar%)
```

```
        CALL SetDrMd(borderRast&,1)
END SUB
```

Offset 62: First gadget

Gadgets are small (or large) "switch elements" that you select with the mouse. Among these are the on/off gadget and sizing gadget. This offset contains the address of the first gadget structure in a chain. BASIC users can disregard this.

Offset 66 and 70: Father and child windows

In our discussion of offset 0 we mentioned that Intuition manages all windows in a data chain. Every window has its own data structure block. In each data structure there is a pointer to the previous (father) window and to the next (child) window. The first window in a chain does not have a father pointer (=0) and the last has no child pointer (=0).

These two fields are very important. Up until now it was possible for us to determine the address of the output window by using the variable `WINDOW(7)`. Now we can use a window to determine where the data for any window is located. The following program demonstrates this:

```
'########################################
'#
'# Section: 3.2
'# Program: Window Finder
'# Date:    04/05/87
'# Author:  tob
'# Version: 1.0
'#
'########################################

init:       windo& = WINDOW(7)

'* Here we search for the end of the Data Chain.
'* The 'parent' field of the first element is zero.

            WHILE found% = 0
              parent.windo& = PEEKL(windo&+66)
              IF parent.windo&=0 THEN
                found% = 1
              ELSE
                windo& = parent.windo&
              END IF
            WEND
            found% = 0

'* windo& contains the address of the Window
'* Data Block of the first Window in the Data
'* chain. Now we read through the chain, until
'* until the 'child' field equals zero.
```

```
        WHILE found%=0
          counter% = counter%+1
          PRINT counter%;
          PRINT ". Window:"
          PRINT "Address of Data Structure: ";windo&

'* Now we provide the name of the window found at
'* Offset +32

          PRINT "Name of Window:                ";
          windo.name& = PEEKL(windo&+32)
          WHILE done% = 0
            gef$ = CHR$(PEEK(windo.name&))
            IF gef$ = CHR$(0) THEN
              done% = 1
              PRINT
            ELSE
              PRINT gef$;
              windo.name& = windo.name&+1
            END IF
          WEND
          PRINT
          done% = 0
          child.windo& = PEEKL(windo&+70)
          IF child.windo& = 0 THEN
            found% = 1
          ELSE
            windo& = child.windo&
          END IF
        WEND
```

Using these methods you have complete control of all the Intuition managed windows. You can write into other windows, change their names, etc. We will present more programs further on that demonstrate this.

Offset 74, 78, 79 and 80: The sprite image

It is possible for every window to have its own mouse pointer. This offset has the address pointer to the data for the new pointer's sprite image, the height and the width (maximum 16 pixels). Changing the values of this offset doesn't accomplish anything. To set up your custom pointer, use `SetPointer` via Intuition. We show an example of how to do this at the end of this chapter.

Offset 82, 86, 90 and 94: IDCMP Flags and Message Ports

IDCMP stands for *Intuition Direct Communications Message Port.* Intuition communicates through this channel with other tasks such as BASIC. This is not important to you as a BASIC programmer since

Intuition receives the messages from BASIC and processes them automatically.

Offset 98 and 99: Window colors

The window determines its colors from these two byte fields where the color registers are stored. You can change the window colors by poking new values here. As before, Intuition does not make the changes until a window refresh is prompted (any window action such as dragging or sizing).

Offset 100: The check mark image

This offset contains the address of a small graphic image. You have probably seen a menu option that sets a specific menu item on and marks it with a check mark. The default value of this offset is zero for the standard check mark image.

Offset 104: The screen title

A screen, that contains your window, can have different names. The screen name depends on which window is selected (=active). Each window can give the screen a different name. This offset holds the address pointer of the name text.

Offset 108, 110, 112 and 114: GimmeZeroZero parameters

These fields are used only for a GimmeZeroZero window. GZZ-mouseX and GZZ-mouseY contain relative values to offset fields 12 and 14. They specify the coordinates of the mouse pointer relative to the drawing plane, not to the window. The zero coordinate is the upper left hand corner of the window contents, not the upper left hand corner of the window border.

GZZ-width and GZZ-height contain values relative to offset fields 8 and 10. The width and height of the window contents without the border are stored here.

Offset 116 and 120: Optional pointers

These two pointers are used to link data blocks of different types with the standard structure. The first pointer is reserved for Intuition, the second is available for the user.

3.3 Functions of the Intuition library

Now that we understand how Intuition manages windows we can look at the Intuition library. The following routines of the Intuition library are responsible for windows:

```
SetPointer()
ClearPointer()
MoveWindow()
SizeWindow()
WindowLimits()
WindowToBack()
WindowToFront()
```

3.3.1 A personalized mouse pointer

The `SetPointer` function makes it possible for you to create your own personalized mouse pointer for your window. As soon as your window is active the normal pointer will be replaced by yours.

This function requires six parameters:

```
SetPointer(window,image,height,width,xOff,yOff)
```

`window:`	Address of window data structure of that window
`image:`	Address of sprite image block
`height:`	Height of sprite
`width:`	Width of sprite (max. 16)
`xOff:`	Marks the "hot spot"
`yOff:`	Marks the "hot spot"

The following is an example program which shows how to change the mouse pointer. The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 3.3.1¶
'# Program: SetPointer-ClearPointer¶
'# Date:    04/05/87¶
'# Author:  tob¶
```

```
'# Version: 1.0¶
'#¶
'########################################¶
¶
' The Amiga standard mouse pointer is¶
' replaced by a user-defined pointer.¶
¶
PRINT "Searching for .bmap file..."¶
¶
'INTUITION-Library¶
'SetPointer()¶
'ClearPointer()¶
LIBRARY "intuition.library"¶
¶
init:       image$=""¶
            sprite.height%  =  14¶
            sprite.width%   =  16¶
            sprite.xOff%    = -7¶
            sprite.yOff%    = -6¶
¶
image:     '* Read Sprite Image Data¶
            RESTORE sprite.data¶
            FOR loop%=0 TO 31¶
              READ info&¶
              hi%    = INT(info&/256)¶
              lo%    = info&-(256*hi%)¶
              image$ = image$+CHR$(hi%)+CHR$(lo%)¶
            NEXT loop%¶
¶
setpoint:  '* Build New Image¶
           CALL SetPointer(WINDOW(7),SADD(image$),
sprite.height%,sprite.width%,sprite.xOff%,sprite.yOff%)¶
¶
mainDemo:  '* Demonstration of the new pointer¶
           CLS¶
           PRINT "Any key to exit"¶
           PRINT "Left Mouse Button to draw"¶
¶
          '* Draw with pattern¶
           DIM area.pat%(3)¶
           area.pat%(0) = &H1111¶
           area.pat%(1) = &H2222¶
           area.pat%(2) = &H4444¶
           area.pat%(3) = &H8888¶
           PATTERN ,area.pat%¶
           COLOR 2,3¶
¶
           WHILE INKEY$=""¶
             state%=MOUSE(0)¶
             IF state%<0 THEN¶
               mouseOldX% = mouseX%¶
               mouseOldY% = mouseY%¶
               mouseX%    = MOUSE(1)¶
               mouseY%    = MOUSE(2)¶
               IF lplot% = 0 THEN¶
                 lplot% = 1¶
```

```
                    PSET (mouseX%,mouseY%)¶
                  ELSE¶
                    LINE (mouseOldX%,mouseOldY%)-(mouseX%,
mouseY%),1,bf¶
                  END IF¶
                ELSE¶
                  lplot% = 0¶
                END IF¶
              WEND¶
¶
              COLOR 1,0¶
¶
endprog:      '* End demo and restore old pointerer¶
              CALL ClearPointer(WINDOW(7))¶
              LIBRARY CLOSE¶
              END¶
¶
sprite.data:      '* Sprite Data¶
              DATA 0,0¶
              DATA 256,256¶
              DATA 256,256¶
              DATA 256,256¶
              DATA 896,0¶
              DATA 3168,0¶
              DATA 12312,0¶
              DATA 256,49414¶
              DATA 256,49414¶
              DATA 12312,0¶
              DATA 3168,0¶
              DATA 896,0¶
              DATA 256,256¶
              DATA 256,256¶
              DATA 256,256¶
              DATA 0,0¶
```

Program Description:

Our new mouse pointer should be 16 pixels wide and 14 pixels high. The "hot spot", the sensitive pixel of the pointer, is seven pixels to the right and six pixels below the upper left hand corner of the sprite.

The program routine `image` defines the shape of the new pointer. The data for the shape is stored in the section named `sprite.data`. Each row of a sprite can be a maximum of 16 pixels wide. The data block is composed of two 16 bit values per sprite row. Since our sample sprite is 14 rows high there have to be 14x2=28 data elements (plus 2 zeros at the beginning and end to switch off DMA). For every possible pixel of the sprite there are two bits. If neither bit is set the pixel will be transparent. The remaining three combinations determine which of the three possible colors the pixel will be.

The sprite data is stored in the string variable `image$`. To do this the 16 bit values are first converted to two 8 bit values (lo and hi byte).

Finally we call `SetPointer` to activate the new pointer.

At the end of the program we use `ClearPointer` to restore the normal pointer.

3.3.2 Moving windows made easy

Instead of using the mouse to drag windows around the screen, you can program Intuition to do the same thing. You can use the Intuition routine "MoveWindow" which requires three parameters:

```
MoveWindow(window,deltaX,deltaY)
```

window:	Address of window data structure
deltaX:	Number of pixels, to move the window to the right (negative = left)
deltaY:	Same action in vertical direction

These routines do not check your parameters for valid coordinates. If your delta values set coordinates outside the screen, Intuition will try to move your window off the monitor. Of course this will not work. To prevent this from happening we have added a small error check routine to the next program. This routine will check your input with the data in the window data structure (Section 2.2).

This program is a SUB named `Move`.

```
'########################################
'#
'# Section: 3.3.2
'# Program: Window Move
'# Date:    04/10/87
'# Author:  tob
'# Version: 1.0
'#
'########################################

'Intuition can move selected Windows under
'program control. This program demonstrates
'the WindowMove() command (including error-
'checking).

PRINT "Searching for .bmap file..."

'INTUITION-Library
'MoveWindow()

LIBRARY "intuition.library"
```

```
demo:       CLS
            WINDOW 2,"Test Window",(10,10)-(400,100),16
            WINDOW OUTPUT 2

            PRINT "Original Position. Please press a key."
            WHILE INKEY$="":WEND

            Move 10,20          '10 right, 20 down
            PRINT "New Position! Press a key."
            WHILE INKEY$="":WEND

            Move -10,-20        '10 left, 20 up
            PRINT "Returned!"

            FOR t=1 TO 3000:NEXT t

            Move 10000,10000    'Window
            'Nothing happened, thanks to error check

            WINDOW CLOSE 2
            LIBRARY CLOSE

SUB Move(x%,y%) STATIC
      win&            = WINDOW(7)
      screen.width%   = 640
      screen.height%  = 200      'PAL systems can use 256
here
      windo.x%        = PEEKW(win&+4)
      windo.y%        = PEEKW(win&+6)
      windo.width%    = PEEKW(win&+8)
      windo.height%   = PEEKW(win&+10)
      min.x%          = windo.x%*(-1)
      min.y%          = windo.y%*(-1)
      max.x%          = screen.width%-windo.width%-
windo.x%
      max.y%          = screen.height%-windo.height%-
windo.y%
      IF x%<min.x% OR x%>max.x% THEN x%=0
      IF y%<min.y% OR y%>max.y% THEN y%=0
      CALL MoveWindow(win&,x%,y%)
END SUB
```

3.3.3 Setting window limits

All windows that can be sized have a minimum and maximum size. The Intuition routine `WindowLimits` sets the limits according to the input values. The data fields at offset 16 (see Section 2.2) are directly manipulated by `WindowLimits`. This routine also checks for valid arguments. When all parameters are okay, `WindowLimits` returns a TRUE (=1), if not, it returns a FALSE (=0).

WindowLimits requires five arguments and returns a result:

```
result%=WindowLimits%(window,minX,minY,maxX,maxY)
```

result%:	1 = all OK 0 = Minimum size greater than maximum size, etc.
minX,minY:	Minimum size of window
maxX,maxY:	Maximum size of window

A short example:

```
REM 3.3.3 Window Limits
DECLARE FUNCTION WindowLimits% LIBRARY
LIBRARY "intuition.library"
minX%=5
minY%=5
maxX%=640
maxY%=200
res%=WindowLimits%(WINDOW(7),minX%,minY%,maxX%,maxY%)

IF res%=0 THEN
  PRINT "Something is incorrect..."
END IF

LIBRARY CLOSE
END
```

3.3.4 Sizing windows

The Intuition function SizeWindow is used to shrink or enlarge a window. SizeWindow requires three arguments:

```
SizeWindow(window,deltaX,deltaY)
```

window:	Address of window data structure
deltaX:	Number of pixels to enlarge the window horizontally (negative = shrink)
deltaY:	Same action, only vertically

This routine does not check for incorrect parameters. If your delta values make the window smaller than is possible or bigger than the screen, you will encounter a system crash. To prevent this, the next program has a routine that recognizes false values and makes them safe. This SUB is named Size:

```
'########################################
'#
```

```
'# Section: 3.3.4
'# Program: Window Limits
'# Date:    04/10/87
'# Author:  tob
'# Version: 1.0
'#
'########################################

'Demonstrates setting a Windows minimum
'and maximum limits for size.

PRINT "Searching for .bmap file..."

'INTUITION-Library
'WindowLimits

DECLARE FUNCTION WindowLimits% LIBRARY

LIBRARY "intuition.library"

demo:       CLS
            WINDOW 2,"Test Window",(10,10)-(400,100),16
            WINDOW OUTPUT 2

           '* Set Window Limits
            r%=WindowLimits%(WINDOW(7),0,0,600,200)
            IF r%=0 THEN ERROR 255

            PRINT "Original Size - Please press a key..."
            WHILE INKEY$="":WEND

            Size 60,40      '60 right, 40 down
            PRINT "New Size - press a key..."
            WHILE INKEY$="":WEND

            Size -60,-40    '60 left, 40 up
            PRINT "Restored!"

           '* wait
            FOR t=1 TO 3000:NEXT t

           '* invalid arguements caught here
            Size 10000,10000    'Error
            'nothing happened thanks to error check

            WINDOW CLOSE 2
            LIBRARY CLOSE

SUB Size(x%,y%) STATIC
      win&           = WINDOW(7)
      windo.width%   = PEEKW(win&+8)
      windo.height%  = PEEKW(win&+10)
      windo.minX%    = PEEKW(win&+16)
      windo.minY%    = PEEKW(win&+18)
      windo.maxX%    = PEEKW(win&+20)
      windo.maxY%    = PEEKW(win&+22)
```

```
        min.x%          = windo.minX%-windo.width%
        min.y%          = windo.minY%-windo.height%
        max.x%          = windo.maxX%-windo.width%
        max.y%          = windo.maxY%-windo.height%
        IF x%<min.x% OR x%>max.x% THEN x%=0
        IF y%<min.y% OR y%>max.y% THEN y%=0
        CALL SizeWindow(win&,x%,y%)
END SUB
```

3.3.5 WindowToFront and WindowToBack

A window can be moved to the background or foreground in relation to other windows. These operations are normally performed with the mouse. However, there are two Intuition functions that can perform the same operation from within a program, `WindowToFront` and `WindowToBack`.

Here is a small demonstration:

```
LIBRARY "intuition.library"

FOR loop%=1 TO 10
  CALL WindowToBack(WINDOW(7))
  PRINT "Behind!"
  FOR t=1 TO 2000:NEXT t
  CALL WindowToFront(WINDOW(7))
  PRINT "In Front!"
  FOR t=1 TO 2000:NEXT t
NEXT loop%
END
```

3.4 The Intuition screen

Intuition also manages screens. An Intuition screen data structure is similar to those of Intuition windows. The pointer to the screen data is located at offset 46 in the window data structure. To obtain the base address of your output window, use the following:

```
screen&=PEEKL(WINDOW(7)+46)
```

To calculate the address of an individual data field you add the offset to the base address. The data structure is as follows:

Data structureScreen/Intuition/342 Bytes

Offset	Type	Description
+ 000	Long	Pointer to next screen
+ 004	Long	Pointer to first window in this screen
+ 008	Word	X coordinate of upper left corner
+ 010	Word	Y coordinate of upper left corner
+ 012	Word	Width of screen
+ 014	Word	Height of screen
+ 016	Word	Y coordinate of mouse pointer
+ 018	Word	X coordinate of mouse pointer
+ 020	Word	Flags
		Bit 0: 1=Workbench screen
		Bit 0-3: 1=Custom screen
		Bit 4: 1=Showtitle
		Bit 5: 1=Screen beeps now
		Bit 6: 1=Custom bit-map
+ 022	Long	Pointer to screen name text
+ 026	Long	Pointer to standard title text
+ 030	Byte	TitleBar height
+ 031	Byte	Vertical limit of TitleBar
+ 032	Byte	Horizontal limit of TitleBar
+ 033	Byte	Vertical limit of menus
+ 034	Byte	Horizontal limit of menus
+ 035	Byte	Top window border
+ 036	Byte	Left window border
+ 037	Byte	Right window border
+ 038	Byte	Bottom window border
+ 039	Byte	unused
+ 040	Long	Pointer to standard font TextAttr
+ 044	---	ViewPort of screen

Offset	Type	Description
+ 084	----	RastPort of screen
+ 184	----	Bit-map of screen
+ 224	----	LayerInfo of screen
+ 326	Long	Pointer to first screen gadget
+ 330	Byte	Detailpen
+ 331	Byte	Blockpen
+ 332	Word	Backup register for `Beep()`, stores `Col0`
+ 334	Long	Pointer to external data
+ 338	Long	Pointer to user data

You probably noticed that the window and screen data structures are quite similar, even though some fields are new. Again, we will explain each field individually.

3.4.1 Screen structure

Offset 0: Next screen

Screens are also organized by Intuition in the form of a data chain. When there are other screens in a chain with your screen, the address of the next screen's data block is at this offset.

Offset 4: First window

Remember the data chain used by windows: two fields, the father and child, provide the addresses of the previous and next windows. By using these you can move back and forth through the chain from one window to another. However, this method has a small flaw. In order to go through the entire chain, you have to find the beginning of the chain first. The active window you access is not necessarily the first window in the chain.

If you are interested only in a window in a specific screen, there is an easier method you can use. This field contains the address of the first window data block. The address of the next window structure is in offset +0 of the window data structure.

```
windo&=WINDOW(7)
scr&=PEEKL(windo&+46)
searcher&=scr&+4
WHILE flag%=0
  searcher&=PEEKL(searcher&)
  IF searcher&=0 THEN
    flag%=1
  ELSE
```

```
        counter%=counter%+1
        PRINT "Window";counter%;
        PRINT "Data Block Address:";
        PRINT searcher&
      END IF
    WEND
    END
```

Note: Although this method is easier to program, it only accesses those windows in the output window of the current screen.

Offset 8, 10, 12 and 14: Screen dimensions

Like the window data structure, these offsets contain the coordinates of the upper left hand corner of the screen and the height and width of the screen. The corner coordinate is relative to the top corner of the display. The current versions of the Amiga (500-2000) do not allow horizontal shifting of the screen. Offset field +8 is available for future compatibility.

Offset 16 and 18: The Mouse Coordinates

Here you find the Y and X coordinates of the mouse pointer relative to the upper left hand corner of the screen. During vertical screen movements the Y value can vary a bit.

Offset 20: Flags

The bit descriptions are self-explanatory. Show title means that the screen's title text is visible. A custom bit-map is suitable when you are generating a new screen that has its own drawing plane.

Offset 22 and 26: The name of the screen

This offset contains the address of either the name string of this screen or a standard text string which is taken as default from a window when a name string is not specified.

Offset 30 - 39: Default parameters

These byte fields contain various standard parameters, such as the dimensions of the title bar, etc. All windows in this screen default to these dimensions automatically.

Offset 40: The standard character set

Whenever a window is opened in your screen, it has a standard character set (a predetermined font). The address of this font is stored here.

We will be discussing character sets in more detail later.

Offset 44: The ViewPort

This is another new term we will be discussing in more detail later. In relation to the data structure of the screen, the *ViewPort* is not a pointer. At offset 44, it is the actual ViewPort of the screen. ViewPort, which is a small data structure 40 bytes long, is an interface to the Amiga graphic hardware (the *Copper* graphic coprocessor).

Offset 84: The RastPort

This offset is not a pointer either, but the actual RastPort. You received an introduction to the RastPort in the window data structure. Since both screens and windows are drawing planes, the screen also has a RastPort.

Offset 184: The Bit-map

This is a new data structure which is 40 bytes long. Bit-map is the interface between the screen and the memory it occupies, called "bit-planes".

Offset 224: The LayerInfo

This is the last internal data structure of the screen. It is the core of the windowing system (the layers). This will be discussed in detail later.

Offset 326: Pointer to screen gadgets

Screens also recognize gadgets that move them to the background or bring them to the foreground. This field is specifically for internal system use.

Offset 330 and 331: The screen colors

Changing the screen color values that are stored here works the same way as for windows. The new colors take effect as soon as the screen is refreshed, such as when you use a menu, etc.

Offset 332: Backup-Register

Intuition stores the color from register 0 here when the screen is flashed or beeped. The following will beep the screen:

```
PRINT CHR$(7)
```

Offset 334 and 338: External and user data

These allow the possibility to link other data blocks with the standard structure.

3.4.2 The Intuition functions for screen handling

Below are the routines from the Intuition library used for screen handling:

```
MoveScreen()
ScreenToBack()
ScreenToFront()
WBenchToBack()
WBenchToFront()
```

To make things easier we have written three SUBs that use all of these routines. They are named `ScrollScreen`, `ScreenHere` and `ScreenBye`.

`ScrollScreen` requires the number of pixels that the screen should scroll down in the current output window (a negative value scrolls it up). `ScreenBye` sends the screen behind all current screens and `ScreenHere` brings the screen to the foreground.

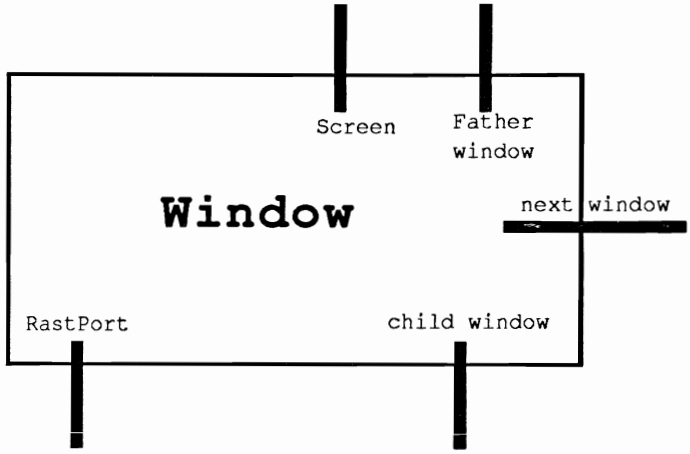
Here are the SUBs in a small demonstration program:

```
'########################################
'#
'# Section: 3.4.2
'# Program: Screen Control
'# Date:    01/04/87
'# Author:  tob
'# Version: 1.0
'#
'########################################

' Program Controlled Screen Handling

PRINT "Searching for .bmap file..."

'INTUITION-Library
'MoveScreen()
'ScreenToFront()
'ScreenToBack()

LIBRARY "intuition.library"

init:     CLS
          SCREEN 1,320,200,1,1
          WINDOW 2,"Hello!",,,1

main:    '* Screen scrolling
          PRINT "This is the 2nd Screen!"
```

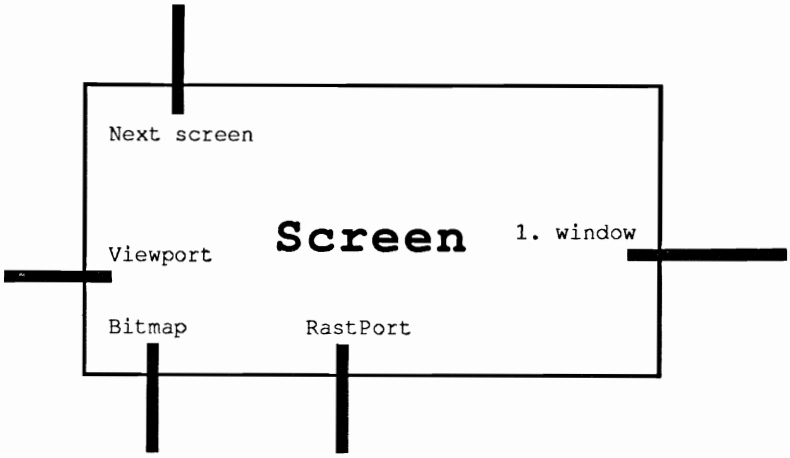
```
            '* Screen 1 down
             WINDOW OUTPUT 2
             FOR loop%=255 TO 0 STEP -1
               ScrollScreen(1)
             NEXT loop%

            '* Screen 0 down
             WINDOW OUTPUT 1
             FOR loop%=255 TO 0 STEP -1
               ScrollScreen(1)
             NEXT loop%

            '* Screen 1 up
             WINDOW OUTPUT 2
             FOR loop%=0 TO 255
               ScrollScreen(-1)
             NEXT loop%

            '* Screen 0 up
             WINDOW OUTPUT 1
             ScreenHere
             FOR loop%=0 TO 255
               ScrollScreen(-1)
             NEXT loop%

            '* Swapping
             FOR t%=1 TO 3000:NEXT t%
             ScreenBye
             FOR t%=1 TO 3000:NEXT t%
             ScreenHere
             FOR t%=1 TO 3000:NEXT t%

            '* Closing
             WINDOW CLOSE 2
             SCREEN CLOSE 1

endprog:    LIBRARY CLOSE
            END

SUB ScrollScreen(pixel%) STATIC
      screenAddress&=PEEKL(WINDOW(7)+46)
      CALL MoveScreen(screenAddress&,0,pixel%)
END SUB

SUB ScreenHere STATIC
      screenAddress&=PEEKL(WINDOW(7)+46)
      CALL ScreenToFront(screenAddress&)
END SUB

SUB ScreenBye STATIC
      screenAddress&=PEEKL(WINDOW(7)+46)
      CALL ScreenToBack(screenAddress&)
END SUB
```

3.5 Intuition and the rest of the world

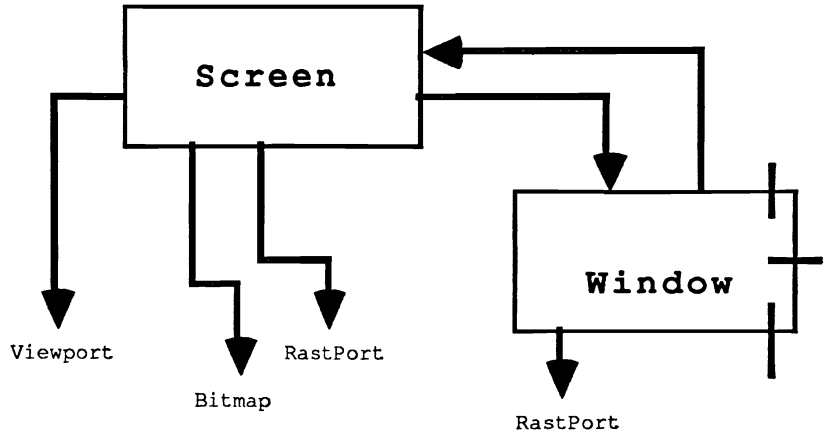
So far you have been introduced to the data structure of "windows" and "screens". It is now time for review and show you the connections between these two data blocks. The following figure symbolizes a window data structure. The exits are pointers within this structure to other components:



The "screen" structure can be displayed using a similar figure:



The next figure shows the system connection with a screen and a window:



We are not finished yet. You still need an understanding of the RastPort, ViewPort and Bit-map structures. Next we will take a closer look at the RastPort.

3.6 The RastPort

With this data block, we get even closer to the basic elements of Amiga graphics, the so called "graphic primitives". We'll leave Intuition to go deeper into the Amiga system architecture. We'll now explore the graphic libraries.

The RastPort, which contains the data that controls how a drawing takes place, manages a drawing plane. The starting address of the data block for an actual window is always in the variable `WINDOW(8)`. This address can also be read directly from the window data structure.

```
PRINT WINDOW(8)
PRINT PEEKL(WINDOW(7)+50)
```

The RastPort structure is constructed as follows:

Data structure RastPort/graphics/100 Bytes

Offset	Type	Description
+ 000	Long	Pointer to the Layer structure
+ 004	Long	Pointer to the Bit-map structure
+ 008	Long	Pointer to the AreaFill pattern
+ 012	Long	Pointer to TmpRas structure
+ 016	Long	Pointer to AreaInfo structure
+ 020	Long	Pointer to GelsInfo structure
+ 024	Byte	Mask: Writemask for this raster
+ 025	Byte	Foreground color
+ 026	Byte	Background color
+ 027	Byte	AreaFill outline color
+ 028	Byte	Drawing mode
		JAM1 = 0
		JAM2 = 1
		COMPLEMENT = 2
		INVERSEVID = 4
+ 029	Byte	AreaPtSz: 2^n Words for AreaFill pattern
+ 030	Byte	unused
+ 031	Byte	line draw pattern preshift
+ 032	Word	various control bits
		FIRST DOT = 1: draw first pixel?
		ONE DOT = 2: one dot mode for lines
		DBUFFER = 4: double buffered set
+ 034	Word	LinePtrn: 16 bits for line pattern
+ 036	Word	X coordinate of graphic cursor

Offset	Type	Description
+ 038	Word	Y coordinate of graphic cursor
+ 040	----	8x1 byte minterms
+ 048	Word	Cursor width
+ 050	Word	Cursor height
+ 052	Long	Pointer to character set
+ 056	Byte	Character set mode (bold, italics, etc.)
+ 057	Byte	textspecific flags
+ 058	Word	Height of characterset
+ 060	Word	Average character width
+ 062	Word	Text height without underline
+ 064	Word	Character spacing
+ 066	Long	Pointer to user data
+ 070	Word	reserved (7x)
+ 084	Long	reserved (2x)
+ 092	Byte	reserved (8x)

3.6.1 The RastPort data structure

The following explains the RastPort structure by offset the same manner we did for windows and screens:

Offset 0: The Layer

The Amiga uses layers to keep each drawing plane separate from the other planes. These layers are actually the principle element of each Intuition window. You might think layers are very complicated and, compared to windows, relatively useless. However, layers are important and a closer look (as seen in a later chapter) will prove that they are really a storehouse of graphics possibilities.

Offset 4: The Bit-map

You have already learned about the bit-map, which is a pointer to the bit-map structure. With this pointer you have indirect access to the screen address through the RastPort.

```
scr&=PEEKL(WINDOW(8)+4)-184
```

The bit-map is the intersection between the data structure and the RAM banks where the window and screen contents are stored.

Offset 8: Pointer to the AreaFill pattern

You have seen how to fill areas with a single color or with a pattern. If you use a pattern, then that pattern has to be stored somewhere in memory. This offset contains the address pointer to where the pattern is stored.

We will provide more information and examples after the offset explanations. One of our subjects will show you how to use multicolor patterns of up to 32 colors.

Offset 12: The TmpRas

TmpRas stands for Temporary Raster. This is a pointer to an area of free RAM used for certain operations. Whenever you use the fill commands like `PAINT` or `LINE BF`, this RAM is used as a temporary holding area for the entire object being filled.

Offset 16: AreaInfo

This pointer is for a data structure used when drawing polygon shapes. There is not much use for this from BASIC, but we will discuss it later.

Offset 20: GelsInfo

Gels are Graphics Elements, such as sprites and bobs (Blitter objects), and also the complete automatic Amiga animation system. Before this system can be activated the GelsInfo structure has to be defined. This structure contains some very important parameters.

Offset 24: Writemask

This variable allows individual bit-planes of the drawing plane to fade. The default value is normally 255. All bits are set which means that all available bit-planes are used. The `POKE`,

```
POKE WINDOW(8)+24,0
```

makes all bit-planes inactive and nothing more will be drawn on the screen. You can also select specific bit-planes to be active or inactive.

Offset 25, 26, and 27: Drawing color

These registers determine the drawing colors. The first contains the number of the color register for the drawing color. The second has the background and the third the AreaOutline mode.

Offset 28: Drawing Mode

The Amiga has four basic drawing modes that you can use. They are:

JAM 1	= 0
JAM 2	= 1
COMPLEMENT	= 2
INVERSEVID	= 4

The normal drawing mode is `JAM 2`. This means that the foreground color is used to draw in the drawing plane and the rest will be in the background color. The following example will make this clear:

(Enter these examples in Direct Mode!)

```
LINE (0,0)-(100,100),2,bf
LOCATE 1,1:PRINT "HELLO!"
```

The white text appears on a blue background. A hole is made in the original black background.

`JAM 1` is different. The foreground color is used to draw, but the background is not changed.

```
LINE (0,0)-(100,100),2,bf
POKE WINDOW(8)+28,0
LOCATE 1,1:PRINT "HELLO!"
POKE WINDOW(8)+28,1
```

`COMPLEMENT` complements the graphic with the background: Where ever a pixel is set, it will be erased, and just the opposite for unset pixels:

```
LINE (0,0)-(100,100),2,bf
POKE WINDOW(8)+28,2
LINE (50,50)-(150,150),3,bf
POKE WINDOW(8)+28,1
```

`INVERSEVID` inverts a graphic: background and foreground are exchanged. It looks like this:

```
POKE WINDOW(8)+28,4
PRINT "INVERSE!"
POKE WINDOW(8)+28,1
```

These pokes work great in direct mode. However, they do not work in a program. Nothing will happen.

The routine `SetDrMd` in the graphic library sets the desired mode safely and reliably.

```
LIBRARY "graphics.library"
CALL SetDrMd(WINDOW(8),mode%)
```

mode% = 0 - 255

Various modes may be combined, only JAM1 and JAM2 work against each other.

Offset 29: AreaPtSz

Whenever you work with patterns, you must specify how high the pattern will be. Patterns can only be defined using powers of two (1,2,4,8,...). This field stores this power of two.

Using a special programming method, you can use this register to activate the multicolor pattern mode. This allows you to create patterns with up to 32 colors. (There will be more on this in the next section).

Offset 30, 31 and 32: for system use

Offset 34: Line pattern

Patterns are not only for areas, but for lines also. The method is easy: 16 pixels in a row can be set or unset. The Amiga then uses this "sample" pattern to draw lines. Take the following line pattern for example:

```
*****.*.*****.*.
```

A dash dot dash dot line.

First you determine the bit values of the line:

$$bit\& = 2^{15}+2^{14}+2^{13}+2^{12}+2^{11}+2^{9}+2^{7}+2^{6}+2^{5}+2^{4}+2^{3}+2^{1}$$

Now the value is stored to this register:

```
POKEW WINDOW(8)+34,bit&
```

A test:

```
LINE (10,10)-(600,10)
```

As you can see, it works.

Offset 36 and 38: Coordinates of the graphic cursor

These two fields are extremely important. Text on the Amiga is nothing more than text shaped graphics. Because of this, text can be

placed in any position on the screen. The following example demonstrates this:

```
WHILE INKEY$=""
  x%=RND(1)*600
  y%=RND(1)*160
  POKEW WINDOW(8)+36,x%
  POKEW WINDOW(8)+38,y%
  PRINT "Commodore AMIGA!"
WEND
```

Offset 40-51: Minterms, internal usage

Offset 52: The character set

Just as in the screen structure this is a pointer to the currently active font. The character set determines what your text looks like. We will return to this subject later.

Offset 56: Actual text style

The Amiga can display a font in various forms on the screen:

normal	= 0
underlined	= Bit 0 set
bold	= Bit 1 set
italics	= Bit 2 set

The last three modes can be combined to form different combinations.

Offset 57: Text-Flags, internal usage

Offset 58: Text height

This field contains the height of the currently active font. This value is used to average the line height to calculate the correct line spacing.

There is nothing to prevent you from setting your own line spacing. It can even be closer together than normal:

```
POKEW WINDOW(8)+58,5
```

or further apart:

```
POKEW WINDOW(8)+58,12
```

Offset 60: Character width

This register contains the average width of the font. Since the Amiga also supports proportional fonts (characters of different widths), you can set the average width in this register.

Offset 62: Text height without underline

Offset 64: Character spacing

The following routine allows you to vary the spacing between individual characters. The default for this field is zero. Storing larger values here will spread the characters over a larger area as in the following example:

```
text$="Hello World!"
text%=LEN(text$)

FOR loop%=1 TO 40
  POKEW WINDOW(8)+36,280-(loop%*text%*.5)
  '(centering)
  POKEW WINDOW(8)+38,90
  POKEW WINDOW(8)+64,loop%
  PRINT text$
NEXT loop%

FOR loop%=39 TO 0 STEP -1
  POKEW WINDOW(8)+36,280-(loop%*text%*.5)
  POKEW WINDOW(8)+38,90
  POKEW WINDOW(8)+64,loop%
  PRINT text$
NEXT loop%

END
```

Offset 66: User data

This is a pointer to more data that can be linked to this structure.

Offset 70 to end: reserved data fields

3.7 Graphic primitives

The RastPort provides us with the graphic primitives, which is the lowest software controlled entry address to the graphics planes. First we will try out a few of the possibilities of the RastPort that we have been discussing.

3.7.1 Multicolor patterns

At the beginning of the book we introduced you to the PATTERN statement. Instead of having plain areas, you can use this statement to fill any desired area with a pattern of your own design.

```
CIRCLE (310,100),100
PAINT (310,100),2,1
```

We can also display patterned areas:

```
DIM area.pat%(3)
area.pat%(0)=&HFFFF
area.pat%(1)=&HCCCC
area.pat%(2)=&HCCCC
area.pat%(3)=&HFFFF

CIRCLE (310,100),100
PATTERN ,area.pat%
PAINT (310,100),3,1
```

You can see that this works, filling the circle with dotted orange pattern. Up to now the patterns have been limited to one color.

We have put together a complete pattern package that will enable you to design and create your patterns and add more color to them.

You know from the previous chapter that a pattern has to be stored in a specific memory area. The start address is stored in the RastPort at offset 8. The height of the pattern is stored in the RastPort field at offset 29, as a power of two. Since we know these basics of patterns, we can now write a small pattern SUB program to activate the multicolor mode. This mode is switched on whenever the power at offset 29 is negative (a power of 256).

The following program manages patterns without the PATTERN statement. It is composed of these six routines:

1. `InitPattern(howmany)%`

   This routine initializes our new pattern system. You pass the parameter for how many lines high your example pattern is to be.

   The routine calculates the required memory and calls `GetMemory`. The starting address of the new buffer is returned in `form&`.

2. `SetPat(number%,pat$)`

   This new command allows you to easily enter the individual lines of your pattern in a binary representation: An A equals an unset pixel and a B equals a set pixel. The `number%` variable passes the row number of your pattern and `pat$` passes the actual bit pattern. Make sure that `pat$` always contains 16 characters. For a solid line `pat$` would look like this:

   ```
   "BBBBBBBBBBBBBBBB"
   ```

3. `MonoPattern`

   This routine is called without any arguments and activates the pattern system. Both RastPort addresses are initialized to their correct values in this routine.

4. `EndPattern`

   This routine is also called without any arguments. It tells the Amiga that your example pattern will no longer be used. The memory reserved for your pattern is released back to the system. You should call `EndPattern` at the end of your program to release all memory back to the system.

5. `GetMemory(size&)`

   This is an all purpose memory get routine. Whenever you need memory, `GetMemory` gets it for you. You specify an `&` variable containing the amount of memory you need in bytes and call this routine with your argument. The start address of the memory area reserved is returned in the same variable.

To reserve an area of 1245 bytes:

```
DECLARE FUNCTION AllocMem& LIBRARY
LIBRARY "exec.library"

myMem&=1245
GetMemory myMem&
PRINT "Starting address: ";myMem&
```

Note: If a value of zero is returned for the starting address the memory allocation did not work. You did not receive a memory area and cannot use `FreeMemory` to return it.

6. `FreeMemory(add&)`

When you no longer require the memory you have reserved you return it to the system with `FreeMemory`. A call to this routine looks like this:

```
DECLARE FUNCTION AllocMem& LIBRARY
LIBRARY "exec.library"

form&=100  '100 Bytes, please!
GetMemory form&

(...)

FreeMemory form&  'memory released
LIBRARY CLOSE
```

These six routines make working with patterns remarkably easy. Here are the routines along with a small demonstration program. The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 3.7.1A ¶
'# Program: Mono-Pattern Module¶
'# Date:    12/27/86¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' Easy Definition of Patterns using¶
' binary character strings¶
¶
PRINT "Searching for .bmap file..."¶
¶
'EXEC-Library¶
DECLARE FUNCTION AllocMem& LIBRARY¶
```

```
'FreeMem()¶
¶
LIBRARY "exec.library"¶
¶
init:       '* Activate System¶
            CLS¶
            InitPattern 4¶
¶
           '* SetPattern¶
            SetPat 0,"BBBBBBBBBBBBBBBB"¶
            SetPat 1,"BAAAAAAAAAAAAAAB"¶
            SetPat 2,"BAAAABBBBBBAAAAB"¶
            SetPat 3,"BAAAABBBBBBAAAAB"¶
¶
           '* Enable New Pattern¶
            MonoPattern¶
¶
           '* Display Pattern on Screen¶
            LINE   (10,10)-(100,100),1,bf¶
            CIRCLE (300,100),100¶
            PAINT  (300,100),2,1¶
¶
           '* Switch System back Off¶
            EndPattern¶
¶
endprog:    LIBRARY CLOSE¶
            END¶
¶
SUB InitPattern(howmany%) STATIC¶
      SHARED form&,plane1%,plane2%,kolor%¶
¶
     '* Is it a power of 2 ?¶
      IF LOG(howmany%)/LOG(2)<>INT(LOG(howmany%)/LOG(2))
THEN¶
        PRINT "2^x! Power of two for InitPattern!
1,2,4,8,16.."¶
        ERROR 17¶
      END IF¶
¶
     '* Read Parameters¶
      planes% = PEEK(PEEKL(WINDOW(8)+4)+5)¶
      DIM SHARED p&(howmany%*planes%)¶
      plane1% = planes%¶
      plane2% = howmany%¶
      kolor% = 2^plane1%-1¶
¶
     '* Definition Pattern Buffer allocate¶
      form& = howmany%*2*planes%¶
      GetMemory form&¶
END SUB¶
¶
SUB SetPat(number%,pat$) STATIC¶
      SHARED form&,plane1%,plane2%¶
¶
     '* Too many Rows?¶
      IF number%>=plane2% THEN¶
```

```
        PRINT "More Rows than you can define with
InitPattern!"¶
        EndPattern¶
        ERROR 17¶
      END IF¶
¶
     '* Error-Handling: Limit string to 16 bytes¶
      IF LEN(pat$)<16 THEN¶
        pat$=pat$+STRING$(16-LEN(pat$),"A")¶
      END IF¶
¶
     '* Read Definition Pattern¶
      FOR loop1% = 0 TO 15¶
        check$ = UCASE$(MID$(pat$,loop1%+1,1))¶
        col%   = ASC(check$)-65¶
        IF col%>=2^plane1% OR col%<0 THEN col%=0¶
        FOR loop2% = col% TO 0 STEP -1¶
          IF col%> = 2^loop2% THEN¶
            col%                          = col%-2^loop2%¶
            p&(number%+loop2%*plane2%) =
p&(number%+loop2%*plane2%)+2^(15-loop1%)¶
          END IF¶
        NEXT loop2%¶
      NEXT loop1%¶
¶
     '* Write value to Buffer¶
      FOR loop3% = 0 TO plane2%*plane1%¶
        POKEW form&+2*loop3%,p&(loop3%)¶
      NEXT loop3%  ¶
END SUB            ¶
¶
SUB MonoPattern STATIC¶
      SHARED form&,plane2%¶
      planes% = LOG(plane2%)/LOG(2)¶
      POKEL WINDOW(8)+8, form&¶
      POKE  WINDOW(8)+29,planes%¶
END SUB¶
¶
SUB EndPattern STATIC¶
      SHARED form&¶
¶
     '* Pattern off and release memory¶
      POKEL WINDOW(8)+8, 0¶
      POKE  WINDOW(8)+29,0¶
      FreeMemory form&¶
END SUB ¶
¶
SUB GetMemory(size&) STATIC¶
        mem.opt&  = 2^0+2^1+2^16¶
        RealSize& = size&+4¶
        size&     = AllocMem&(RealSize&,opt&)¶
        IF size&  = 0 THEN ERROR 255¶
        POKEL   size&,RealSize&¶
        size& = size&+4¶
END SUB¶
¶
```

```
SUB FreeMemory(add&) STATIC¶
        add&       = add&-4¶
        RealSize& = PEEKL(add&)¶
        CALL FreeMem(add&,RealSize&)¶
END SUB¶
```

So far our SUB programs do not provide colored patterns. To fix this, we'll add the routine `ColorPattern` to our program. It replaces `MonoPattern` and activates the multicolor system. Depending on your screen depth, you can use up to 32 colors in your patterns. Additional characters are available now for defining your pattern:

Color 1	A	Color register 0 (background)
Color 2	B	Color register 1
Color 3	C	Color register 2
Color 4	D	Color register 3

(for the Workbench screen)

The following program demonstrates multicolor patterns. The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 3.7.1B¶
'# Program: Mono & Color Pattern¶
'# Date:    12/27/86¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' Makes "Multi-Color" Patterns possible¶
' (via RastPort manipulation)¶
¶
PRINT "Searching for .bmap file..."¶
¶
'EXEC-Library¶
DECLARE FUNCTION AllocMem& LIBRARY¶
'FreeMem()¶
¶
LIBRARY "exec.library"¶
¶
init:       CLS¶
            rows%=8¶
            InitPattern rows%¶
            '         0123456789ABCDEF ¶
            SetPat 0,"DCDAAAAAAABBAAAA"¶
            SetPat 1,"DCDAAAAAABBBBAAA"¶
            SetPat 2,"DCDAAAAABAABBAAA"¶
            SetPat 3,"DCDAAAABAAAABBAA"¶
            SetPat 4,"DCDAAABBBBBBBBBA"¶
```

```
            SetPat 5,"DCDAABAAAAAAABBA"¶
            SetPat 6,"DCDBBBBAAAAABBBB"¶
            SetPat 7,"CCCCCCCCCCCCCCCC"¶
¶
drawing:    PRINT TAB(7);"MONO";TAB(36);"COLOR!"¶
¶
            MonoPattern¶
            CIRCLE (60,60),60¶
            PAINT  (60,60),kolor%,1¶
¶
            ColorPattern ¶
            CIRCLE (310,100),100¶
            PAINT  (310,100),kolor%,1¶
¶
endprog:    EndPattern¶
            LIBRARY CLOSE¶
            END¶
¶
SUB ColorPattern STATIC¶
      SHARED form&,plane2%¶
      planes% = LOG(plane2%)/LOG(2)¶
      POKEL WINDOW(8)+8,form&¶
      POKE  WINDOW(8)+29,256-planes%¶
END SUB¶
¶
SUB InitPattern(howmany%) STATIC¶
      SHARED form&,plane1%,plane2%,kolor%¶
¶
     '* Is it a power of 2 ?¶
      IF LOG(howmany%)/LOG(2)<>INT(LOG(howmany%)/LOG(2))
THEN¶
        PRINT "2^x! A power of two for InitPattern!
1,2,4,8,16..."¶
        ERROR 17¶
      END IF¶
¶
     '* Read Parameters¶
      planes% = PEEK(PEEKL(WINDOW(8)+4)+5)¶
      DIM SHARED p&(howmany%*planes%)¶
      plane1% = planes%¶
      plane2% = howmany%¶
      kolor% = 2^plane1%-1¶
¶
     '* Reserve Definition Pattern Buffer¶
      form& = howmany%*2*planes%¶
      GetMemory form&¶
END SUB¶
¶
SUB SetPat(number%,pat$) STATIC¶
      SHARED form&,plane1%,plane2%¶
¶
     '* Too many Rows?¶
      IF number%>=plane2% THEN¶
        PRINT "More rows than can be defined with
InitPattern!"¶
        EndPattern¶
```

```
        ERROR 17¶
      END IF¶
¶
     '* Error-Handling: Limit string to 16 bytes¶
      IF LEN(pat$)<16 THEN¶
        pat$=pat$+STRING$(16-LEN(pat$),"A")¶
      END IF¶
¶
     '* Read Pattern Definition¶
      FOR loop1% = 0 TO 15¶
        check$ = UCASE$(MID$(pat$,loop1%+1,1))¶
        col%   = ASC(check$)-65¶
        IF col%>=2^plane1% OR col%<0 THEN col%=0¶
        FOR loop2% = col% TO 0 STEP -1¶
          IF col%> = 2^loop2% THEN¶
            col%                          = col%-2^loop2%¶
            p&(number%+loop2%*plane2%) =
p&(number%+loop2%*plane2%)+2^(15-loop1%)¶
          END IF¶
        NEXT loop2%¶
      NEXT loop1%¶
¶
     '* Write value to Buffer¶
      FOR loop3% = 0 TO plane2%*plane1%¶
        POKEW form&+2*loop3%,p&(loop3%)¶
      NEXT loop3%  ¶
END SUB            ¶
¶
SUB MonoPattern STATIC¶
      SHARED form&,plane2%¶
      planes% = LOG(plane2%)/LOG(2)¶
      POKEL WINDOW(8)+8, form&¶
      POKE  WINDOW(8)+29,planes%¶
END SUB¶
¶
SUB EndPattern STATIC¶
      SHARED form&¶
¶
     '* Pattern off and release memory¶
      POKEL WINDOW(8)+8, 0¶
      POKE  WINDOW(8)+29,0¶
      FreeMemory form&¶
END SUB ¶
¶
SUB GetMemory(size&) STATIC¶
        mem.opt&  = 2^0+2^1+2^16¶
        RealSize& = size&+4¶
        size&     = AllocMem&(RealSize&,opt&)¶
        IF size&  = 0 THEN ERROR 255¶
        POKEL   size&,RealSize&¶
        size& = size&+4¶
END SUB¶
¶
SUB FreeMemory(add&) STATIC¶
        add&      = add&-4¶
        RealSize& = PEEKL(add&)¶
```

```
        CALL FreeMem(add&,RealSize&)¶
END SUB¶
```

If the four Workbench colors are not enough, you can create your own screen with more than 2 bit-planes. The following program does just this. After the initialization you will see, as a fill pattern, a familiar logo in 11 colors. If you hold down the left mouse button, you can draw using this pattern. The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 3.7.1C ¶
'# Program: Multi-Color-Pattern¶
'# Date:    12/27/86¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
'Demonstrates the use of a Multi-Color Pattern with¶
'up to 16 Colors (Screen Depth = 4); up to 32 Colors
possible¶
'(colorvalue: A=0 to Z=25, Colors 26-32 = chr$(91)-
chr$(97) )¶
'On OUT OF HEAP SPACE close other Window!¶
¶
PRINT "Searching for .bmap file..."¶
¶
'EXEC-Library¶
DECLARE FUNCTION AllocMem& LIBRARY¶
'FreeMem()¶
¶
LIBRARY "exec.library"¶
¶
init:       SCREEN 1,640,200,4,2¶
            WINDOW 1,"Hello!",,,1¶
¶
            LOCATE 4,15¶
            PRINT "** Patience! **"¶
¶
            rows%=8¶
            InitPattern rows%¶
            '         0123456789ABCDEF ¶
            SetPat 0,"AAAAAAAAAAABBABB"¶
            SetPat 1,"AAAAAAAAAABBABBA"¶
            SetPat 2,"AAAAAAAACCACCAA"¶
            SetPat 3,"AAAAAAADDADDAAA"¶
            SetPat 4,"FFAFFAAEEAEEAAAA"¶
            SetPat 5,"AGGAGGHHAHHAAAAA"¶
            SetPat 6,"AAKKAIIAIIAAAAAA"¶
            SetPat 7,"AAAJJJJJJAAAAAAA"¶
¶
```

```
patcol:     '* Color choices for the pattern¶
            PALETTE 0,0,0,0      'A¶
            PALETTE 1,.9,.3,.4   'B¶
            PALETTE 2,.8,.5,.4   'C¶
            PALETTE 3,.8,.6,0    'D¶
            PALETTE 4,1,.8,0     'E¶
            PALETTE 5,0,0,.6     'F¶
            PALETTE 6,0,.3,.6    'G¶
            PALETTE 7,.7,.9,0    'H¶
            PALETTE 8,.3,.9,0    'I¶
            PALETTE 9,0,.5,0     'J¶
            PALETTE 10,0,.3,0!   'K¶
¶
drawing:    ColorPattern¶
            LOCATE 3,10¶
¶
            PRINT "Press Left Mouse Button to Draw"¶
            PRINT TAB(10);"Touch Left Screen Border to
Exit"¶
            CIRCLE (310,100),100¶
            PAINT  (310,100),kolor%,1¶
¶
mousecontr: test% = MOUSE(0)¶
            WHILE MOUSE(1)<>0¶
              x%  = MOUSE(1)¶
              y%  = MOUSE(2)¶
              IF test%<>0 THEN¶
                LINE (x%,y%)-(x%+10,y%+5),kolor%,bf¶
              END IF¶
              test% = MOUSE(0)¶
            WEND  ¶
¶
endprog:    EndPattern¶
¶
            WINDOW 1,"Demo Over.",,,-1¶
¶
            SCREEN CLOSE 1¶
            LIBRARY CLOSE¶
            END¶
¶
SUB ColorPattern STATIC¶
      SHARED form&,plane2%¶
      planes% = LOG(plane2%)/LOG(2)¶
      POKEL WINDOW(8)+8,form&¶
      POKE  WINDOW(8)+29,256-planes%¶
END SUB¶
¶
SUB InitPattern(howmany%) STATIC¶
      SHARED form&,plane1%,plane2%,kolor%¶
¶
     '* Is it a power of 2 ?¶
      IF LOG(howmany%)/LOG(2)<>INT(LOG(howmany%)/LOG(2))
THEN¶
        PRINT "2^x! Power of two for InitPattern!
1,2,4,8,16."¶
        ERROR 17¶
```

```
      END IF¶
¶
     '* Read parameters¶
      planes% = PEEK(PEEKL(WINDOW(8)+4)+5)¶
      DIM SHARED p&(howmany%*planes%)¶
      plane1% = planes%¶
      plane2% = howmany%¶
      kolor%  = 2^plane1%-1¶
¶
     '* Allocate Pattern Definition Buffer¶
      form& = howmany%*2*planes%¶
      GetMemory form&¶
END SUB¶
¶
SUB SetPat(number%,pat$) STATIC¶
      SHARED form&,plane1%,plane2%¶
¶
     '* Too many Rows?!¶
      IF number%>=plane2% THEN¶
        PRINT "More rows than can be defined with
InitPattern!"¶
        EndPattern¶
        ERROR 17¶
      END IF¶
¶
     '* Error-Handling: Limit String to 16 bytes¶
      IF LEN(pat$)<16 THEN¶
        pat$=pat$+STRING$(16-LEN(pat$),"A")¶
      END IF¶
¶
     '* Read Pattern Definition¶
      FOR loop1% = 0 TO 15¶
        check$ = UCASE$(MID$(pat$,loop1%+1,1))¶
        col%   = ASC(check$)-65¶
        IF col%>=2^plane1% OR col%<0 THEN col%=0¶
        FOR loop2% = col% TO 0 STEP -1¶
          IF col%> = 2^loop2% THEN¶
            col%                          = col%-2^loop2%¶
            p&(number%+loop2%*plane2%) = p&(number%+
loop2%*plane2%)+2^(15-loop1%)¶
          END IF¶
        NEXT loop2%¶
      NEXT loop1%¶
¶
     '* Write value to Buffer¶
      FOR loop3% = 0 TO plane2%*plane1%¶
        POKEW form&+2*loop3%,p&(loop3%)¶
      NEXT loop3%  ¶
END SUB            ¶
¶
SUB MonoPattern STATIC¶
      SHARED form&,plane2%¶
      planes% = LOG(plane2%)/LOG(2)¶
      POKEL WINDOW(8)+8, form&¶
      POKE  WINDOW(8)+29,planes%¶
END SUB¶
```

```
¶
SUB EndPattern STATIC¶
      SHARED form&¶
¶
     '* Pattern Off and release memory¶
      POKEL WINDOW(8)+8, 0¶
      POKE  WINDOW(8)+29,0¶
      FreeMemory form&¶
END SUB ¶
¶
SUB GetMemory(size&) STATIC¶
        mem.opt&  = 2^0+2^1+2^16¶
        RealSize& = size&+4¶
        size&     = AllocMem&(RealSize&,opt&)¶
        IF size&  = 0 THEN ERROR 255¶
        POKEL   size&,RealSize&¶
        size& = size&+4¶
END SUB¶
¶
SUB FreeMemory(add&) STATIC¶
        add&      = add&-4¶
        RealSize& = PEEKL(add&)¶
        CALL FreeMem(add&,RealSize&)¶
END SUB¶
```

3.7.2 Shadows using cursor positioning

The multicolored patterns in the last section were created by skillfully manipulating the RastPorts. With some creativity, you can do a lot more with these data structures. To demonstrate this, we will show you what can be done with a little help from offset fields 28, 36, and 38.

These fields manage:

Offset	Type	Description		
+ 028	Byte	Drawing modes		
		JAM1	= 0	
		JAM2	= 1	
		COMPLEMENT		= 2
		INVERSEVID	= 4	
+ 036	Word	X coordinate of graphic cursor		
+ 038	Word	Y coordinate of graphic cursor		

With these offsets, we will create a shadowed text effect. You may have seen this method on television, which has used it for a long time. First the text is printed in black, then the text is printed again in white over the black but slightly offset. This creates a shadow effect that makes

the text visibly stand out. It doesn't matter if the background is dark or light, the contrast is still visible.

We will use three routines from the graphic library to create our effect:

```
SetDrMd()
Text()
Move()
```

The first routine, `SetDrMd()`, sets the drawing mode and directly affects RastPort offset 28 (see Section 3.6.1). The text routine, `Text()`, which was discussed in Chapter 2, prints text on the screen. The move routine, `Move()`, puts the graphic cursor at a selected position and affects RastPort offsets 36 and 38. Instead of using the `Move()` routine, you could poke the values into the memory offsets.

Our routine is named "shadows" and it requires two arguments:

```
Shadow text$,mode%
```

`text$:`	The text that is to be displayed
`mode%:`	0 = `PRINT text$`
	1 = `PRINT text$;`

Here is the program:

```
'########################################
'#
'# Section: 3.7.2
'# Program: Shadow Print
'# Datum:   12/25/86
'# Author:  tob
'# Version: 1.0
'#
'########################################

PRINT "Searching for .bmap files..""

'GRAPHICS-Library
'Text()
'Move()
'SetDrMd()

LIBRARY "graphics.library"

main:       '* Contrast Color
            PALETTE 0,.5,.5,.5
            CLS
            LOCATE 5,1
            PRINT  "This is the Normal text without
contrast."
            PRINT  "Not very exciting..."
```

```
                PRINT
                Shadow "Shadow print is just as fast as
PRINT!",0
                Shadow "It works due to the actions of
the",0
                Shadow "Text() function of the Graphic-
Library!",0
                PRINT
                Shadow "The text effectively appears to
float",0
                Shadow "in FRONT of the screen.",0

endprog:        LIBRARY CLOSE
                END

SUB Shadow(Text$,mode%) STATIC

      '* Lock in parameters
       textlen% = LEN(Text$)
       depth%   = 2
       cX%      = PEEKW(WINDOW(8)+36)
       cY%      = PEEKW(WINDOW(8)+38)

      '* Draw shadows
       COLOR 2,0
       CALL  Move(WINDOW(8),cX%+depth%,cY%+depth%)
       CALL  Text(WINDOW(8),SADD(Text$),textlen%)

      '* Draw JAM1 and Foreground
       CALL  SetDrMd(WINDOW(8),0)
       COLOR 1,0
       CALL  Move(WINDOW(8),cX%,cY%)
       CALL  Text(WINDOW(8),SADD(Text$),textlen%)

      '* CR as desired
       IF mode% = 0 THEN
         PRINT
       END IF

      '* and again with JAM2 and finish
       CALL SetDrMd(WINDOW(8),1)
END SUB
```

During the drawing process it is necessary to switch the drawing mode from JAM2 to JAM1. Otherwise the white text would completely cover the black text.

We use the text function instead of the PRINT statement to gain more speed. Text is more than three times faster than PRINT. This is why our shadow text is displayed faster than the normal text. To see the difference just switch the lines around:

```
CALL Text(WINDOW(8),SADD(text$),textlen%)
```

for the line

```
PRINT text$
```

The difference in speed is enormous.

3.7.3 Outline print - a special flair

This type of text displays only the silhouette of the characters. It quickly catches the eye and is especially useful for creating titles.

You can access this mode by using the following technique. The text is output using drawing mode JAM1 and, while being displayed, is moved one pixel in all directions. The result is a smeared character. Now the text is output again in the original position using the background color. The final result is a silhouette of the character.

Here is our routine named Outline which is very similar to the previous shadow routine. The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 3.7.3 ¶
'# Program: Outline Print¶
'# Date:    12/25/86¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
PRINT "Searching for .bmap file..."¶
¶
 'GRAPHICS-Library¶
 'Text()¶
 'Move()¶
 'SetDrMd()¶
¶
LIBRARY "graphics.library"¶
¶
main:         CLS¶
              LOCATE 8,1¶
              Outline "  OUTLINE PRINT stands out and
catches the eye!",0¶
              Outline "  Although the drawing process is
complicated, it is",0¶
              Outline "  extremely fast due to the use of
the Text() function.",0¶
```

```
                Outline "  OUTLINE will also work with other
character sets.",0¶
¶
endprog:        LIBRARY CLOSE¶
                END¶
¶
SUB Outline(text$,mode%) STATIC¶
¶
        '* Parameter¶
         textlen% = LEN(text$)¶
         cX%      = PEEKW(WINDOW(8)+36)¶
         cY%      = PEEKW(WINDOW(8)+38)¶
¶
        '* JAM1 and Smear Text¶
        '* a loop is faster and more effective¶
         CALL SetDrMd(WINDOW(8),0)¶
         CALL Move(WINDOW(8),cX%+1,cY%)¶
         CALL text(WINDOW(8),SADD(text$),textlen%)¶
         CALL Move(WINDOW(8),cX%-1,cY%)¶
         CALL text(WINDOW(8),SADD(text$),textlen%)¶
         CALL Move(WINDOW(8),cX%,cY%+1)¶
         CALL text(WINDOW(8),SADD(text$),textlen%)¶
         CALL Move(WINDOW(8),cX%,cY%-1)¶
         CALL text(WINDOW(8),SADD(text$),textlen%)¶
         CALL Move(WINDOW(8),cX%-1,cY%-1)¶
         CALL text(WINDOW(8),SADD(text$),textlen%)¶
         CALL Move(WINDOW(8),cX%+1,cY%+1)¶
         CALL text(WINDOW(8),SADD(text$),textlen%)¶
         CALL Move(WINDOW(8),cX%+1,cY%-1)¶
         CALL text(WINDOW(8),SADD(text$),textlen%)¶
         CALL Move(WINDOW(8),cX%-1,cY%+1)¶
         CALL text(WINDOW(8),SADD(text$),textlen%)¶
¶
        '* Background color and hollow Text¶
         COLOR 0,0¶
         CALL  Move(WINDOW(8),cX%,cY%)¶
         CALL  text(WINDOW(8),SADD(text$),textlen%)¶
¶
        '* Reset Modes and Color, CR as needed¶
         COLOR 1,0¶
         IF mode%=0 THEN¶
           PRINT¶
         END IF¶
         CALL SetDrMd(WINDOW(8),1)¶
END SUB¶
```

3.7.4 Text styles

The four different text styles of the Amiga can be program controlled. The value stored at RastPort offset 56 controls how the Amiga displays the text.

a) normal
b) **bold**
c) underlined
d) *italics*

In addition to these, you can combine the styles. There are basically two ways to switch between these styles:

a) Direct RastPort manipulation

With this method you POKE directly to the RastPort to switch. It works like this:

```
normal%=0
underline%=2^0
bold%=2^1
italic%=2^2

POKE WINDOW(8)+56,underline%
PRINT "Underlined Text!"

POKE WINDOW(8)+56,bold%
PRINT "Bold Text."

POKE WINDOW(8)+56,italic%+underline%
PRINT "Combined: Italic and underlined"

POKE WINDOW(8)+56,normal%
```

b) Via graphic libraries

This library has two functions that handle text styles:

`AskSoftStyle()` and `SetSoftStyle()`

The principle is similar. Again the four types are available and can be mixed. The call to `SetSoftStyle` requires a third parameter:

```
newStyle%=SetSoftStyle%(RastPort,mode,enable)
```

`RastPort:`	Address of RastPort
`mode:`	The desired style
`enable:`	The available styles

It is possible that a font won't be compatible with a specific style and therefore, will not be legible. To prevent this, the graphic library provides the enable field. The `AskSoftStyle` function checks a mask that returns all the legal styles available for the current font. This value is then returned to `SetSoftStyle`. Here is a sample program:

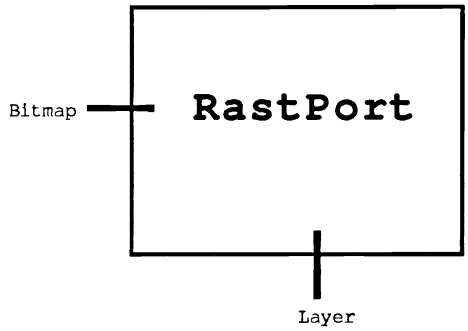
```
'########################################¶
'#¶
'# Section: 3.7.4¶
'# Program: SoftStyle¶
'# Date:    12/20/86¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' Demonstrates the use of different text styles¶
' named "SoftStyles" that are software¶
' controlled.¶
¶
PRINT "Searching for .bmap file..."¶
¶
' GRAPHICS-Library¶
DECLARE FUNCTION AskSoftStyle& LIBRARY¶
DECLARE FUNCTION SetSoftStyle& LIBRARY¶
' SetDrMd()¶
¶
LIBRARY "graphics.library"¶
¶
init:     normal        = 0¶
          underline     = 2^0¶
          bold          = 2^1¶
          italic        = 2^2¶
          CLS¶
¶
main:'* JAM1 for legible slanted text¶
      CALL   SetDrMd(WINDOW(8),0)¶
      LOCATE 4,1¶
      SetStyle underline+bold¶
      PRINT TAB(8);"ALGORITHEM GENERATED TEXT STYLES"¶
      PRINT¶
      SetStyle normal¶
      PRINT "The Amiga can do a lot with the existing
fonts."¶
      PRINT "Without a ";¶
      SetStyle underline¶
      PRINT "Definition change";¶
      SetStyle normal¶
      PRINT " you can use these styles:"¶
      PRINT¶
      SetStyle bold¶
      PRINT "BOLD Print"¶
      SetStyle italic¶
      PRINT "ITALIC Print"¶
      SetStyle underline¶
      PRINT "UNDERLINE Text"¶
      SetStyle underline+italic¶
      PRINT "and MIXED."¶
      PRINT¶
      SetStyle normal¶
      PRINT "These tricks let you display text
prfessionally."¶
```

```
      CALL SetDrMd(WINDOW(8),1)¶
¶
LIBRARY CLOSE¶
END¶
¶
SUB SetStyle(mode) STATIC¶
  '0!   = normal¶
  '2^0  = underline¶
  '2^1  = bold¶
  '2^2  = italic¶
  mode% = CINT(mode)¶
¶
 '* Check font and if possible set new style¶
  enable%   = AskSoftStyle&(WINDOW(8))¶
  newStyle% = SetSoftStyle&(WINDOW(8),mode%,enable%)¶
END SUB¶
```

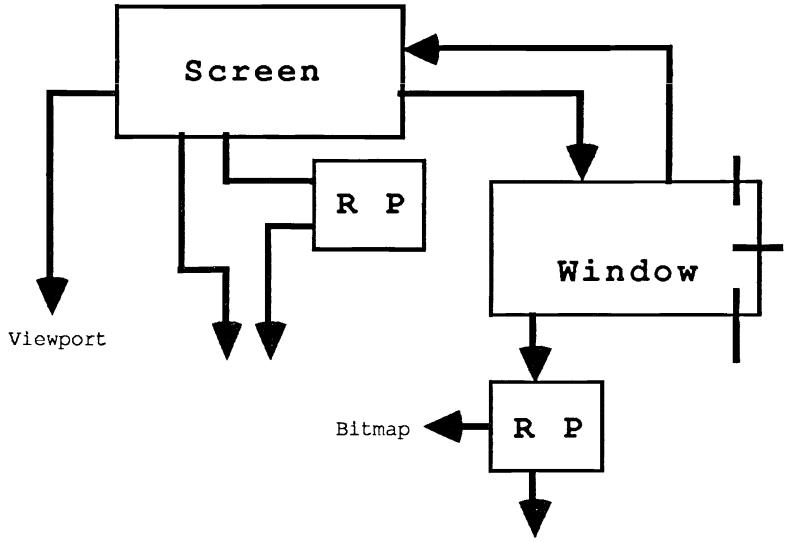
You probably noticed that the italic print in the first example was somewhat deformed but in the above example the italics looked much better. The reason for this is that italics only work correctly in JAM1 mode. When characters are slanted, the right side of each character extends slightly into the next character's position. If the normal JAM2 mode is active, this overlap is covered by the background color making the character look chopped off.

3.8 The RastPort and the graphic system

We'll now look at RastPort in relation to the graphic system of the Amiga. You will see that the RastPort becomes a component of windows and screens.



With this we can represent the entire system much better. Below we have added the RastPort to the figure from Section 3.5:



We still need information about ViewPort, Bit-map, and Layers. However, the picture we are building of the Amiga system is becoming much clearer. Our next subject is the bit-map data structure.

3.9 The Bit-map structure

Through this structure we gain access to the RAM banks where the screen contents are stored. This data structure is 40 bytes long:

Data structure Bitmap/graphics/40 Bytes

Offset	Type	Description
+ 000	Word	Bytes per display line
+ 002	Word	Number of display lines
+ 004	Byte	System flag (unused)
+ 005	Byte	Number of bit-planes (depth)
+ 006	Word	unused
+ 008	Long	Pointer to 1st bit-plane
+ 012	Long	Pointer to 2nd bit-plane
+ 016	Long	Pointer to 3rd bit-plane
+ 020	Long	Pointer to 4th bit-plane
+ 024	Long	Pointer to 5th bit-plane
+ 028	Long	Pointer to 6th bit-plane
+ 032	Long	Pointer to 7th bit-plane
+ 036	Long	Pointer to 8th bit-plane

You obtain the starting address of this data structure like this:

```
bitmap&=PEEKL(WINDOW(8)+4)
```

The first two fields contain the screen measurements stored by the bit-planes:

```
bitmap&=PEEKL(WINDOW(8)+4)
x%=PEEKW(bitmap&)*8
y%=PEEKW(bitmap&+2)
PRINT "Extent: horiz. ";x%;

PRINT "vert. ";y%
```

The fourth field contains the number of used bit-planes. Presently you can only have up to six active bit-planes. The pointers for planes seven and eight are for future expansion compatibility of the Amiga.

4. The ViewPort

We have now covered almost all of the graphic hardware. The ViewPort, which consists of a data block of 40 bytes, represents the most elementary display of the Amiga.

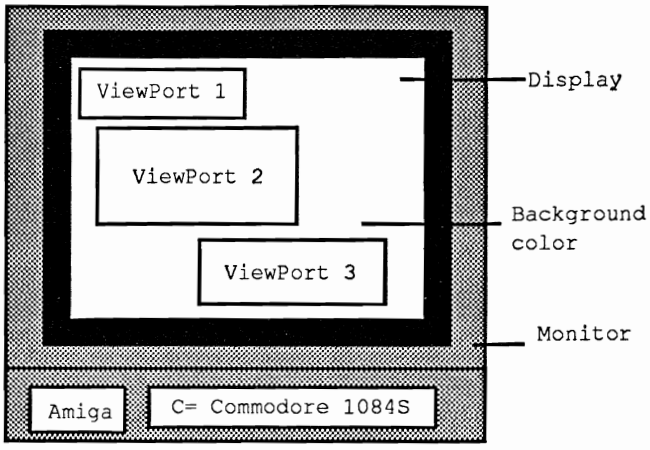
Data structure ViewPort/graphics/40 Bytes:

Offset	Type	Description
+ 000	Long	Pointer to next ViewPort
+ 004	Long	Pointer to ColorMap
+ 008	Long	DspIns: Copper list from MakeView
+ 012	Long	SprIns: Copper list for sprites
+ 016	Long	ClrIns: Copper list for sprites
+ 020	Long	UCopIns: User Copper list
+ 024	Word	Width of display
+ 026	Word	Height of display
+ 028	Word	X Offset from (0,0) of bit-plane
+ 030	Word	Y Offset from (0,0) of bit-plane
+ 032	Word	ViewPort mode
		Bit 1: 1=GENLOCK VIDEO
		Bit 2: 1=INTERLACE
		Bit 6: 1=PFBA
		Bit 7: 1=EXTRA HALFBRITE
		Bit 8: 1=GENLOCK AUDIO
		Bit 10: 1=DUALPF
		Bit 11: 1=HAM
		Bit 13: 1=VP-HIDE
		Bit 14: 1=SPRITES
		Bit 15: 1=HIRES
+ 034	Word	reserved
+ 036	Long	Pointer to RasInfo structure

Before we cover the offset descriptions, we must explain the importance of a ViewPort.

A ViewPort is a data structure in RAM which represents a part of what you see on the screen. If you take a close look at the screen structure in Section 3.4, you will see that the ViewPort is part of this structure. The reason for this is that an Intuition screen is really a ViewPort with accessories. So the heart of every screen is a ViewPort.

A display consists of one or more ViewPorts. The following figure illustrates this:



ViewPorts have the following restrictions: 1.) it is not possible to place ViewPorts next to each other, 2.) a ViewPort cannot overlap another ViewPort, and 3.) there must be at least one pixel row between them.

Each ViewPort can have its own graphic attributes like color and bit-plane. Also, the ViewPort itself is further divided among several areas called windows. However, windows do not have the same limitations as ViewPorts and they can overlap each other.

Now we will present a detailed description of the data structure.

4.1 The ViewPort data structure

Offset 0: Next ViewPort

A display can consist of one or more ViewPorts connected in a chain. This field points to the next ViewPort of a display. If there are no further ViewPorts, this field equals 0.

Offset 4: Colormap

Every ViewPort can define its own color. This offset points to a data structure called "colormap" which stores the RGB value of this color. Depending on how many bit-planes are available, a ViewPort can use up to 32 individual colors (without using special modes).

Offset 8, 12, 16 and 20: Copper List

The Copper, which is one of three Amiga graphic coprocessors, controls the entire display, manipulates registers, moves sprites and programs the Blitter (the copy processor). A special programming language, designed exclusively for the Copper, consists of only three commands. This field of the ViewPort structure contains the Copper command list which the Copper uses to build a display. The first list determines how to link the other three lists and use them to display the ViewPort.

Offset 24 and 26: Width and Height

These two offsets contain the width and height of the display section controlled by this ViewPort.

Offset 28 and 30: Bitmap Offset

At this offset we find the coordinates of the upper left hand corner of the ViewPort relative to the complete display. These values are used to position the ViewPort. DyOffset can vary between -16 and +200 (in interlace -32 to +400). DxOffset can vary between -16 and +352 (in hi-res -32 to +704).

Offset 32: The ViewPort Modes

The Amiga has many different graphic modes. The most well known are hi-res (640 pixels horizontal) and interlace (400 pixels vertically). This field contains a value for the current mode.

Offset 36: The RasInfo Block

Every ViewPort has at least one RasInfo data structure connected to it. We will discuss this further on.

4.2 The graphic modes of the Amiga

The Amiga has the following nine special graphic modes:

Genlock Video
Interlace
PFBA
Extra Halfbrite
DUALPF
HAM
VP-Hide
Sprites
Hi-Res

You should already be familiar with the hi-res and interlace modes supported by AmigaBASIC. The AmigaBASIC SCREEN statement can specify screens from lo-res (normal, 320 pixels wide), hi-res (640 pixels wide), to interlace (400 instead of 200 pixels high).

If the ViewPort will contain sprites or vsprites, you must set the sprite flag (this is normally true).

VP-Hide is set whenever this ViewPort is covered by other ViewPorts (for example, a screen covers another screen). The other ViewPort is not displayed.

Genlock Video means that instead of the background color, an external video signal, like a video recorder or camera, provides the background. You need a genlock interface to use this mode.

DUALPF stands for "Dual Playfield". This mode allows you to display two different planes in one ViewPort. The background of the top plane becomes transparent to show the plane underneath.

PFBA works with the dual playfield mode. It determines the video priority of the two planes.

HAM stands for hold-and-modify. This mode allows you to display all 4096 colors of the Amiga on the screen at the same time. However, this display mode is very difficult to program. More on this subject shortly.

Extra halfbrite is a new graphic mode that allows you to display up to 64 colors at the same time instead of the normal 32.

4.2.1 The halfbrite mode

Extra halfbrite is one of the special graphic modes not supported by the BASIC SCREEN statement. It is therefore impossible to create a screen in this mode using SCREEN.

It is possible to convert an existing screen to a halfbrite screen. Before we show you how to do this, we are going to explain the halfbrite technique.

Normally the Amiga can only display up to 32 colors at one time. This is because of two factors: 1.) the number of available colors is determined by the number of bit-planes (5, 2^5=32) and 2). The Amiga has 32 color registers to store color values that were defined using the AmigaBASIC PALETTE statement.

When the halfbrite mode is active, you have six bit-planes available and the number of available colors increases accordingly from 2^5=32 to 2^6=64. However, since there are only 32 color registers, there isn't space for the other 32 colors. To solve this problem, we use the 32 registers twice. We obtain colors 0 to 31 directly from registers 0 to 31. Colors 32 to 63 are also obtained from registers 0 to 31 but the RGB values in the registers are shifted one bit to the right.

However this causes three effects: 1.) the 32 extra halfbrite colors cannot be freely defined because they are dependent on the values of the first 32 colors, 2.) the extra colors are copies of the existing colors but are darker (therefore the name halfbrite), and 3.) if the first 32 colors are very dark, there will be little visible difference between them and the second 32 colors.

Although these limits may sound troublesome, the halfbrite mode is still rewarding because you can have an extra 32 slightly darker variations of your first 32 colors to work with.

Since we cannot normally activate the halfbrite mode with BASIC, we are going to create a screen with a depth of five (5 bit-planes). From the information in the previous chapters, we know the Amiga graphics system fairly well by now. Therefore, it should not be difficult to set up a sixth bit-plane in the bit-map structure of the screen. Finally, set the halfbrite flag in the ViewPort. (There is just one small problem that we will cover shortly.)

To make our screen a reality, we have to access two system libraries, exec and Intuition. We need the following functions:

```
RemakeDisplay()
AllocMem()
FreeMem()
```

Our program, the halfbrite activator, follows. In addition to the demonstration program, there are two SUB programs, `HalfBriteOn` and `HalfBriteOff`. Neither SUB requires any arguments. The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 4.2.1¶
'# Program: Halfbrite Activator¶
'# Date:    01/17/87¶
'# Author:  tob¶
'# Version: 1.1¶
'#¶
'########################################¶
¶
' Activates the Amigas special graphic mode "Halfbrite"¶
' not normally available in BASIC. With 6 bit-planes¶
' there are a total of 64 different colors available.¶
' We will explain the functions and the most effective¶
' programming methods for this mode in this book. NOTE:¶
' This mode only functions in LoRes (Low Resolution).¶
¶
PRINT "Searching for .bmap file..."¶
¶
'EXEC-Library¶
DECLARE FUNCTION AllocMem& LIBRARY¶
'FreeMem¶
'INTUITION-Library¶
'RemakeDisplay()¶
¶
LIBRARY "intuition.library"¶
LIBRARY "exec.library"¶
¶
main:  '* Open a screen with a depth of 5¶
        loRes             = 1¶
        screen.nr%        = 1¶
        screen.x%         = 320¶
        screen.y%         = 200¶
        screen.depth%     = 5 '5 Planes required¶
        screen.resolution% = loRes¶
        SCREEN screen.nr%,screen.x%,screen.y%,
screen.depth%,screen.resolution%¶
¶
       '* Open a window in the new screen¶
        windo.nr%   = 1¶
        windo.name$ = "Halfbrite!"¶
        WINDOW windo.nr%,windo.name$,,,screen.nr%¶
¶
```

```
demo:  '* Activate HalfBrite¶
       HalfBriteOn¶
¶
       PRINT TAB(10);"The HalfBrite Mode!"¶
¶
       '* The original colors...¶
       LOCATE 3,2:COLOR 1,0¶
¶
       PRINT "A "; ¶
       FOR loop%=0 TO 31¶
         COLOR 0,loop%¶
         PRINT " ";¶
       NEXT loop%¶
¶
       '* ...and the HalfBrite Colors!¶
       LOCATE 4,2:COLOR 1,0¶
¶
       PRINT "B ";¶
       FOR loop%=32 TO 63¶
         COLOR 0,loop%¶
         PRINT " ";¶
       NEXT loop%¶
¶
       LINE   (22,15)-(280,32),1,b¶
       LOCATE 7,2:COLOR 1,0¶
       PRINT "A: The 32 original colors, stored"¶
       PRINT "   in the hardware Color Registers."¶
¶
       LOCATE 10,2¶
       PRINT "B: The additional 32 HalfBrite"¶
       PRINT "   colors, corresponding to the"¶
       PRINT "   original colors at half intensity.""¶
¶
       LOCATE 14,2¶
       PRINT   "The blinking sample shows when the"¶
       PRINT "  color register of the original color"¶
       PRINT "  is changed, the HalfBrite color is"¶
       PRINT "  changed accordingly.""¶
¶
       LOCATE 19,4¶
       PRINT "Press the left mouse button"¶
¶
       WHILE check% = 0¶
         check% = MOUSE(0)¶
         PALETTE 30,.7,.2,.9¶
         FOR t  = 1 TO 500:NEXT t¶
         PALETTE 30,.3,.8,.1¶
         FOR t  = 1 TO 500:NEXT t¶
       WEND¶
¶
       FOR loop% = 0 TO 31¶
         COLOR  loop%,loop%+32¶
         LOCATE 20,1¶
         PRINT "TEST COLOR ";loop%¶
         PRINT "Text Color       - Original Color "¶
         PRINT "Background Color - HalfBrite Color"¶
```

```
          FOR t = 1 TO 500:NEXT t¶
        NEXT loop%¶
        CLS  ¶
        COLOR 1,0¶
¶
endprog:  '* HalfBrite off and close SCREEN¶
        HalfBriteOff¶
        WINDOW windo.nr%,windo.name$,,,-1¶
        SCREEN CLOSE screen.nr%¶
        PRINT "End of DEMO!"¶
        LIBRARY CLOSE¶
        END¶
¶
SUB HalfBriteOn STATIC¶
      SHARED screen.mode%¶
      SHARED screen.viewport&¶
¶
     '* Define variables¶
      MEM.CHIP  = 2^1¶
      MEM.CLEAR = 2^16¶
      memory.option&   = MEM.CHIP+MEM.CLEAR¶
      window.base&     = WINDOW(7)¶
      screen.base&     = PEEKL(window.base&+46)¶
      screen.bitmap&   = screen.base&+184¶
      screen.viewport& = screen.base&+44¶
      screen.rastport& = screen.base&+84¶
      screen.width%    = PEEKW(screen.bitmap&)¶
      screen.height%   = PEEKW(screen.bitmap&+2)¶
      screen.size&     = screen.width%*screen.height%¶
      screen.depth%    = PEEK(screen.bitmap&+5)¶
      screen.mode%     = PEEKW(screen.viewport&+32)¶
¶
     '* SCREEN already has 6 BitPlanes?¶
      IF screen.depth%>5 THEN screen.depth%=2^8¶
¶
     '* add missing Bitplanes¶
      FOR loop1%=screen.depth%+1 TO 6¶
        plane&(loop1%)    = AllocMem&(screen.size&,
memory.option&)¶
        IF plane&(loop1%) = 0 THEN¶
          FOR loop2% = screen.depth%+1 TO loop1%-1¶
            CALL FreeMem(plane&(loop2%),screen.size&)¶
          NEXT loop2%¶
          ERROR 7¶
        END IF¶
        POKEL screen.bitmap&+4+4*loop1%,plane&(loop1%)¶
      NEXT loop1%¶
      POKE screen.bitmap&+5,6¶
¶
     '* HalfBrite On¶
      POKEW screen.viewport&+32,(screen.mode% OR 2^7)¶
      CALL  RemakeDisplay¶
END SUB¶
¶
SUB HalfBriteOff STATIC¶
      SHARED screen.mode%¶
```

```
      SHARED screen.viewport&¶
¶
     '* Reset HalfBrite Flag¶
      POKEW screen.viewport&+32,screen.mode%¶
      CALL  RemakeDisplay¶
END SUB¶
```

Working with the halfbrite program:

After you call the SUB "HalfBriteOn" you will have 64 different colors available. You can freely define the first 32 colors using the BASIC PALETTE statement:

```
PALETTE register,red,green,blue
```

register:	0-31
red, green, blue:	0.0 - 1.0

Colors 32 to 63 are defined at the same time, except that they are half as bright.

With the COLOR statement you can select any color from 0 to 63, draw, print text and fill. But you should remember that for AmigaBASIC, the screen still has only five bit-planes. When BASIC scrolls the screen (when you write text in the last row) only five planes will scroll; the sixth plane stays in place. To avoid this problem do not print to the last screen line.

When you no longer need the halfbrite mode you can deactivate it by using the SUB "HalfBriteOff".

At the beginning of this section we mentioned a problem involving changes to the halfbrite flag in the ViewPort. Setting this flag does not do anything. It doesn't matter what type of manipulation you perform in the ViewPort, nothing is changed in the display.

This happens because the display is only changed by the hardware registers. Since ViewPort is a data block in RAM and not a hardware register, the display isn't changed. In order to affect the display, the information in ViewPort about how the display is formed must first be sent to the Copper. This is because the Copper controls and programs the hardware.

We make ViewPort changes effective when we call the Intuition function "ReMakeDisplay". This function creates a new Copper list and reflects the change in the ViewPort structure. Finally this list is sent to the Copper.

4.2.2 The hold-and-modify mode: 4096 colors

The hold-and-modify mode (abbr. HAM) is also not supported by AmigaBASIC. You cannot access it by using the SCREEN statement.

We can activate this mode for a screen that already exists. Before we do this, let's take a look at the principles for using this mode.

When the HAM mode is active, you can display up to 4096 colors at the same time. If we followed the normal rules for displaying 4096 colors, we would require 12 bit-planes. This would require a large amount of memory (1 bit-plane = 64,000 bytes in lo-res, 12 bit-planes = 768,000 bytes!). Also, the Amiga's DMA (Direct Memory Access) is not fast enough to retrieve and build a new screen every 1/60 second from 12 different RAM areas. Obviously, we need a special procedure.

Actually, HAM works with only six bit-planes, just as the halfbrite mode. The first 16 colors are the exact colors that were defined for the first 16 color registers. All other colors are determined by the HAM principle. They use the color of the pixel to the left and modify the RGB value.

Before we undertake the complex arrangement of a HAM graphic, we need to activate the mode. This is very similar to the halfbrite mode. We create a sixth bit-plane, add it to the bit-map and then set the HAM flag in the ViewPort. A call to RemakeDisplay switches the display to HAM. Again there are two SUB programss, HAMOn and HAMOff.

The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 4.2.2     ¶
'# Program: HAM Activator¶
'# Date:    02/16/87¶
'# Author:  tob¶
'# Version: 1.4¶
'#¶
'########################################¶
¶
' Activates the Amiga Special Graphic Mode "HAM" (Hold¶
' And Modify) not normally available to BASIC. Provides ¶
' up to 4096 colors at the same time (with 6 .¶
' bit-planes) NOTE: This Mode only functions in LoRes¶
' (Low Resolution) Displays.¶
¶
```

```
PRINT "Searching for .bmap file..."¶
¶
'EXEC-Library¶
DECLARE FUNCTION AllocMem& LIBRARY¶
'FreeMem()¶
¶
'INTUITION-Library¶
'RemakeDisplay()¶
¶
LIBRARY "intuition.library"¶
LIBRARY "exec.library"¶
¶
main:  '* Open a screen with a depth of 5¶
        loRes              = 1      ¶
        screen.nr%         = 1¶
        screen.x%          = 320¶
        screen.y%          = 200¶
        screen.depth%      = 5 '5 Planes noetig¶
        screen.resolution% = loRes¶
        SCREEN screen.nr%,screen.x%,screen.y%,
screen.depth%,screen.resolution%¶
        ¶
       '* Open a window in the new screen¶
        windo.nr%   = 1¶
        windo.name$ = "HAM! 4096 Colors available!"¶
        WINDOW windo.nr%,windo.name$,,,screen.nr%¶
        ¶
demo:  '* Activate HAM¶
        HAMOn¶
        ¶
        PRINT TAB(7) "256 of 4096 Colors"¶
        ¶
        s = 10              'Box size¶
        x = 40              'Position of upper¶
        y = 20              'left corner of Demo¶
        PALETTE 3,0,0,0     'Frame Color¶
        PALETTE 4,.5,0,.5   'dark purple¶
        PALETTE 5,1,0,1     'light purple¶
        PALETTE 6,1,0,0     'light red¶
        PALETTE 7,0,0,1     'light blue¶
        ¶
       '* Set Orientation Marks¶
        LINE (5,y)-(5+8,y+8),4,bf¶
        LINE (240,y)-(240+8,y+8),7,bf¶
        LINE (5,166)-(5+8,166+8),6,bf¶
        LINE (240,166)-(240+8,166+8),5,bf¶
        ¶
       '* Draw Frame¶
        LINE (x-1,y-1)-(x+17*s+1,y+16*s+1),3,b¶
        ¶
       '* Draw first 256 HAM Colors¶
        FOR loop% = 0 TO 15¶
          LINE (x,loop%*s+y)-(s+x,loop%*s+s+y),
32+loop%,bf¶
          FOR loop2% = 0 TO 15¶
```

```
            LINE (s+loop2%*s+x,loop%*s+y)-
(2*s+loop2%*s+x,loop%*s+s+y),loop2%+16,bf¶
          NEXT loop2%¶
        NEXT loop%¶
        ¶
       '* Raise Green level¶
        FOR loop2% = 0 TO 15¶
          PALETTE 3,0,loop2%*(1/15),0¶
          LOCATE  10,28¶
          PRINT   "Green Level:"¶
          PRINT   TAB(31) loop2%¶
          FOR t = 1 TO 3000:NEXT t¶
        NEXT loop2%¶
        ¶
        LOCATE 2,7¶
        PRINT "Please Press a Key!"¶
        WHILE INKEY$="":WEND¶
        ¶
endprog:¶
       '* HAM off and close screen¶
        HAMOff¶
        WINDOW windo.nr%,windo.name$,,,-1¶
        SCREEN CLOSE screen.nr%¶
        PRINT "End of DEMO!"¶
        LIBRARY CLOSE¶
        END¶
          ¶
SUB HAMOn STATIC¶
      SHARED screen.modus%¶
      SHARED screen.viewport&¶
¶
     '* Define Variables¶
      MEM.CHIP  = 2^1¶
      MEM.CLEAR = 2^16¶
      memory.option&   = MEM.CHIP+MEM.CLEAR¶
      window.base&     = WINDOW(7)¶
      screen.base&     = PEEKL(window.base&+46)¶
      screen.bitmap&   = screen.base&+184¶
      screen.viewport& = screen.base&+44¶
      screen.rastport& = screen.base&+84¶
      screen.width%    = PEEKW(screen.bitmap&)¶
      screen.height%   = PEEKW(screen.bitmap&+2)¶
      screen.size&     = screen.width%*screen.height%¶
      screen.depth%    = PEEK(screen.bitmap&+5)¶
      screen.mode%     = PEEKW(screen.viewport&+32)¶
     ¶
     '* SCREEN has 6 BitPlanes already?¶
      IF screen.depth%>5 THEN screen.depth%=2^8¶
¶
     '* Build missing BitPlanes¶
      FOR loop1% = screen.depth%+1 TO 6¶
        plane&(loop1%)    =
AllocMem&(screen.size&,memory.option&)¶
        IF plane&(loop1%) = 0 THEN¶
          FOR loop2% = screen.depth%+1 TO loop1%-1¶
            CALL FreeMem(plane&(loop2%),screen.size&)¶
```

```
          NEXT loop2%¶
          ERROR 7¶
        END IF¶
        POKEL screen.bitmap&+4+4*loop1%,plane&(loop1%)
¶
      NEXT loop1%¶
      POKE screen.bitmap&+5,6¶
      ¶
     '* HAM On¶
      POKEW screen.viewport&+32,(screen.mode% OR 2^11)¶
      CALL RemakeDisplay¶
END SUB¶
¶
SUB HAMoff STATIC¶
      SHARED screen.mode%¶
      SHARED screen.viewport&¶
      ¶
     '* Reset Flag¶
      POKEW screen.viewport&+32,screen.mode%¶
      CALL RemakeDisplay¶
END SUB¶
```

After you start this program you will see a field of 256 colors that were created by using only red and blue. In the top left hand corner is a dark purple color and in the lower right hand corner, a light purple color. Light red is in the lower left hand corner and light blue is in the upper right hand corner.

Now we blend a green slowly and evenly with the other colors to display all 4096.

This mass of colors looks very impressive but programming them is not easy. But, as you will see, it is not too complicated.

First, let's define the difference between real colors and HAM colors. Real colors, the colors 0-15, actually display the values in color registers 0-15. These colors are permanent and can only be changed with the PALETTE statement. The HAM colors 16-63 are different because they are always affected by their neighboring color to the left. A HAM color takes the color of the pixel to the left and modifies the RGB components. Which of the three components that is changed depends on the HAM color itself.

Color	0	-	15	Real color
Color	16+0	-	16+15	HAM type 1
Color	32+0	-	32+15	HAM type 2
Color	48+0	-	48+15	HAM type 3

HAM color type 1 takes the neighboring color and changes the blue component. The blue component of the HAM color corresponds to a

value higher than 16. The HAM color 16+12=28 uses the neighboring color and makes a value of 12 for blue.

HAM color type 2 takes the neighboring color and modifies the red component. The HAM color 32+8=40 takes the neighboring color and uses a value of 8 in the red component.

HAM color type 3 performs exactly the same function for colors 48 and up, changing the green component.

In our example program we created a black frame making red, green and blue =0. Directly to the right of the frame we drew with a HAM color of type 2. This color checks the pixel color to the left, the black frame and uses the same color. Red, green and blue are still zero. The blue field is set by the HAM color itself. This blue value increases in a loop that changes every screen row.

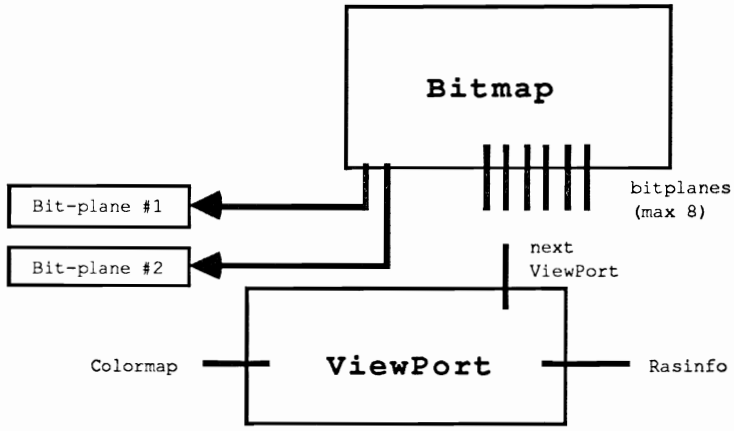
Directly to the right of this HAM color we draw 16 HAM colors of type 1. These add in a red value of one.

This creates the color pattern. The red intensity increases to the right and the blue intensity increases downward.

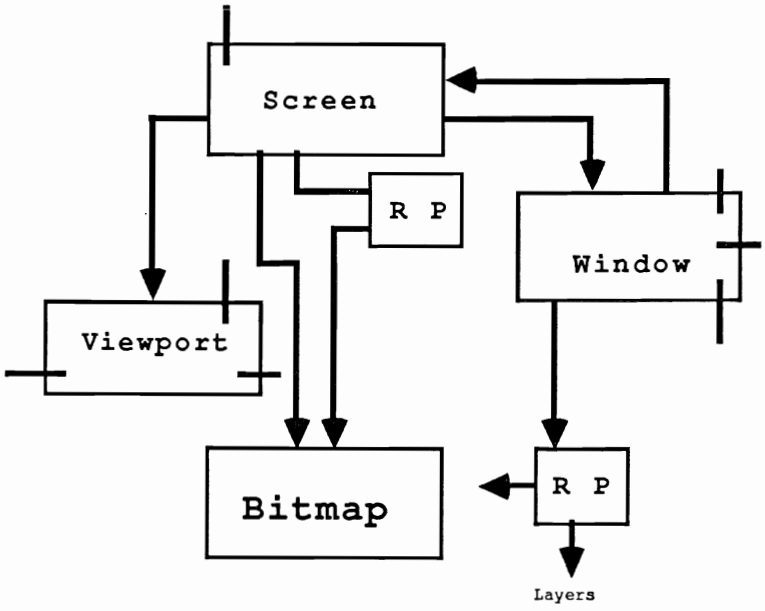
Now we change the color of the frame by making the previously black frame greener with the `PALETTE` command. The green value of the frame passes immediately to the HAM colors. The entire color graphic effect results from an increasing green intensity.

4.3 The ViewPort in the system

We now have a clearer picture of the Amiga system. You should be familiar with two components, the Bit-map and the ViewPort. Both are easily illustrated:



Now the system connection with these components:



The most elemental part of a display, the ViewPort, is controlled by Intuition with the help of a screen. The screen and window are displayed through a private RastPort in a shared video RAM, the bit-planes.

We still have not explained the Layer, Colormap and RasInfo components.

Before we continue our study of the graphics system, we have one more point to discuss. There is an additional data structure, called *View*, that seems to appear out of nowhere.

4.4 View: the graphic brain

It is not surprising that the address of View, or even its name, has not appeared yet. View is the contact point between the graphic software and the graphic hardware of your Amiga. It is where everything begins. You can obtain the address of View by calling an Intuition function named `ViewAddress`.

```
DECLARE FUNCTION ViewAddress& LIBRARY
LIBRARY "Intuition.library"

view&=ViewAddress&
```

Note: Remember that the Intuition routine `ViewPortAddress` provides you with the address of the ViewPort where your actual output window is located:

```
DECLARE FUNCTION ViewPortAddress& LIBRARY
LIBRARY "Intuition.library"

vp&=ViewPortAddress&(WINDOW(7))
```

Here is the View data structure:

Data Structure View/graphics/18 Bytes

Offset	Type	Description
+ 000	Long	Pointer to first ViewPort
+ 004	Long	LongFrame Copper list
+ 008	Long	ShortFrame Copper list
+ 012	Word	DyOffset
+ 014	Word	DxOffset
+ 016	Word	Mode

Although View doesn't seem like an essential part of the graphics system, the entire display (including all screens) is dependent on it. A detailed description of the data fields follows:

Offset 0: Next ViewPort

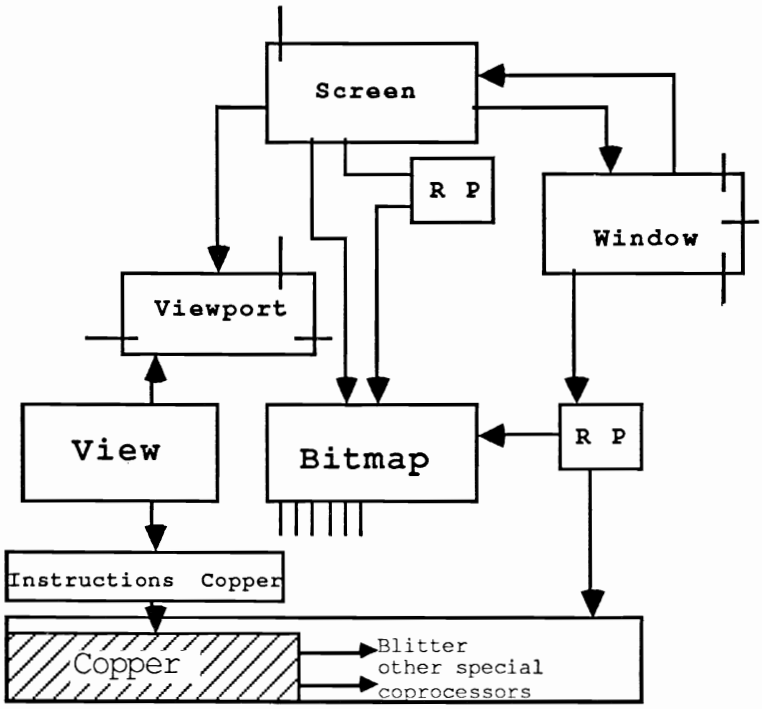
This offset contains the address pointer to the first ViewPort structure of the display. From that structure you can obtain the addresses of further ViewPorts, if there are any.

Offset 4 and 8: Copper lists

We have dealt with Copper lists before when discussing the ViewPorts. Just as the Copper list of the ViewPort is responsible for the drawing region of the ViewPort, this Copper list controls the entire display (or in other words, all the ViewPorts). A normal display requires only the LongFrame list. The second Copper list is only necessary when using the interlace mode.

The rest of the fields are self explanatory since they have exactly the same functions as the same named fields in the ViewPort (see Chapter 4).

Now that we've added View to our explanation of the graphics system, we have covered the most important components. The connection between hardware and Intuition is complete.



Before we continue with the graphic system, we need to test our model of the system. Since we have now advanced far enough, we can create our own display. There are several functions from the graphic libraries that we need.

```
InitView()
InitVPort()
GetColorMap()
InitBitMap()
AllocRaster()
LoadRGB4()
MakeVPort()
MrgCop()
LoadView()
FreeRaster()
FreeColorMap()
FreeVPortCopLists()
FreeCprList()
```

In the following program we are going to demonstrate all the steps required to create a simple display using a ViewPort. Use our program as a model that shows the correct programming methods. Since Copper programming is the subject of the next chapter, this program is a good practice exercise.

The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 4.4¶
'# Program: Graphic Primitive Display¶
'# Date:    01/01/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' Demonstrates the creation of a graphic display on¶
' the Amiga using the "Graphic Primitives", the base¶
' commands of the Graphic Library. This screen is HiRes¶
' (High Resolution) with one bit-plane (depth = 1).¶
' The contents of the first bit-plane of this screen¶
' is copied to the new display.¶
¶
PRINT "Searching for .bmap file..."¶
¶
'GRAPHICS-Library¶
DECLARE FUNCTION AllocRaster& LIBRARY¶
DECLARE FUNCTION GetColorMap& LIBRARY¶
```

```
'FreeRaster()¶
'FreeColorMap()¶
'FreeVPortCopLists()¶
'FreeCprList()¶
'InitView()¶
'InitVPort()¶
'InitBitMap()¶
'LoadRGB4()¶
'MakeVPort()¶
'MrgCop()¶
'LoadView()¶
¶
'EXEC-Library¶
DECLARE FUNCTION AllocMem& LIBRARY¶
'FreeMem()¶
¶
'INTUITION-Library¶
DECLARE FUNCTION ViewAddress& LIBRARY¶
¶
LIBRARY "exec.library"¶
LIBRARY "graphics.library"¶
LIBRARY "intuition.library"¶
¶
init:     '* Define Screen Parameters¶
           wide%        = 640¶
           height%      = 200¶
           depth%       = 1¶
           o.bitplane1& = PEEKL(PEEKL(WINDOW(8)+4)+8)¶
           ¶
          '* Store our View Pointer to enable¶
          '* us to return to it later¶
           oldview&     = ViewAddress&¶
          '* Reserve memory for required structures¶
          '* View - Brain of Displays¶
           view&        = 18¶
           GetMemory view&¶
          '* ViewPort - our Screen¶
           viewport&    = 40¶
           GetMemory viewport&¶
          '* BitMap - Manager of Bit-Planes¶
           bitmap&      = 40¶
           GetMemory bitmap&¶
          '* RasInfo - Information for the ViewPort¶
           RasInfo&     = 12¶
           GetMemory RasInfo&¶
           ¶
          '* Prepare View and ViewPort for use¶
           CALL InitView(view&)¶
           CALL InitVPort(viewport&)¶
           ¶
          '* Hires¶
           hires& = &H8000¶
           POKEW viewport&+32,hires&¶
            ¶
          '* Place ViewPort in View¶
           POKEL view&,viewport&¶
```

```
 ¶
'* Setup Color Table¶
 colorMap&     = GetColorMap&(2)¶
 IF colorMap& = 0 THEN ERROR 7¶
¶
'* Fill ViewPort with our parameters¶
 POKEW viewport&+24,wide%¶
 POKEW viewport&+26,height%¶
¶
'* Place RasInfo in ViewPort¶
 POKEL viewport&+36,RasInfo&¶
 ¶
'* Place Color Table in the ViewPort¶
 POKEL viewport&+4,colorMap&¶
 ¶
'* Fill BitMap Structure with our parameters¶
 CALL InitBitMap(bitmap&,depth%,wide%,height%)¶
 ¶
'* Get a BitPlane¶
 plane&     = AllocRaster&(wide%,height%)¶
 IF plane& = 0 THEN ERROR 7¶
¶
'* Place BitPlane in BitMap¶
 POKEL bitmap&+8,plane&¶
 ¶
'* Place BitMap in RasInfo¶
 POKEL RasInfo&+4,bitmap&¶
 ¶
'* Define Colors¶
 red$         = CHR$(15)+CHR$(0)¶
 black$       = CHR$(0)+CHR$(0)¶
 colortable$ = red$+black$¶
 ¶
'* Load Colors into Color Table¶
 CALL LoadRGB4(viewport&,SADD(colortable$),2)¶
 ¶
'* Construct Copper Instruction List¶
 CALL MakeVPort(view&,viewport&)¶
 CALL MrgCop(view&)¶
 ¶
'* Load New Display into Copper¶
 CALL LoadView(view&)¶
 ¶
'* Play with the Display¶
 BEEP¶
 size& = wide%*height%/8¶
 ¶
 FOR loop& = 0 TO size&-1¶
   POKE plane&+loop&,PEEK(o.bitplane1&+loop&)¶
 NEXT loop&¶
 BEEP¶
 ¶
'* Restore old Copper List¶
 CALL LoadView(oldview&)¶
¶
'* Cleanup: Return Memory for Bit-Plane¶
```

```
            CALL FreeRaster(plane&,wide%,height%)¶
           '* Release Color Table¶
            CALL FreeColorMap(colorMap&)¶
           '* Release interim ViewPorts Lists¶
            CALL FreeVPortCopLists(viewport&)¶
           '* Release Copper Instruction List¶
            copperlist& = PEEKL(view&+4)¶
            CALL FreeCprList(copperlist&)¶
           '* Release Structure Memory¶
            FreeMemory view&¶
            FreeMemory viewport&¶
            FreeMemory RasInfo&¶
            FreeMemory bitmap&¶
            ¶
           '* And that's it!¶
            LIBRARY CLOSE¶
            END¶
            ¶
SUB GetMemory(size&) STATIC¶
        opt&      = 2^0+2^1+2^16¶
        RealSize& = size&+4¶
        size&     = AllocMem&(RealSize&,opt&)¶
        IF size&  = 0 THEN ERROR 255¶
        POKEL size&,RealSize&¶
        size&     = size&+4¶
END SUB¶
¶
SUB FreeMemory(add&) STATIC¶
        add&      = add&-4¶
        RealSize& = PEEKL(add&)¶
        CALL FreeMem(add&,RealSize&)¶
END SUB¶
```

The program: The first step in our program is to choose what kind of display we want to create (how wide and how high). Our choice is a hi-res screen with a standard resolution of 640*200 pixels and a depth of one bit-plane.

To be able to return to our original display we must store the addresses of our structure in several variables. The Intuition function `ViewAddress` provides the required pointers.

To create our display we use the following structures:

View (18 Bytes)
ViewPort (40 Bytes)
Bitmap (40 Bytes)
RasInfo (12 Bytes)

The View structure forms the brain of our future display. There is only one active View and from this View you can have any number of ViewPort branches.

View and ViewPort must be created ready to use. `InitView` fills the View structure with the standard values, which automatically sets the View structure to appear about a half inch from the edge of the screen. `InitVPort` does the same thing with the ViewPort. It is normally set for lo-res and the pointer for the next ViewPort is set to zero because usually there are no additional ViewPorts.

Now we have to make a connection between View and ViewPort. To do this, we use the first field of the View structure to store the address of the first (and only) ViewPort structure.

Next we must have a color table that will later be used by our screen. `GetColorMap` will take care of this.

Then we store the resolution for our ViewPort and the RasInfo block is placed in the ViewPort. Now we make the Bit-map structure ready for use by using `InitBitMap()`. We write the address of our bit-plane into the Bit-map structure.

Next we write the address of the bit-map into the RasInfo structure. Then we use `LoadRGB4` to store the colors to the ViewPort.

Now that all the required data has been stored, our display is ready to use. The Amiga creates the instructions for the graphic processor from our information with the following steps: 1.) the function `MakeVPort` creates the Copper lists from the data in the ViewPorts and writes the pointer for this list into the ViewPort, and 2.) the function `MrgCop` integrates the instructions of our ViewPort with those of all the displays (we have only one ViewPort).

The completed Copper list is stored in View. A list of Copper commands has been created from the data and will be used for our display. We just have to send the commands to the Copper and our new display will appear. This action is performed by `LoadView` and immediately we see our bright red display.

To show you that this is a fully functional display, we copy the first bit-plane of the Workbench screen to it. This takes a few moments.

Everything worked, but now we want to get back to the original display. Since we stored the address for our old View, we shouldn't have any problems. We can use `LoadView` to send the old Copper lists to the Copper and return to the original.

Although the demonstration is over, we still need to "cleanup" because the display has used up a lot of memory that we will want to release again.

4.5 Copper programming

The Copper has just demonstrated how powerful it is. We are going to take advantage of that power with our next program, which will use a technique known as double buffering.

4.5.1 Double buffering for lightening fast graphics

The drawing speed of the Amiga does not affect the Copper because the Copper operates independently at full speed. Because of this you can amaze the users of your programs with lightening fast graphics created with double buffering. To create this effect, show the user a display on which nothing is happening. While the user stares at this boring display, build your graphics in a second invisible display. When your graphics are complete, switch on the second display and your graphics will instantly appear on the screen.

We will now explain how this works. The pointer to the Copper lists of the old display are read from View and are stored. The pointer in View is erased. Now a new bit-map with new bit-planes is prepared for the second display. `MakeVPort` and `MrgCop` are used to generate the new Copper lists, which are then stored. To switch from one display to the other we just write the new pointer to the View structure and use `LoadView` to activate it.

Again we have designed a small program package. It consists of the following SUB programs:

```
MakeDoubleBuffer
DoubleBufferOn
DoubleBufferOff
AbortDoubleBuffer
transmit
```

The ¶ characters in the program listing for double buffering signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 4.5.1 ¶
'# Program: Double Buffered Display¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' This program creates a second screen, that¶
' works as a backup buffer for normal screen¶
¶
PRINT "Searching for .bmap file..."¶
¶
'GRAPHICS-Library¶
DECLARE FUNCTION BltBitMap& LIBRARY¶
DECLARE FUNCTION AllocRaster& LIBRARY¶
'FreeRaster()¶
'MakeVPort()¶
'MrgCop()¶
'LoadView()¶
'FreeCprList()¶
¶
'EXEC-Library¶
DECLARE FUNCTION AllocMem& LIBRARY¶
'FreeMem()¶
'CopyMem()¶
¶
'INTUITION-Library¶
DECLARE FUNCTION ViewPortAddress& LIBRARY¶
DECLARE FUNCTION ViewAddress& LIBRARY¶
¶
LIBRARY "intuition.library"¶
LIBRARY "graphics.library"¶
LIBRARY "exec.library"¶
¶
init:       CLS¶
            PRINT "WITHOUT DOUBLE BUFFERING!"¶
            FOR t=1 TO 20¶
              PRINT STRING$(80,"+")¶
            NEXT t¶
            FOR t=1 TO 20¶
              x% = RND(1)*600¶
              y% = RND(1)*150¶
              r% = RND(1)*100¶
              CIRCLE (x%,y%),r%¶
            NEXT t¶
            CLS¶
            PRINT "AND NOW WITH DOUBLE BUFFERING!!!"¶
            MakeDoubleBuffer¶
            DoubleBufferOn¶
            FOR t=1 TO 20¶
              PRINT STRING$(80,"+")¶
            NEXT t¶
            transmit¶
            LOCATE 1,1¶
```

```
                FOR t=1 TO 20¶
                  x% = RND(1)*600¶
                  y% = RND(1)*150¶
                  r% = RND(1)*100¶
                  CIRCLE (x%,y%),r%¶
                NEXT t¶
                transmit¶
                LOCATE 5,10¶
                LINE (38,29)-(442,67),3,b¶
                PRINT "Even this works!"¶
                PRINT TAB(10);"Double Buffering = Backup-
Display"¶
                PRINT TAB(10);"These are two separate screens
that"¶
                PRINT TAB(10);"are switched back and forth!"¶
                FOR loop%=1 TO 15¶
                  DoubleBufferOn¶
                  FOR t=1 TO 1000:NEXT t¶
                  DoubleBufferOff¶
                  FOR t=1 TO 1000:NEXT t¶
                NEXT loop%  ¶
                PRINT ¶
                PRINT "PRESS LEFT MOUSE BUTTON!"¶
                SLEEP:SLEEP          ¶
                AbortDoubleBuffer¶
                LIBRARY CLOSE¶
                END¶
                ¶
¶
SUB MakeDoubleBuffer STATIC¶
     '* Create second display¶
      SHARED TargetBitmap&,rasInfo&,SourceBitMap&,view&¶
      SHARED bufferx%,buffery%,vp&¶
      SHARED home1&,home2&,guest1&,guest2&¶
      view&        = ViewAddress&¶
      vp&          = ViewPortAddress&(WINDOW(7))¶
      rasInfo&     = PEEKL(vp&+36)¶
      SourceBitMap& = PEEKL(rasInfo&+4)¶
      opt&         = 2^0+2^1+2^16¶
      TargetBitmap&=AllocMem&(40,opt&)¶
      ¶
     '* Copy BitMaps¶
      IF  TargetBitmap& = 0 THEN ERROR 7¶
     '* NOTE: FOR KICKSTART VERSION 1.2 AND ABOVE¶
     '* FOR 1.0 UND 1.1 USE LINES BELOW¶
     '* ¶
     '* FOR loop&=0 to 40 STEP 4¶
     '*   POKEL TargetBitMap&+loop&,PEEKL(SourceBitMap&+
loop&)¶
     '* NEXT loop&¶
       ¶
      CALL CopyMem(SourceBitMap&,TargetBitmap&,40)¶
¶
     '* Get Planes¶
      bufferx%  = PEEKW(SourceBitMap&)*8¶
      buffery%  = PEEKW(SourceBitMap&+2)¶
```

```
      depth%    = PEEK(SourceBitMap&+5)¶
      FOR loop% = 0 TO depth%-1¶
        plane&(loop%)    =
AllocRaster&(bufferx%,buffery%)¶
        IF plane&(loop%) = 0 THEN ERROR 7¶
        POKEL TargetBitmap&+8+loop%*4,plane&(loop%)¶
      NEXT loop%¶
      ¶
     '* Copy active display to Buffer¶
      plc% = BltBitMap&(SourceBitMap&,0,0,TargetBitmap&,
0,0,bufferx%,buffery%,200,255,0)¶
      IF plc%<>depth% THEN ERROR 17¶
      ¶
     '* Store Original Copper List¶
      home1& = PEEKL(view&+4)¶
      home2& = PEEKL(view&+8)¶
      ¶
     '* Generate Second Copper List¶
      POKEL view&+4,0¶
      POKEL view&+8,0¶
      POKEL rasInfo&+4,TargetBitmap&¶
      CALL  MakeVPort(view&,vp&)¶
      CALL  MrgCop(view&)¶
      CALL  LoadView(view&)¶
      guest1& = PEEKL(view&+4)¶
      guest2& = PEEKL(view&+8)¶
      ¶
     '* Reset ¶
      POKEL rasInfo&+4,SourceBitMap&¶
      POKEL view&+4,home1&¶
      POKEL view&+8,home2&¶
      CALL  LoadView(view&)¶
      ¶
END SUB¶
¶
SUB DoubleBufferOn STATIC¶
     '* Activate New Copper List¶
      SHARED view&,guest1&,guest2&¶
      SHARED rasInfo&,TargetBitmap&¶
      POKEL  view&+4,guest1&¶
      POKEL  view&+8,guest2&¶
      CALL   LoadView(view&)¶
END SUB¶
¶
SUB DoubleBufferOff STATIC¶
     '* Activate Old Copper List¶
      SHARED view&,home1&,home2&¶
      SHARED rasInfo&,SourceBitMap&¶
      POKEL  view&+4,home1&¶
      POKEL  view&+8,home2&¶
      CALL   LoadView(view&)¶
END SUB           ¶
      ¶
SUB transmit STATIC¶
     '* Copy Old Display to the new Buffer¶
```

```
      SHARED SourceBitMap&,TargetBitmap&,bufferx%,
buffery%¶
      plc% = BltBitMap&(SourceBitMap&,0,0,TargetBitmap&,0
,0,bufferx%,buffery%,200,255,0)¶
END SUB¶
¶
SUB AbortDoubleBuffer STATIC¶
      SHARED rasInfo&,view&,TargetBitmap&¶
      SHARED vp&,bufferx%,buffery%¶
      SHARED home1&,home2&,guest1&,guest2&¶
      ¶
     '* Restore Old Display and VPort Copper Lists¶
      POKEL view&+4,home1&¶
      POKEL view&+8,home2&¶
      CALL  MakeVPort(view&,vp&)¶
      CALL  MrgCop(view&)¶
      CALL  LoadView(view&)¶
      ¶
     '* Delete New VPort Copper Lists¶
      CALL  FreeCprList(guest1&)¶
      ¶
     '* Delete Second Copper List Set¶
      IF guest2&<>0 THEN CALL FreeCprList(guest2&)¶
      add& = TargetBitmap&+8¶
      pl&  = PEEKL(add&)¶
      ¶
     '* Delete BitPlanes and BitMap¶
      WHILE pl&<>0¶
        CALL FreeRaster(pl&,bufferx%,buffery%)¶
        add& = add&+4¶
        pl&  = PEEKL(add&)¶
      WEND¶
      CALL FreeMem(TargetBitmap&,40)¶
END SUB¶
```

Decsription: You switch the double buffer system on with the command:

```
MakeDoubleBuffer
```

The double buffer is used to create the invisible display. This command can only be used once. When you are ready to start, execute:

```
DoubleBufferOn
```

This command activates the hidden display. Your old display, where you are drawing, becomes invisible. Now you can take your time to create your graphics, no drawing can be seen on the screen.

As soon as your graphic is complete you only have to call:

```
transmit
```

to display the contents of the hidden display (in other words, to send your graphic to the visible screen). You can use the transmit command as often as desired.

When you want to quickly switch to an unbuffered display simply call:

```
DoubleBufferOff
```

All graphic and print commands will, from this point on, appear immediately on the screen. With `DoubleBufferOn`, you can activate the buffered system again.

Should you desire to leave the system entirely because your program is finished or you are tired of double buffered drawing, then use:

```
AbortDoubleBuffer
```

All memory areas used for the buffer displays are returned to the system.

4.5.2 Programming the Copper yourself

So far we have let the Amiga create the Copper instruction lists for us, from the data we provided. Another possibility is to program the Copper ourselves.

Before you do this, you need to understand the Copper functions. The Copper works very closely with the electronic beams of the display. These electronic beams scan from the upper left hand corner to the lower right hand corner of the screen sixty times a second.

The Copper is capable of waiting for this electronic beam to reach a specific position. This is handled by the `WAIT` instruction of the processor. The instruction requires a Y and X coordinate and tells the Copper to wait until the electronic beam has reached this coordinate. Until this occurs, the Copper will not process any more instructions.

The `MOVE` instruction allows the Copper to directly address hardware registers in the special purpose chips (the hardware registers are detailed in Appendix C). The `MOVE` instruction requires an offset for the hardware register and a value to store in the register.

The third and last instruction for the Copper is named `SKIP`. This instruction is used to actually skip past items in the Copper list.

Writing a Copper list for an entire display would be a very tedious job. Fortunately, this isn't necessary because most of the work can be accomplished easily by using MakeVPort. However, if you want to add your own Copper instructions to the Copper list for the displays, there is another method. In the structure of every ViewPort you will find a pointer for a user Copper list. This pointer is normally set to zero. When you want to integrate your own instructions with a display, create a stand alone Copper list with the desired commands. Then store the starting address of your list to the user Copper list pointer. Now you can continue as usual. MakeVPort links the user list to the display list of the ViewPort, MrgCop links this list to the entire list in View and LoadView activates the manipulated Copper lists.

We will now show you how to create your own Copper list. The next program contains four SUB programs for this purpose.

```
InitCop
ActiCop
WaitC
MoveC
```

First you need to create a data structure named UCopList. This structure requires 12 bytes of free memory which is reserved with InitCop.

We can program the user list with the commands MoveC and WaitC (Skip is not necessary for our application).

The call to the wait command looks like this:

```
WaitC y%,x%
```

The Y coordinate, which the Copper waits for, must be first. WaitC requires that this coordinate is first because it is easier to combine the Copper lists using MrgCop if they all have the same order.

MoveC can write any desired 16 bit value to a hardware register. In this chapter we use only a few of the many available registers. The complete register list is located in Appendix C. Here is the call:

```
MoveC register%, value%
```

register%:	Offset of the desired hardware register
value%:	16 bit value

Here is a selection of the most important hardware registers we are going to use:

Register	Meaning
384	Color Register 0 (Background color)
386	Color Register 1 (Drawing Color)
388	Color Register 2
(...)	
444	Color Register 30
446	Color Register 31

Now we can return to our user Copper list. After calling `InitCop`, you can add as many `MoveC`'s and `WaitC`'s as you like. However, you must make sure that your `WaitC`'s correspond with the screen coordinates you are calling. The top left hand corner of the display is at coordinate (0,0). This is where the electronic beam starts. Your `WaitC` coordinates must be in ascending X and Y coordinate sequence.

When your user Copper list is finished, it is linked to the existing display with `ActiCop`. Your assignments will then be executed by the Copper.

In our example program, we open our own screen. In order to return the memory used by our Copper instructions (including those in our reserved user list), we simply close the Intuition screen. Intuition automatically takes care of this. Do not attempt to create user instructions in the Workbench screen because when the instructions are executed, there will be no way to restore the normal display and release the assigned memory.

The following program is an example of Copper programming basics. The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 4.5.2¶
'# Program: Copper Raster Interrupt I¶
'# Date:    12/15/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' Demonstrates programming the Amiga graphic¶
' co-processor (Copper) from AmigaBASIC.¶
¶
PRINT "Searching for .bmap file..."¶
¶
'INTUITION-Library¶
DECLARE FUNCTION ViewAddress& LIBRARY¶
DECLARE FUNCTION ViewPortAddress& LIBRARY¶
```

```
'RethinkDisplay()¶
¶
'EXEC-Library¶
DECLARE FUNCTION AllocMem& LIBRARY¶
'FreeMem()¶
¶
'GRAPHICS-Library¶
'CWait()¶
'CMove()¶
'CBump()¶
¶
LIBRARY "intuition.library"¶
LIBRARY "graphics.library"¶
LIBRARY "exec.library"¶
¶
pre:        CLS¶
            SCREEN 1,640,200,2,2¶
            WINDOW 2,"COPPER!",(0,0)-(630,186),16,1¶
¶
            PRINT "Raster Interrupt through Copper-
Programming: A Split Screen!"¶
¶
init:       kolorregister% = 384¶
            red%           = 15 '0...15¶
            green%         =  4 '0...15¶
            blue%          =  4 '0...15¶
            kolorvalue%    = red%*2^8+green%*2^4+blue%¶
            yCoordinate%   = 100¶
            xCoordinate%   = 20¶
            ¶
main:       InitCop¶
            waitC yCoordinate%,xCoordinate%¶
            moveC kolorregister%,kolorvalue%¶
            ActiCop¶
            ¶
            PRINT "Press a Key!"¶
            WHILE INKEY$="":WEND¶
           ¶
            WINDOW CLOSE 2¶
            SCREEN CLOSE 1¶
             ¶
¶
endprog:    LIBRARY CLOSE¶
            END¶
           ¶
           ¶
SUB InitCop STATIC¶
      SHARED UCopList&¶
      opt&      = 2^0+2^1+2^16¶
      UCopList& = AllocMem&(12,opt&)¶
      IF UCopList& = 0 THEN ERROR 7¶
END SUB¶
¶
SUB ActiCop STATIC¶
      SHARED UCopList&¶
      waitC 10000,256¶
```

```
      viewport& = ViewPortAddress&(WINDOW(7))¶
      POKEL viewport&+20,UCopList&¶
      CALL  RethinkDisplay¶
END SUB¶
            ¶
SUB waitC(y%,x%) STATIC¶
      SHARED UCopList&¶
      CALL CWait(UCopList&,y%,x%)¶
      CALL CBump(UCopList&)¶
END SUB¶
¶
SUB moveC(reg%,value%) STATIC¶
      SHARED UCopList&¶
      CALL CMove(UCopList&,reg%,value%)¶
      CALL CBump(UCopList&)¶
END SUB¶
```

SUB program Descriptions:

`InitCop`:

the exec function `AllocMem` assigns a 12 byte memory area for the `UCopList` data structure.

`WaitC`:

a wait instruction is placed in the user list. The graphic library command `CWait`, also called `CBump()`, raises the internal pointer for the user list.

`MoveC`:

calls the function `CMove`, from the graphic library, which places a move instruction in the user list. `CBump()` raises the user list pointer again.

`ActiCop`:

a last `WaitC` is placed in the user list. This wait is for a screen position that the electronic beam will never reach. This assignment closes the list and equals the macro `CEND`.

The address of our user list is written to the appropriate pointer in ViewPort for the desired screen. The Intuition function `RethinkDisplay` generates the new Copper list for View and sends it to the Copper. The new display then appears.

At the end, the screen is closed and the Copper list is removed from the main Copper list in View. All reserved memory is returned to the system.

4.5.3 Programming 400 simultaneous colors

We now know the principles of Copper programming. The next program is a small demonstration of this powerful technique. We are going to change the background color for each screen row by using `WaitC`. At the same time we will also be changing the drawing color for each row. With 200 screen rows per display, we finish with 400 colors on the screen at the same time. Colors two and three remain normal.

Note: Do not use more than 1600 Copper instructions in one list.

The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 4.5.3 ¶
'# Program: Copper Raster Interrupt II¶
'# Date:    04/11/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' Copper Programming can create 400 different colors¶
' in the Background and Foreground instead of the¶
' usual 4 colors with 2 bit-planes.¶
¶
PRINT "Searching for .bmap files..."¶
¶
'INTUITION-Library¶
DECLARE FUNCTION ViewAddress& LIBRARY¶
DECLARE FUNCTION ViewPortAddress& LIBRARY¶
'RethinkDisplay()¶
¶
'EXEC-Library¶
DECLARE FUNCTION AllocMem& LIBRARY¶
'FreeMem()¶
¶
'GRAPHICS-Library¶
'CWait()¶
'CMove()¶
'CBump()¶
¶
LIBRARY "intuition.library"¶
LIBRARY "graphics.library"¶
LIBRARY "exec.library"¶
¶
```

```
pre:        CLS¶
            SCREEN 1,640,200,2,2¶
            WINDOW 2,"COPPER!",(0,0)-(630,186),16,1¶
            ¶
            PRINT "Direct Copper programming makes it
possible."¶
            PRINT "200 Background Colors!"¶
            PRINT "Patience Please...Calculating
Instruction Lists."¶
            ¶
init:       koloregister1% = 384¶
            koloregister2% = 386¶
            xCoordinate%   = 20¶
            maxY%          = 200¶
            ¶
main:       InitCop¶
            FOR loop%=1 TO maxY%¶
              waitC loop%,xCoordinate%¶
              moveC koloregister1%,loop%¶
              moveC koloregister2%,4096-loop%¶
            NEXT loop%¶
            ActiCop¶
            ¶
           '* Display text for this effect¶
            LOCATE 5,1¶
            PRINT "The Background color shows 200
individual"¶
            PRINT "colors! The text colors are also not
plain"¶
            PRINT "anymore: 200 yellows, one per raster"¶
            PRINT "line. Here a useful program could"¶
            PRINT "take over the job instead of"¶
            PRINT "this useless program loop...!"¶
            LOCATE 15,1¶
            PRINT "Please press a key when you"¶
            PRINT "have finished reading."¶
            WHILE INKEY$=""¶
            WEND¶
            ¶
            LOCATE 11,1¶
            PRINT "That's not going to happen this
time!"¶
            ¶
            FOR t=1 TO 2000:NEXT t¶
            CLS¶
            PRINT "Graphic Demo coming up!"¶
            ¶
            LINE (0,100)-(630,190),2,bf¶
            FOR loop%=0 TO 630 STEP 30¶
              LINE (loop%*1.5,190)-(loop%,100),1¶
            NEXT loop%¶
            FOR loop%=100 TO 190 STEP 20¶
              LINE (0,loop%)-(630,loop%),1¶
            NEXT loop%    ¶
           ¶
            CIRCLE (300,80),120,3¶
```

```
            PAINT (300,80),3,3¶
                        ¶
            CIRCLE (300,80),100,1¶
            PAINT (300,80),1,1¶
            ¶
            CIRCLE (300,146),180,3,,,1/15¶
            PAINT (300,146),1,3¶
          ¶
            LOCATE 1,1¶
            PRINT "Press a Key!"+SPACE$(40)¶
            ¶
            WHILE INKEY$="":WEND¶
            ¶
            WINDOW CLOSE 2¶
            SCREEN CLOSE 1¶
            ¶
¶
ende:       LIBRARY CLOSE¶
            END¶
           ¶
           ¶
SUB InitCop STATIC¶
      SHARED UCopList&¶
      opt&      = 2^0+2^1+2^16¶
      UCopList& = AllocMem&(12,opt&)¶
      IF UCopList&=0 THEN ERROR 7¶
END SUB¶
¶
SUB ActiCop STATIC¶
      SHARED UCopList&¶
      waitC 10000,256¶
      viewport& = ViewPortAddress&(WINDOW(7))¶
      POKEL viewport&+20,UCopList&¶
      CALL RethinkDisplay¶
END SUB¶
          ¶
SUB waitC(y%,x%) STATIC¶
      SHARED UCopList&¶
      CALL CWait(UCopList&,y%,x%)¶
      CALL CBump(UCopList&)¶
END SUB¶
¶
SUB moveC(reg%,value%) STATIC¶
      SHARED UCopList&¶
      CALL CMove(UCopList&,reg%,value%)¶
      CALL CBump(UCopList&)¶
END SUB¶
```

The text displayed by the program makes the details of the changed display clearer and more impressive. A simple graphic is created but it has a fascinating appearance because of the Copper programming. This graphic is formed from more than 400 colors with only two bit-planes. After pressing a key, the normal screen appears.

4.6 The layers: soul of the windows

We will continue building our picture of the graphics system. In the RastPort structure (see Section 3.6) there is a pointer to the *layers*. Layers refers to the independent system components of the operating system that are controlled through the layer libraries.

To discover what layers are, take a look at your Amiga monitor. You probably cannot see the layers because of all the windows. Actually, every window is a layer.

Just as the screen is simply another ViewPort, a window is just another layer. A layer handles most of the work required to create windows. One problem which always occurs when a computer works with windows is that everything you see on the screen, including the screen background and windows, is stored in the bit-planes of the bit-map. An ideal example of this problem is a display that contains the screen background and many window fragments. These windows can overlap, can be covered completely by others or can be displayed by themselves. As soon as one window overlaps another, this must be recorded because the overlapped portion consists of two windows divided within the same bit-map. The layers insure that the covered window section is saved to another area of memory. Whenever the covered window (or a portion of it) becomes visible again, the layers copy the uncovered piece back into the screen bit-map.

Before we discuss this theory in more detail we are going to trap and display a layer for you. The following small program performs this operation. The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 4.6¶
'# Program: A Layer¶
'# Date:    01/05/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' A simple layer - the basis of every window¶
' is generated.¶
¶
PRINT "Searching for .bmap files..."¶
```

```
¶
'LAYERS-Library¶
DECLARE FUNCTION CreateUpFrontLayer& LIBRARY¶
'DeleteLayer()¶
'MoveLayer()¶
¶
'GRAPHICS-Library¶
'Text()¶
'Move()¶
¶
LIBRARY "graphics.library"¶
LIBRARY "layers.library"¶
¶
initPars:    CLS¶
             scrAdd&          = PEEKL(WINDOW(7)+46)¶
             screenLayerInfo& = scrAdd&+224¶
             screenBitMap&    = scrAdd&+184¶
             x0%              = 10¶
             y0%              = 20¶
             x1%              = 400¶
             y1%              = 80¶
             yp%              = 1¶
             ¶
thatYou:    'can also see it¶
             CLS¶
             LINE (1,1)-(600,180),2,bf¶
                          ¶
LayerHer:    layer& = CreateUpFrontLayer&(
screenLayerInfo&,screenBitMap&,x0%,y0%,x1%,y1%,typ%,0)¶
           ¶
whatToDo:    layerRast& = PEEKL(layer&+12)¶
             text$      = "This is the soul of a Window:
A Layer!"¶
             CALL Move(layerRast&,3,8)¶
             CALL
text(layerRast&,SADD(text$),LEN(text$))¶
¶
moveit:      dx% = 2¶
             dy% = 1¶
             FOR loop1%=1 TO 30¶
               CALL
MoveLayer(screenLayerInfo&,layer&,dx%,dy%)¶
             NEXT loop1%¶
                          ¶
waitlp:      LOCATE 1,1¶
             PRINT "Press any key to End!"¶
             WHILE in$=""¶
               in$=INKEY$¶
             WEND¶
             ¶
removeIt:    CALL DeleteLayer(screenLayerInfo&,layer&)¶
¶
thatsIt:     LIBRARY CLOSE¶
             END¶
```

As you can see, our small layer acts much like a large window. When you move the mouse pointer over the layer and click the left mouse button, the layer is activated and the window ripples. In order for the layer to be a complete window, it would need the frame, the gadgets and a menu.

When you press a key the layer completely disappears.

To create the above program, we used functions from both the graphic and layers libraries. However, the layers library is more important in this application. The layers function, `CreateUpfrontLayer`, which generates our layer, requires eight arguments and returns the start address of the layer data block to the BASIC program.

```
layer&=CreateUpfrontLayer&(layerInfo&,bitmap&,x0%,yo%,x
          1%,y1%,typ%,sbitmap&)
```

`layer&:`	The address of our new layer data block
`layerInfo&:`	The address of the structure LayerInfo
`bitmap&:`	The address of the bit-map, where the new layer is to be produced
`x0%,y0%:`	Coordinates of the upper left hand corner of the layer
`x1%,y1%:`	Coordinates of the lower right hand corner

The `LayerInfo` structure address and the Bit-map structure address are found in our well known screen structure (see Section 3.4).

Now we return to the program. Using the above functions opens a layer. Now we want to display text in the layer. Keep in mind that a layer also contains a RastPort (see Section 3.6). By using the functions text and move from the graphic library (see Section 3.6.1), we can display text through this RastPort.

After studying the layer, we close the layer again by using `DeleteLayer`.

Now we can take a look at the layer's data structure. Just like every window, the layers also have this structure. It is constructed like this:

Data structure Layer/layers/ 192 Bytes

Offset	Type	Description
+ 000	Long	Pointer to layer in foreground
+ 004	Long	Pointer to layer in background
+ 008	Long	Pointer to first ClipRect
+ 012	Long	Pointer to the RastPort of this layer
+ 016	---	Rectangle structure, the limits of layer
		+ 16 Word MinX
		+ 18 Word MinY
		+ 20 Word MaxX
		+ 22 Word MaxY
+ 024	Byte	Lock
+ 025	Byte	LockCount
+ 026	Byte	LayerLockCount
+ 027	Byte	reserved
+ 028	Word	reserved
+ 030	Word	Layer flags
+ 032	Long	Pointer to superbitmap, when available
+ 036	Long	SuperClipRect
+ 040	Long	Pointer to window
+ 044	Word	ScrollX
+ 046	Word	ScrollY
+ 048	---	Message port "LockPort"
+ 082	---	Message "LockMessage"
+ 102	---	Message port "ReplyPort"
+ 136	---	Message "1 LockMessage"
+ 156	Long	Pointer to first rectangle of Damagelist
+ 160	Long	Pointer to ClipRects
+ 164	Long	Pointer to LayerInfo structure
+ 168	Long	Pointer to Task with actual lock
+ 172	Long	Pointer to SuperSaveClipRects
+ 176	Long	Pointer to CR ClipRects
+ 180	Long	Pointer to CR2 ClipRects
+ 184	Long	Pointer to CRNEW ClipRects
+ 188	Long	System use

4.6.1 The Layer data structure

The data structure we just introduced requires further explanation. We will examine it field by field with the same format we have been using.

To obtain the starting address for the layer of your actual output window, use the following:

```
layer&=PEEKL(WINDOW(8))
```

The starting address of the layer data structure you have created is automatically returned by the appropriate layer functions. Later we will come back to this subject.

Offset 0 and 4: Pointer to other layers

This pointer contains the starting address of the layer data block for a layer that is behind or in front of your layer. The same relationship applies to layers as to windows; you can move from one layer to all other layers in the system.

Offset 8: First ClipRect

`ClipRect` is another data structure which describes a current rectangular piece of a layer. This pointer is to the first `ClipRect` structure of this layer, from which you obtain the next and the following `ClipRect` address. This chain of `ClipRects` describes the visible portion of this layer.

Offset 12: The RastPort

This offset contains the starting address of the RastPort for this layer. Most functions of the graphic library require the RastPort address with which they will be working.

Since every Intuition window has a layer, it also has its own layer RastPort. This RastPort is identical with the RastPort of the window data structure:

```
RastPort1&=PEEKL(WINDOW(7)+50)
RastPort2&=WINDOW(8)
layer&=PEEKL(WINDOW(8))
RastPort3&=PEEKL(layer&+12)
PRINT RastPort1&
PRINT RastPort2&
PRINT RastPort3&
```

The starting addresses returned for the RastPorts are the same.

Offset 16, 18, 20, and 22: Bounds

The X and Y values stored here set the layer limits. Any drawing function that uses coordinates is clipped off if it passes outside these boundaries. Let's first determine the limits of our own layers:

```
windo&=WINDOW(7)
RastPort&=WINDOW(8)
layer&=PEEKL(RastPort&)
```

```
x0%=PEEKW(layer&+16)
y0%=PEEKW(layer&+18)
x1%=PEEKW(layer&+20)
y1%=PEEKW(layer&+22)

PRINT x0%,y0%
PRINT x1%,y1%

END
```

The result is the coordinates of the upper left and lower right hand corners of our drawing plane.

Naturally you could use this method to define your own drawing plane. Everything outside of this area would be clipped off.

```
REM 4.6.1 Offsets 16-22 Example B

PRINT "Searching for .bmap file"

init:   '* Address of the Data Structures
         CLS
         windo&     = WINDOW(7)
         RastPort& = WINDOW(8)
         layer&     = PEEKL(RastPort&)

        '* Current Valid Limits
         x0% = PEEKW(layer&+16)
         y0% = PEEKW(layer&+18)
         x1% = PEEKW(layer&+20)
         y1% = PEEKW(layer&+22)

         scrWidth%  = x1%-x0%
         scrHeight% = y1%-y0%

main:   '* Demo
         LINE (x0%,y0%)-(x1%,y1%),2,bf

        '* Set new Limits
         nx0% = x0%+.25*scrWidth%
         nx1% = x1%-.25*scrWidth%
         ny0% = y0%+.25*scrHeight%
         ny1% = y1%-.25*scrHeight%

         POKEW layer&+16,nx0%
         POKEW layer&+18,ny0%
         POKEW layer&+20,nx1%
         POKEW layer&+22,ny1%

        '* It looks like this:
         FOR test%=0 TO 40
           PRINT STRING$(50,"*")
         NEXT test%

         CLS
```

```
        PRINT "Enter CONT!"

        STOP

       '* Restore Original Limits
        POKEW layer&+16,x0%
        POKEW layer&+18,y0%
        POKEW layer&+20,x1%
        POKEW layer&+22,y1%

END
```

Offset 24, 25 and 26: Lock-Fields

The Amiga is a multitasking computer, which means that many programs can be running at the same time. So, it is possible for several programs to attempt access to the same layer at the same time. These conflicting access attempts would cause the programs to abort. To prevent this from happening, there are the layer functions, `LockLayer` and `UnLockLayer`. Through these functions the tasks have unlimited access to the layers. As long as a task utilizes lock, another task cannot change the contents of the layer data structure.

These fields control the lock technique. The first field determines whether this layer is currently locked. The second is a counter for the program that is now using the layer. The third is a counter that keeps track of other tasks attempting access to the layer.

Offset 30: Flags

There are various layer types that we will discuss shortly. This field contains an identity flag for this layer:

Bit 0: 1=Layersimple
Bit 1: 1=Layersmart
Bit 2: 1=Layersuper
Bit 6: 1=Layerbackdrop
Bit 7: 1=Layerrefresh

Offset 32: Superbitmap

A layer has its own drawing plane and bit-map when it is in the layersuper mode. The pointer to these is stored in this offset. We will explain this in detail later.

Offset 36: SuperClipRect

When they are used, the `ClipRects` for the Superbitmap are here (see offset 8).

Offset 40: Window

Normally layers are linked with Intuition windows. When this happens, this pointer contains the address of the corresponding window data structure.

This field is extremely important when you want to integrate layers with existing windows. We are going to cover this in more detail shortly.

Offset 44 and 46: Scrolling

The referenced drawing plane for a layer of type layersuper can be much larger than the layer limit parameters. You can then use the layer like a peephole that moves around a giant graphic. More on this later.

Offsets 48 - 136: Messages and Message Ports

Messages and message ports are handled by the `exec.library`. Different tasks can communicate with each other through the message and message ports, which are similar to a mailbox and transmitter. Messages are their letters. The reply port is the mailbox and the other message port sends the messages.

Offset 156: Damage List

We mentioned before that layers are responsible for restoring covered portions of windows once they are no longer covered. Because of this, we have a damage list which consists of a chain of data structures called "regions". These regions represent the rectangular portions of a layer. The damage list contains all the damaged portions of its own window (or layer) that is overlapped by other windows (or layers).

The remaining offset fields contain system information that BASIC can not use.

4.7 The different layer types

In all, the Amiga has four different layer types:

```
Layersimple
Layersmart
Layersuper
Layerbackdrop
```

These modes determine how, and by what method, the covered portions of a layer are handled:

Simple Refresh (Layersimple)

Each time a piece of this layer becomes visible (is uncovered or brought to the foreground), the program that created the layer must redraw the new visible portion. Since this type of layer does not automatically save covered sections so that it can repair the damage later, it is the responsibility of the program (or, in this case, your responsibility) to repair the sections.

Layers of this type are fast and require little memory. However, they require more work because their contents have to be redrawn whenever they are covered by another layer.

Smart Refresh (Layersmart)

When part of this layer type is covered, the system automatically creates an intermediate storage area where the covered portion is temporarily saved. Whenever the layer is uncovered, the portion temporarily saved is automatically transferred back to its original location.

Superbitmap (Layersuper)

This layer contains its own bit-planes where the entire contents of the layer are stored. The portion of the layer that is currently visible on the screen is copied to the common screen bit-map.

It is possible to create a layer bit-map that is (much) larger than the layer itself (it can be up to 1024 x 1024 pixels in size). This giant area is easily scrolled.

Backdrop (Layerbackdrop)

A backdrop layer exists behind all other current layers.

We are now going to give you a look into the world of layers and show you what can be accomplished with their help.

4.7.1 Simple layers: your own requester

An excellent use for simple layers is the creation of requesters, which help highlight special parts of the program. For example, when the user places a disk in the drive, a graphic will be loaded. The following program uses simple layers to help create your own requester. You can call a request with:

```
Request nr%,x%,y%,text$
```

`nr%:`	Number of the request (0-10)
`x%:`	X coordinate of left hand corner of the requester
`y%:`	Y coordinate of top corner of the requester
`text$:`	Text for the requester

The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 4.7.1 ¶
'# Program: A layer - Your Own Requester¶
'# Date:    01/05/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
PRINT "Searching for .bmap files..."¶
¶
' Demonstrates the use of layers¶
¶
'LAYERS-Library¶
DECLARE FUNCTION CreateUpFrontLayer& LIBRARY¶
'DeleteLayer()¶
¶
'GRAPHICS-Library¶
'Draw()¶
'Move()¶
'Text()¶
```

```
¶
LIBRARY "graphics.library"¶
LIBRARY "layers.library"¶
¶
variables:   DIM SHARED layer&(10)¶
¶
init:       'Background¶
             CLS¶
             FOR loop%=1 TO 15¶
               PRINT STRING$(80,"#")¶
             NEXT loop%  ¶
¶
main:        Request 1,80,40,"Request Nr. 1"¶
             Request 2,50,50,"Request 2: These are
Layers!"¶
             FOR t%=1 TO 30000:NEXT t%¶
             CloseRequest 1¶
             Request 1,30,30,"Positioned as desired"¶
             FOR t%=1 TO 30000:NEXT t%¶
             CloseRequest 2¶
             CloseRequest 1¶
             Request 1,200,100,"Thats it!"¶
             FOR t%=1 TO 2000:NEXT t%¶
             CloseRequest 1¶
¶
thatsIt:     LIBRARY CLOSE¶
             END¶
¶
SUB Request(nr%,x0%,y0%,text$) STATIC¶
        SHARED screenLayerInfo&¶
        IF layer&(nr%)<>0 THEN EXIT SUB¶
        scrAdd&          = PEEKL(WINDOW(7)+46)¶
        screenLayerInfo& = scrAdd&+224¶
        screenBitMap&    = scrAdd&+184¶
        x1%         = (LEN(text$)+2)*8-8¶
        y1%         = 12¶
        layer&(nr%) =
CreateUpFrontLayer&(screenLayerInfo&,
screenBitMap&,x0%,y0%,x0%+x1%,y0%+y1%,typ%,0)¶
        layerRast&  = PEEKL(layer&(nr%)+12)¶
        CALL Draw(layerRast&,x1%,0)¶
        CALL Draw(layerRast&,x1%,y1%)¶
        CALL Draw(layerRast&,0,y1%)¶
        CALL Draw(layerRast&,0,0)¶
        CALL Move(layerRast&,3,9)¶
        CALL text(layerRast&,SADD(text$),LEN(text$))¶
END SUB¶
¶
SUB CloseRequest(nr%) STATIC¶
        SHARED screenLayerInfo&¶
        IF layer&(nr%) = 0 THEN EXIT SUB                ¶
        CALL DeleteLayer(screenLayerInfo&,layer&(nr%))¶
        layer&(nr%) = 0¶
END SUB¶
```

You can open up to 11 requesters at the same time (this can be increased but usually eleven requesters is enough). Since the X and Y coordinates of the upper left hand corner of every requester are relative to the upper left hand corner of the screen, not to your window, requesters can appear anywhere, not just inside your windows. However, outside the window they can still cause some small damage because the damage list is not activated there.

The command `CloseRequest` closes the requester (and also the layer) again.

4.7.2 The superlayer - 1024 x 1024 pixels!

Now we come to a very special layer type called the *superlayer*, which is completely different from all other layer types because it is equipped with its own graphic memory area. This memory area can also be larger than the visible portion on your screen. This layer can manage a total drawing area of up to 1024 x 1024 pixels large.

Creating a layer of this type should not be a problem because you are familiar with the `CreateUpfrontLayer` command from the layer library. However, to make this layer useful is a more difficult task.

First we need to position the new layer. The best way to do this is to plant this layer on top of an existing window by selecting a layer that corresponds to the size of the window and then place your layer exactly on top of it. By using this method, no one will notice what you did. We will test this in the following program:

The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 4.7.2 ¶
'# Program: Superbitmap¶
'# Date:    01/04/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' Shows how up to 1024 x 1024 -pixel layers¶
' are created, programmed and scrolled.¶
' First Demo.¶
¶
```

```
'LAYERS-Library¶
DECLARE FUNCTION CreateUpFrontLayer& LIBRARY¶
'DeleteLayer()¶
'ScrollLayer()¶
¶
'GRAPHICS-Library¶
DECLARE FUNCTION AllocRaster& LIBRARY¶
'FreeRaster()¶
'SetRast()¶
'Move()¶
'Draw()¶
'WaitTOF()¶
'Text()¶
¶
'EXEC-Library¶
DECLARE FUNCTION AllocMem& LIBRARY¶
'FreeMem()¶
¶
'INTUITION-Library¶
'SetWindowTitles()¶
¶
PRINT "Searching for .bmap files... ";¶
¶
LIBRARY "layers.library"¶
LIBRARY "graphics.library"¶
LIBRARY "exec.library"¶
LIBRARY "intuition.library"¶
¶
PRINT "found."¶
¶
initPar:    '* Screen Parameters¶
            scrWidth%  = 320¶
            scrHeight% = 200¶
            scrDepth%  = 1¶
            scrMode%   = 1¶
            scrNr%     = 1¶
¶
           '* Window Parameters¶
            windWidth%  = scrWidth%-9¶
            windHeight% = scrHeight%-26¶
            windNr%     = 1¶
            windTitle$  = "Working Area"¶
            windMode%   = 0¶
¶
           '* Super Bitmap¶
            superWidth%  = 800¶
            superHeight% = 400¶
            superFlag%   = 4¶
¶
initDisp:  '* Open Screen and Window¶
            SCREEN scrNr%,scrWidth%,scrHeight%,
scrMode%,scrDepth%¶
            WINDOW windNr%,windTitle$,(0,0)-
(windWidth%,windHeight%),windMode%,scrNr%¶
            WINDOW OUTPUT windNr%¶
            PALETTE 1,0,0,0¶
```

```
              PALETTE 0,1,1,1¶
¶
             '* Layer Size¶
              windLayer& = PEEKL(WINDOW(8))¶
              LayMinX%   = PEEKW(windLayer&+16)¶
              layMinY%   = PEEKW(windLayer&+18)¶
              layMaxX%   = PEEKW(windLayer&+20)¶
              layMaxY%   = PEEKW(windLayer&+22)¶
                           ¶
initSys:     '* Read System Parameters¶
              windAdd&      = WINDOW(7)¶
              scrAdd&       = PEEKL(windAdd&+46)¶
              scrBitMap&    = scrAdd&+184¶
              scrLayerInfo& = scrAdd&+224¶
¶
initSBMap:   '* Create Superbitmap¶
              opt&         = 2^0+2^1+2^16¶
              superBitMap& = AllocMem&(40,opt&)¶
              IF superBitMap& = 0 THEN¶
                PRINT "Hmm. Not even 40 Bytes, okay?"¶
                ERROR 7¶
              END IF¶
¶
             '* ...and make use of it¶
              CALL InitBitMap(superBitMap&,scrDepth%,
superWidth%,superHeight%)¶
              superPlane&    = AllocRaster&(superWidth%,
superHeight%)¶
              IF superPlane& = 0 THEN¶
                PRINT "No Room!"¶
                CALL FreeMem(superBitMap&,40)¶
                ERROR 7¶
              END IF¶
              POKEL superBitMap&+8,superPlane&¶
¶
             '* Open Superbitmap Layer¶
              superLayer&=CreateUpFrontLayer&(
scrLayerInfo&,scrBitMap&,LayMinX%,layMinY%,layMaxX%,layMa
xY%,superFlag%,superBitMap&)¶
              IF superLayer&=0 THEN¶
                PRINT "No Layer Today!"¶
                CALL FreeRaster(superPlane&,superWidth%,
superHeight%)¶
                CALL FreeMem(superBitMap&,40)¶
                ERROR 7¶
              END IF¶
¶
             '* ignore next line for now!¶
             '*** PUT 4.7.3 EXPANSION HERE ***¶
¶
             '* Run new RastPort¶
              superRast& = PEEKL(superLayer&+12)¶
¶
prepare:     '* Prepare Drawing Area¶
              CALL SetRast(superRast&,0)¶
¶
```

```
                CALL Move(superRast&,0,0)¶
                CALL
Draw(superRast&,superWidth%,superHeight%)¶
¶
                CALL Move(superRast&,0,10)¶
                text1$="Use Cursor Keys to Scroll"¶
                CALL Text(superRast&,SADD(text1$),
LEN(text1$))¶
¶
                CALL Move(superRast&,0,30)¶
                text2$="'S' Key to Abort"¶
                CALL Text(superRast&,SADD(text2$),
LEN(text2$))¶
¶
               '* Coordinates¶
                POKEW superRast&+34,&HAAAA ¶
                FOR loop%=0 TO superWidth% STEP 50¶
                  CALL Move(superRast&,loop%,0)¶
                  CALL Draw(superRast&,loop%,superHeight%)¶
                NEXT loop%¶
                FOR loop%=0 TO superHeight% STEP 50¶
                  CALL Move(superRast&,0,loop%)¶
                  CALL Draw(superRast&,superWidth%,loop%)¶
                NEXT loop%¶
                POKEW superRast&+34,&HFFFF¶
¶
doScroll:      '* Control Scrolling¶
                WHILE in$<>"S" ¶
                  in$ = UCASE$(INKEY$)¶
                  y%  = 0¶
                  x%  = 0¶
                  IF in$ = CHR$(30) THEN '<-¶
                    IF ox%<(superWidth%-layMaxX%+LayMinX%-1)
THEN¶
                      x%  = 1¶
                      ox% = ox%+1¶
                    END IF¶
                  ELSEIF in$=CHR$(31) THEN '->¶
                    IF ox%>0 THEN¶
                      x%  = -1¶
                      ox% = ox%-1¶
                    END IF¶
                  ELSEIF in$=CHR$(29) THEN 'up¶
                    IF oy%<(superHeight%-layMaxY%+layMinY%-1)
THEN¶
                      y%  = 1¶
                      oy% = oy%+1¶
                    END IF¶
                  ELSEIF in$=CHR$(28) THEN 'down¶
                    IF oy%>0 THEN¶
                      y%  = -1¶
                      oy% = oy%-1¶
                    END IF¶
                  END IF¶
                  IF in$<>"" THEN¶
```

```
                CALL ScrollLayer(scrLayerInfo&,
superLayer&,x%,y%)¶
                actu$ = windTitle$+" [X]="+STR$(ox%)+"
[Y]="+STR$(oy%)+CHR$(0)¶
                CALL WaitTOF¶
                CALL SetWindowTitles(windAdd&,
SADD(actu$),0)¶
              END IF¶
            WEND¶
            ¶
deleteSys: '* Delete System¶
            CALL DeleteLayer(scrLayerInfo&,superLayer&)¶
            CALL FreeRaster(superPlane&,superWidth%,
superHeight%)¶
            CALL FreeMem(superBitMap&,40)¶
            SCREEN CLOSE scrNr%¶
            WINDOW windNr%,"hi!",,,-1¶
            LIBRARY CLOSE¶
            END¶
```

Immediately after the program starts, you see a window named "Working Area", which contains a line that is a downward oriented diagonal. What you are actually seeing is a superbitmap layer that has displayed itself in this window. To prove this, move the mouse into the raster area and click the left button. The title bar will immediately ghost because you have just activated the invisible layer in front of your window. Click the mouse on the window title bar and everything will return to the way it was.

More than once we have mentioned that a superbitmap can manage a much larger area than what will fit on your screen. Our demonstration program uses this ability. Press any of the cursor keys next to the number pad of your keyboard. You can shift the position of our layer and see a different position of the layer controlled drawing using the cursor keys.

When you have seen enough of this program, please press the <S> key (for stop). The old display will return immediately (do not use <Ctrl> <C>, for BREAK, because then the superbitmap layer will not disappear).

We have now come to the reason for this project. In this program, we have used functions from the layers, graphic, exec- and Intuition libraries. The following functions are especially important:

```
CreateUpfrontLayer()
AllocRaster()
AllocMem()
ScrollLayer()
```

In addition we used:

```
InitBitMap()
SetRast()
Move()
Draw()
WaitTOF()
SetWindowTitles()
```

and naturally:

```
DeleteLayer()
FreeRaster()
FreeMem()
```

Now we move on to the program. First we open a screen with a depth of one, which means one bit-plane and a maximum of two colors. We are using this depth because our superbitmap layer requires the same amount of memory planes as the screen depth in which it appears. Since the planes of a superbitmap are very memory intensive, we are only able to create one single plane.

Our superbitmap is going to be 800 pixels wide and 400 pixels high.

After the window and screen are open, we determine the size of the layers (we mean the size of the layer on the screen, not the size of the layers drawing plane). Since the layer is going to fill the entire window, we read the required parameters from the existing layer of our window (see Section 4.5, offsets 16-22).

Before we can use "`CreateUpfrontLayer`" to bring our layer to life, we must create the private bit-map of our layer. This only applies to layers of the layersuper type because memory is automatically allocated for the other layer types. We now continue with a display that is similar to the display in Section 4.4. We create a 40 byte sized Bit-map structure and make it ready for use with `InitBitMap`. We can obtain an additional bit-plane by using the graphic function, `AllocRaster`, which requires, in pixels, the X and Y dimensions of the bit-plane and returns a pointer for the starting address of the new plane (when sufficient memory is available).

After the bit-plane is linked to our new Bit-map structure we can finally call `CreateUpfrontLayer`. The variable `superflag%` contains the value four (=Superlayer). The address of our new Bit-map structure is also sent.

As soon as the layer has successfully opened, it should have something to display. Using the function `SetRast`, we erase the layer contents and draw a diagonal with the help of the draw command in the graphic library.

The program routine, `doScroll`, manages the scrolling of the superbitmap through the use of cursor keys. We use the layer function `scrolllayer` which requires four parameters:

```
ScrollLayer(layerinfo&,layer&,x%,y%)
```

`layerinfo&:`	Address of the Layerinfo structure (see screen)
`layer&:`	Address of our new superlayer
`x%,y%:`	Number of pixels, to scroll the layer contents (negative values = opposite direction)

After each scroll, we use the Intuition function `SetWindowTitles` to display the current X and Y position in the title bar of the window. The function `WaitTOF` (Top Of Frame), which comes from the graphic library, waits for the electronic raster beam to reach the topmost display line. This prevents the window title bar from being changed while the electronic beam is moving through it. This would produce an unsightly flickering effect.

When you press the <S> key, the superlayer is closed. Finally we return the memory for the bit-plane and Bit-map structure to the system.

This program is useful as a first test. However, our programming technique is incomplete because there are a few serious problems:

a) When the user clicks the mouse in the layer, this layer can accidentally be activated and their own window can be deactivated. This means that their own program will no longer recognize any key or mouse entries.

b) Since we generated the superlayer directly from the system, it is not possible for us to draw in the superlayer with the BASIC graphic commands. We must use the functions of the graphic libraries.

In order to use the superbitmap, we have to solve these problems. We will do just that in the following section.

4.7.3 Permanently deactivating layers

To solve the first problem, we must prevent the activation of the layer when using the mouse and also prevent the mouse from having any

effect on our layer (or in other words, keep our window permanently active).

After looking into the Amiga graphic system, we find our solution. In every layer structure at offset 40 there is a field named *Pointer to Window*. This field is set to zero for simple layers. For layers used for Intuition windows, this field contains a pointer to the window data structure of the window used by this layer. This pointer's only function is to tell Intuition when the user activates this layer by clicking the left mouse button.

To prevent Intuition from deactivating our window when the layer is activated, we must write, into the data field of our layer, the address of the window structure we want kept active. The following lines do this:

```
POKEL layer&+40,WINDOW(7)
```

You can test this technique on our demonstration program from Section 4.7.2. Add the following line to the program after the '**** PUT 4.7.3 EXPANSION HERE ****' that marks the entry position:

```
POKEL superLayer&+40,WINDOW(7)
```

After the program starts, you can move the mouse freely around the layer and click the left mouse button. Our window stays active.

This solves the first problem. Now we will move on to the solution for the second problem.

4.7.4 Using the BASIC commands within a layer

We are going to analyze the problem of AmigaBASIC graphic commands, such as `LINE`, `CIRCLE` and `PRINT`, not being able to draw in our layer. Since there is no way to direct the commands to the layer, we have to reach into the system.

It is possible to transfer the graphic output from our window to a layer by carefully manipulating the pointer. It is very important that you remember to restore the display to normal before closing the layer. If you don't do this, the system becomes confused, hangs and usually creates a Guru.

The technique looks like this:

```
 backupRast&=PEEKL(layer&+12)
'* Save RastPort of layer
```

```
 backupLayer&=PEEKL(WINDOW(8))
'* Save Layer of Window
 POKEL WINDOW(8),layer&
 POKEL layer&+12,WINDOW(8)
```

Now all the graphics commands sent from AmigaBASIC are executed in the layer.

Note: You can use all the BASIC commands with confidence except for the various fill commands, like `PAINT`, `LINE()-(),,bf`. The reason for this is found in the data structure, `TmpRas`, which is located in the RastPort. For fill commands, `TmpRas` has to point to a memory area that is at least as big as one bit-plane of the layer. You could provide more memory to the `TmpRas` structure which would enable you to use the fill commands. However, this would use more memory than is worth the effort. For this example, we would need 40,000 bytes. If you do have sufficient memory, we have outlined the technique for remodeling the `TmpRas` structure further on in the book.

The following lines restore your display to the original window:

```
POKEL WINDOW(8),backupLayer&
POKEL layer&+12,backupRast&
```

We are going to use our new knowledge in the following demonstration program. The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 4.7.4¶
'# Program: Superbitmap with BASIC¶
'#          Graphics Commands¶
'# Date:    04/12/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' Makes it possible to use AmigaBASIC graphic¶
' commands in a Superbitmap layer.¶
¶
PRINT "Searching for .bmap files..."¶
¶
'LAYERS-Library¶
DECLARE FUNCTION CreateUpFrontLayer& LIBRARY¶
'DeleteLayer()¶
'ScrollLayer()¶
¶
'GRAPHICS-Library¶
```

```
DECLARE FUNCTION AllocRaster& LIBRARY¶
'FreeRaster()¶
'SetRast()¶
'Move()¶
'Draw()¶
'WaitTOF()¶
¶
'EXEC-Library¶
DECLARE FUNCTION AllocMem& LIBRARY¶
'FreeMem()¶
¶
'INTUITION-Library¶
'SetWindowTitles()¶
¶
LIBRARY "layers.library"¶
LIBRARY "graphics.library"¶
LIBRARY "exec.library"¶
LIBRARY "intuition.library"¶
¶
¶
initPar:    '* Screen Parameters¶
             scrWidth%  = 320¶
             scrHeight% = 200¶
             scrDepth%  = 1¶
             scrMode%   = 1¶
             scrNr%     = 1¶
             ¶
            '* Window Parameter¶
             windWidth%  = scrWidth%-9¶
             windHeight% = scrHeight%-26¶
             windNr%     = 1¶
             windTitle$  = "Working Area"¶
             windMode%   = 0¶
             ¶
            '* Super Bitmap¶
             superWidth%  = 800¶
             superHeight% = 400¶
             superFlag%   = 4¶
             ¶
initDisp:   '* Open Screen and Window¶
             SCREEN scrNr%,scrWidth%,scrHeight%,
scrMode%,scrDepth%¶
             WINDOW windNr%,windTitle$,(0,0)-
(windWidth%,windHeight%),windMode%,scrNr%¶
             WINDOW OUTPUT windNr%¶
             PALETTE 1,0,0,0¶
             PALETTE 0,1,1,1¶
            ¶
            '* Layer Size¶
             windLayer& = PEEKL(WINDOW(8))¶
             LayMinX%   = PEEKW(windLayer&+16)¶
             layMinY%   = PEEKW(windLayer&+18)¶
             layMaxX%   = PEEKW(windLayer&+20)¶
             layMaxY%   = PEEKW(windLayer&+22)¶
                          ¶
initSys:    '* Read System Parameters¶
```

```
                windAdd&       = WINDOW(7)¶
                scrAdd&        = PEEKL(windAdd&+46)¶
                scrBitMap&     = scrAdd&+184¶
                scrLayerInfo& = scrAdd&+224¶
                ¶
initSBMap:      '* Create Superbitmap¶
                opt&          = 2^0+2^1+2^16¶
                superBitMap& = AllocMem&(40,opt&)¶
                IF superBitMap& = 0 THEN¶
                  PRINT "Hmm. Not even 40 bytes, okay?"¶
                  ERROR 7¶
                END IF¶
                ¶
               '* ...and put it in use¶
                CALL InitBitMap(superBitMap&,scrDepth%,
superWidth%,superHeight%)¶
                superPlane&    = AllocRaster&(superWidth%,
superHeight%)¶
                IF superPlane& = 0 THEN¶
                  PRINT "No Room!"¶
                  CALL FreeMem(superBitMap&,40)¶
                  ERROR 7¶
                END IF¶
                POKEL superBitMap&+8,superPlane&¶
                ¶
               '* Open Superbitmap Layer¶
                superLayer&=CreateUpFrontLayer&(
scrLayerInfo&,scrBitMap&,LayMinX%,layMinY%,layMaxX%,layMa
xY%,superFlag%,superBitMap&)¶
                IF superLayer&=0 THEN¶
                  PRINT "No Layer Today!"¶
                  CALL FreeRaster(superPlane&,superWidth%,
superHeight%)¶
                  CALL FreeMem(superBitMap&,40)¶
                  ERROR 7¶
                END IF¶
               ¶
               '* ignore next line for now!¶
               '*** PUT EXPANSION HERE ***¶
                ¶
               '* new RastPort¶
                superRast& = PEEKL(superLayer&+12)¶
¶
prepare:       '* Setup Drawing Area¶
                CALL SetRast(superRast&,0)¶
               ¶
               '* Activate Layer¶
                POKEL superLayer&+40,WINDOW(7)¶
                backup.rast&  = PEEKL(superLayer&+12)¶
                backup.layer& = PEEKL(WINDOW(8))¶
                POKEL superLayer&+12,WINDOW(8)¶
                POKEL WINDOW(8),superLayer&¶
                           ¶
               '* Coordinates¶
                POKEW superRast&+34,&HAAAA ¶
                FOR loop%=0 TO superWidth% STEP 50¶
```

```
                LINE (loop%,0)-(loop%,superHeight%)¶
               NEXT loop%¶
               FOR loop%=0 TO superHeight% STEP 50¶
                 LINE (0,loop%)-(superWidth%,loop%)¶
               NEXT loop%¶
               POKEW superRast&+34,&HFFFF¶
¶
draw:         '* Here come the AmigaBASIC commands¶
               CIRCLE (400,200),250¶
               CIRCLE (400,200),300¶
               LINE   (200,100)-(600,300),1,bf¶
               ¶
scrollD:      '* scroll Display¶
               FOR loop%=0 TO 150¶
                 y% = 1¶
                 GOSUB scrollIt¶
               NEXT loop%¶
               ¶
               FOR loop%=0 TO 500¶
                 y% = 0¶
                 x% = 1¶
                 GOSUB scrollIt¶
               NEXT loop%¶
               ¶
               FOR loop%=0 TO 150¶
                 y% = -1¶
                 x% = -1¶
                 GOSUB scrollIt¶
               NEXT loop%¶
               ¶
               FOR loop%=0 TO 350¶
                 y% = 0¶
                 x% =-1¶
                 GOSUB scrollIt¶
               NEXT loop%        ¶
               ¶
deleteSys:    '* Delete System¶
               POKEL WINDOW(8),backup.layer&¶
               POKEL superLayer&+12,backup.rast&¶
               POKEL superLayer&+40,0¶
               ¶
               CALL DeleteLayer(scrLayerInfo&,superLayer&)¶
               CALL FreeRaster(superPlane&,superWidth%,
superHeight%)¶
               CALL FreeMem(superBitMap&,40)¶
               SCREEN CLOSE scrNr%¶
               WINDOW windNr%,"hi!",,,-1¶
               LIBRARY CLOSE¶
               END¶
                                  ¶
                                  ¶
scrollIt:     '* Scroll Function¶
               CALL
ScrollLayer(scrLayerInfo&,superLayer&,x%,y%)¶
               RETURN¶
```

This program creates a supergraphic with AmigaBASIC commands. It is scrolled back and forth across the screen. This program was adequate as a test. However, a complete drawing program that uses these layer techniques and many more exciting effects is included in Chapter 8. We have one more tip before we complete this chapter. With the routines we have demonstrated, you could use all the AmigaBASIC graphic commands in a layer. Two of these commands require careful handling. The CLS command erases only an area the size of a window's content in relation to the upper left hand corner of the layer. To erase the entire layer, you must use the graphic command SetRast. This command is called as follows:

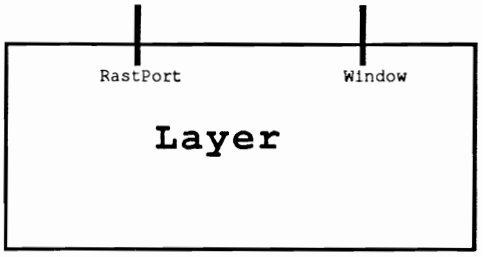
```
CALL SetRast(RastPort&,kolor%)
```

RastPort&:	RastPort address of your layer/window
kolor%:	The color, to fill the RastPort with. To erase it is normally =0.

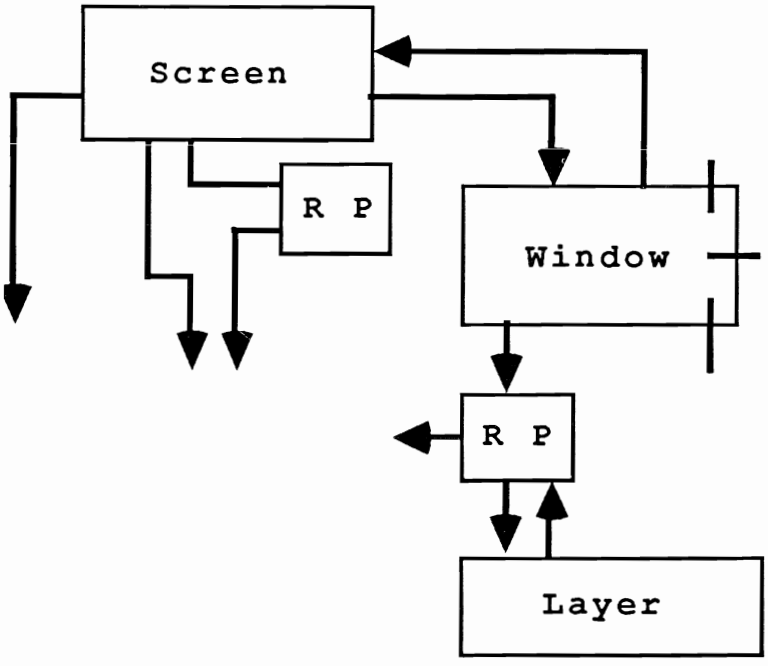
When you print (LOCATE instruction) text in a portion of the layer that is below your visible window, BASIC will scroll your output window and your layer will be moved up one line. To avoid this problem, use the graphic function Text (from Chapter 2) instead of PRINT.

4.8 Layers in the system

You probably recall our attempt in Section 4.3 to illustrate the system data structure. By now the layers should be familiar to you also. With all this accumulated knowledge, it is much easier to represent the system more accurately. First our layer:



...and here the entire system:



At this point, we have covered all the important system components. The BASIC elements of the graphic system are open to you. We will now move on to the subordinate data structures that, in some cases, are just as important.

5. The Amiga fonts

The Amiga is capable of using various character sets called fonts. Like all computers, the Amiga has a specific memory area for storing the shapes for the current font. However, unlike other computers, the Amiga contains two fonts that are built into the system. These fonts are:

topaz 8 and topaz 9

Which of these is the active font depends on your preference setting for the Workbench (60 or 80 character line width).

The Amiga, unlike other computers, has the option of loading other fonts from disk. Because of this capability, an unlimited supply of fonts are available for use with your projects.

It also possible for you to create your own fonts with no limits to your creativity.

Three paths are available for AmigaBASIC programming. Each of these methods will be explained in this chapter and many example programs will demonstrate how each function works.

5.1 The Amiga character generator

Before we can begin any type of project, we need an entry point into the font system of the Amiga. We can find this point in the RastPort of our windows. The address of the RastPort is always in the variable `WINDOW(8)` (see Section 3.6). At offset 52 is the starting address of the currently active font or, to be more precise, the starting address of a structure named `TextFont`. The following is the internal structure:

```
textFont&=PEEKL(WINDOW(8)+52)
```

Data structure TextFont/graphics/ 52 Bytes

Offset	Type	Description		
+ 000	—	Message structure		
+ 010	Long	Pointer to name string		
+ 020	Word	Height of font		
+ 022	Byte	Style of font		
		normal	= 0	
		underlined	= 1	= bit 0 = 1
		bold	= 2	= bit 1 = 1
		italic	= 4	= bit 2 = 1
		extended	= 8	= bit 3 = 1
+ 023	Byte	Preferences and flags		
		ROM-font	= 1	= bit 0 = 1
		Disk-font	= 2	= bit 1 = 1
		Rev-path	= 4	= bit 2 = 1
		Talldot	= 8	= bit 3 = 1
		Widedot	= 16	= bit 4 = 1
		Proportional	= 32	= bit 5 = 1
		Designed	= 64	= bit 6 = 1
		Removed	= 128	= bit 7 = 1
+ 024	Word	Width of character (average)		
+ 026	Word	Height of character without underline		
+ 028	Word	Smear effect for bold		
+ 030	Word	Access counter		
+ 032	Byte	ASCII code of first character		
+ 033	Byte	ASCII code of last character		
+ 034	Long	Pointer to font data		
+ 038	Word	Bytes per font line (modulo)		
+ 040	Long	Pointer to offset-data for character decoding		
+ 044	Long	Pointer to width table of font		
+ 048	Long	Pointer to font kern table		

This data structure contains all the parameters the Amiga needs to display a font. The following is a detailed explanation of all the data fields.

Offset 0: Message

Since a font operates independently from the other running tasks and simulates a stand alone program, it can use the message port techniques. This message structure keeps the font separate from the rest of the system.

Offset 10: Pointer to name string

This field is located inside the message structure and is a pointer to the name string of the font. The end of the name is indicated by a zero byte. You can determine the name of the current font with the following lines:

```
windo.rast&=WINDOW(8)
font.add&=PEEKL(windor.rast&+52)
font.name&=PEEKL(font.add&+10)

found%=PEEK(font.name&)
WHILE gefunden%)0
  font.name&=font.name&+1
  font.name$=font.name$+CHR$(found%)
  found%=PEEK(font.name&)
WEND

PRINT "Name of Font: ";font.name$
```

Offset 20 and 22: Font attribute

The characteristics of the font are stored here - the height in pixels and the style in bits.

Offset 23: Preferences and flags

The bits in these bytes represent the current status of the font. When searching for a font, you can set bits similar to the desired font and use them as a preference. This means that it's not necessary for the found font to have the exact settings but, normally, the settings will be as close as possible.

Offset 24 and 26: Further dimensions of the font

Offset 28: Counter for bold print

Normally, the Amiga moves the text one pixel to the right and prints it again when outputting bold text. This counter contains a value of how

many pixels to offset the text. Using larger values allows you to increase this width.

```
font&=PEEKL(WINDOW(8)+52)
POKE  WINDOW(8)+56,2 'Bold on
POKEW font&+28,3 'Offset 3 Pixels to the right
PRINT "Demo-Text"
POKE  WINDOW(8)+56,0 'normal
```

Offset 30: Access counter

When a font is opened, it becomes available to the entire system. So more than one program (task) can use the same font at once. Every task that opens a font for its own use must close it when finished. When a task opens a font that is already being used by another task, a new, memory intensive data structure is not opened. Instead, the second task simply accesses the existing data structure that was opened by the first task. At the same time, the access counter is incremented from one to two. If a task passes an assignment to close the font, the access counter is decreased by one. The counter must be equal to zero before the data structure is deleted from memory. This prevents the first task, which opened the font, from deleting this font while another task is using it.

Offset 32 and 33: ASCII codes

As you may know, the Amiga can provide up to 256 different characters in font. However, it isn't always necessary to define that many characters. For example, the alphabet only has 26 characters. Because of this, most fonts don't contain all 256 characters. Instead, these fonts simply specify a low and high limit for the defined characters. These limits are stored in these two fields.

Offset 34: The Font Data

This is a pointer to the character definitions for this font. We will cover the data block format later in this chapter.

Offset 38: Modulo

Modulo determines the number of bytes used per line of data block. The Amiga stores a font in rows of equal length. By using modulo, you can determine, for the current font, where the next row begins. There will be more on this subject later.

Offset 40: Data decoding

With data decoding it is possible to find the data, for a specific character, in the existing data lines. More information about this will be provided later.

Offset 44: Width table

The characters of a font are not necessarily the same width. A proportional font has a separate width for each character. For example, an "i" is narrower than a "W". This offset contains a pointer to a width table, which we will discuss later.

Offset 48: Font kern

This will be covered in detail in a later section.

5.2 Opening your first font

You have just learned about the innermost data structure of a font. Before we continue, we will introduce you to a much shorter data structure called `TextAtt`.

Data structure TextAttr/graphics/8 Bytes

Offset	Type	Description
+ 000	Long	Pointer to zero terminated name string
+ 004	Word	Height of font
+ 006	Byte	Style bits
+ 007	Byte	Preferences

For the definition of fields see Section 5.1

This structure helps you describe a font or specify a search description for it. The graphic routine `OpenFont` uses this data to find the specified font. We will try this out shortly. First, the syntax of the function `OpenFont` is:

```
newFont&=OpenFont&(textAttr&)
```

`textAttr&:` Starting address of the correctly filled `TextAttr` data structure (see above).

`newFont&:` When the font is successfully opened this is the starting address of the `TextFont` data structure of the new font (see Section 5.1).

The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 5.2¶
'# Program: Font Loader¶
'# Date:    04/10/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' Loads both ROM Fonts (TOPAZ8 and TOPAZ9)¶
¶
PRINT "Searching for .bmap file..."¶
```

```
¶
'GRAPHICS-Library¶
DECLARE FUNCTION OpenFont& LIBRARY¶
'CloseFont()¶
'SetFont()¶
¶
LIBRARY "graphics.library"¶
¶
demo:      '* Demonstrates both ROM Fonts¶
           demo.1$ = "TOPAZ 9 *** topaz 9 "¶
           demo.2$ = "TOPAZ 8 *** topaz 8 "¶
           CLS¶
           ¶
           FOR demo%=1 TO 10¶
             OpenFont 9¶
             FOR loop%=1 TO 10¶
               PRINT demo.1$;¶
             NEXT loop%¶
             PRINT¶
             ¶
             OpenFont 8¶
             FOR loop%=1 TO 10¶
               PRINT demo.2$;¶
             NEXT loop%¶
             PRINT¶
           NEXT demo%¶
           ¶
           LIBRARY CLOSE¶
           END¶
               ¶
SUB OpenFont(height%) STATIC¶
      font.name$   = "topaz.font"+CHR$(0)¶
      font.height% = height%¶
      font.stil%   = 0¶
      font.prefs%  = 0¶
      font.old&    = PEEKL(WINDOW(8)+52)¶
      ¶
     '* Fill TextAttr Structure¶
      textAttr&(0) = SADD(font.name$)¶
      textAttr&(1) = font.height%*2^16+font.stil%*2^4
+font.prefs%¶
      ¶
     '* Open New Font¶
      font.new& = OpenFont&(VARPTR(textAttr&(0)))¶
      IF font.new&<>0 THEN¶
        CALL CloseFont(font.old&)¶
        CALL SetFont(WINDOW(8),font.new&)¶
      END IF¶
END SUB¶
```

This program opens and uses both the ROM fonts, topaz8 and topaz9. If you attempt to enter a character height smaller than either font, it will not be accepted. Instead, one of the two ROM fonts will appear.

The program uses two additional library routines:

CloseFont() and SetFont()

Whenever you open a new font you must also close the old font so that the access counter in the TextFont structure contains the correct value (see Section 5.1). Use CloseFont to do this with an argument for the address of the TextFont structure of the old font. You can find this in your RastPort structure.

Just opening the font doesn't actually do anything. Much more is required in order to pass information about the new font to your RastPort. To accomplish this, use SetFont, which requires two arguments. These are the address of your RastPort and the address of the TextFont structure of a correctly opened font. These values are returned by the OpenFont function.

5.3 Accessing the disk fonts

The previous example program demonstrated that is not very difficult to activate a different font and that you can easily switch back and forth between them.

There isn't much difference in the appearance of the two fonts because they were designed to display 60 and 80 column text. In order to view a font that has a very different appearance, we have to use another method. On every Workbench disk there are numerous fonts available, which are stored in a subdirectory named "fonts". This directory should contain the following fonts:

Nr.	Height	Name
01	08	ruby.font
02	12	ruby.font
03	15	ruby.font
04	12	diamond.font
05	20	diamond.font
06	09	opal.font
07	12	opal.font
08	17	emerald.font
09	20	emerald.font
10	11	topaz.font
11	09	garnet.font
12	16	garnet.font
13	14	sapphire.font
14	19	sapphire.font

The two topaz fonts from the previous example are obviously not on the list. They are either loaded from the Kickstart disk or found in ROM.

Disk fonts cannot be activated by using `OpenFont` because this command only works with fonts currently in memory. Instead, we must use `OpenDiskFont` from the Diskfont.library. This is called in the same way as the `OpenFont` and also uses a TextAttr data structure.

The following program enables you to use disk fonts. Since it is designed as a first test, it is rather simple. The program requires the Workbench disk and the fonts found on this disk. The routine `OpenDiskFont` first searches for a font in the current directory and

then looks in the system directory FONTS. If the searched for font is not on the Workbench disk, the currently active font will not change.

The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 5.3¶
'# Program: Disk Font Loader¶
'# Date:    04/10/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' Loads a selected Disk Font¶
¶
PRINT "Searching for .bmap files..."¶
¶
'DISKFONT-Library¶
DECLARE FUNCTION OpenDiskFont& LIBRARY¶
¶
'GRAPHICS-Library¶
'CloseFont()¶
'SetFont()¶
¶
LIBRARY "diskfont.library"¶
LIBRARY "graphics.library"¶
¶
demo:     '* Demonstrates Disk-Fonts!¶
           demo.1$   = "DIAMOND 20 *** diamond 20 "¶
           demo.2$   = "SAPPHIRE 14 *** sapphire 14 "¶
           font.old& = PEEKL(WINDOW(8)+52)¶
           CLS¶
           ¶
           FOR demo%=1 TO 5¶
             OpenDiskFont "diamond",20¶
             FOR loop%=1 TO 5¶
               PRINT demo.1$;¶
             NEXT loop%¶
             PRINT¶
             ¶
             OpenDiskFont "Sapphire",14¶
             FOR loop%=1 TO 5¶
               PRINT demo.2$;¶
             NEXT loop%¶
             PRINT¶
           NEXT demo%¶
          ¶
          '* Activate normal Font¶
           font.new& = PEEKL(WINDOW(8)+52)¶
           CALL CloseFont(font.new&)¶
           CALL SetFont(WINDOW(8),font.old&)¶
```

```
          ¶
          LIBRARY CLOSE¶
          END¶
              ¶
SUB OpenDiskFont(n$,height%) STATIC¶
      font.name$   = n$+".font"+CHR$(0)¶
      font.height% = height%¶
      font.style%  = 0¶
      font.prefs%  = 0¶
      font.old&    = PEEKL(WINDOW(8)+52)¶
      ¶
     '* Fill TextAttr Structure¶
      textAttr&(0) = SADD(font.name$)¶
      textAttr&(1) = font.height%*2^16+font.style%*2^4+
font.prefs%¶
      ¶
     '* Open New Font¶
      font.new& = OpenDiskFont&(VARPTR(textAttr&(0)))¶
      IF font.new&<>0 THEN¶
        CALL CloseFont(font.old&)¶
        CALL SetFont(WINDOW(8),font.new&)¶
      END IF¶
END SUB¶
```

You can clearly see that every time the demo prints a line of text, the Amiga starts loading a font again. This makes sense because each time you switch fonts, the old font is deleted from RAM. Obviously this isn't very practical. You can prevent this by leaving the often-used fonts open and then closing all the fonts when the program has ended. When doing this, it is important to remember that: 1.) all the fonts that are opened by your program must also be closed by your program and 2.) a font can only be loaded once (otherwise you will waste useful memory space.) The following program uses this method. It can activate both disk and ROM fonts and loads a disk font only once. This makes the program much faster and more efficient Here is the listing:

The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 5.3B¶
'# Program: Font Load & Hold¶
'# Date:    04/10/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' Loads Disk and RAM/ROM Fonts. Whenever a new Font is¶
' loaded, the program does not close the old Font, but¶
' adds the new Font to a list. The next time this Font¶
```

```
' is opened, it is already in RAM and appears
immediately.¶
' At the end of the program all Fonts are closed at the¶
' same time.¶
¶
PRINT "Searching for .bmap files..."¶
¶
'DISKFONT-Library¶
DECLARE FUNCTION OpenDiskFont& LIBRARY¶
¶
'GRAPHICS-Library¶
DECLARE FUNCTION OpenFont& LIBRARY¶
'CloseFont()¶
'SetFont()¶
¶
LIBRARY "diskfont.library"¶
LIBRARY "graphics.library"¶
¶
init:     '* Dimension storage fields¶
           DIM SHARED storage&(30) 'max. 30 different
types¶
           CLS¶
           ¶
demo:     '* Here we go:¶
           LOCATE 3,1¶
           OpenPrgFont "Opal",12¶
           PRINT "Working with Amiga Fonts!"¶
           OpenPrgFont "Diamond",12¶
           ¶
           WHILE z$<>"end"¶
             LINE INPUT "Name of Font: ";z$¶
             IF z$<>"end" THEN¶
               INPUT "Height";h%¶
               OpenPrgFont z$,h%¶
               PRINT "This is ";z$;", ";h%;" Points
high."¶
               PRINT "Enter 'end' to Exit!"¶
               PRINT "Opened Fonts: ";counter%¶
               OpenPrgFont "opal",12¶
             END IF¶
           WEND¶
               ¶
          '* Activate normal Font¶
          '* and delete all others from RAM¶
           ClosePrgFont¶
                      ¶
           LIBRARY CLOSE¶
           END¶
              ¶
SUB OpenPrgFont(n$,height%) STATIC¶
      SHARED counter%,mode%,font.original&¶
      ¶
      IF mode% = 0 THEN¶
        mode%          = 1¶
        font.original& = PEEKL(WINDOW(8)+52)¶
      END IF¶
```

```
        ¶
      font.name$    = n$+".font"+CHR$(0)¶
      part2$        = RIGHT$(font.name$,LEN(font.name$)-
1)¶
      part1%        = ASC(LEFT$(font.name$,1))¶
      part1%        = part1% OR 32¶
      ¶
      font.name$    = CHR$(part1%)+part2$¶
      font.height%  = height%¶
      font.style%   = 0¶
      font.prefs%   = 0¶
¶
     '* Fill TextAttr Structure¶
      textAttr&(0) = SADD(font.name$)¶
      textAttr&(1) = font.height%*2^16+font.style%*2^4+
font.prefs%¶
      ¶
     '* New Font in RAM?¶
      font.new&     = OpenFont&(VARPTR(textAttr&(0)))¶
      IF font.new&<>0 THEN¶
       '* Yes, there is a new Font in RAM with that name¶
        test.height%=PEEKW(font.new&+20)¶
        CALL CloseFont(font.new&)¶
        IF test.height%<>font.height% THEN¶
         '* not as high as the one we want¶
         '* so search again¶
          font.new&=0¶
        END IF¶
      END IF¶
          ¶
     '* Open New Font¶
      IF font.new& = 0 THEN¶
       '* Look on the Disk (last chance...)¶
        font.new&   = OpenDiskFont&(VARPTR(textAttr&(0)))¶
        IF font.new&<>0 THEN¶
         '* found!¶
          counter%           = counter%+1¶
          storage&(counter%) = font.new&¶
        END IF ¶
      END IF¶
      IF font.new&<>0 THEN¶
       '* New Font to be activated¶
        CALL SetFont(WINDOW(8),font.new&)¶
      END IF        ¶
END SUB         ¶
      ¶
SUB ClosePrgFont STATIC¶
      SHARED counter%,font.original&¶
      ¶
      FOR loop%=1 TO counter%¶
        IF storage&(loop%)<>0 THEN¶
          CALL CloseFont(storage&(loop%))¶
        ELSE¶
          ERROR 255¶
        END IF¶
        storage&(loop%) = NULL¶
```

```
      NEXT loop%¶
      ¶
      CALL SetFont(WINDOW(8),font.original&)¶
END SUB¶
```

The variables `counter%` and `mode%` are reserved for the SUB programs and must not be changed or deleted anywhere in the program.

The SUB `OpenPrgFont` lets you select any desired font:

```
OpenPrgFont name$,height%
```

`name$:`	Name of font
`height%:`	Height of font

Next, the SUB makes the variable `mode%`. If `mode%` equals zero then an alternate font is not loaded. In our program, the pointer font `original&` is set for the original font. With this pointer and some experimentation, you can always return to the orginal font.

At the end of the SUB, a TextAttr structure is created. The statement `UCASE$` converts the search fonts name to uppercase. The routine `OpenFont` handles upper and lower case letters differently, which means that it is case sensitive. So when the diamond font is in RAM, it can't be located using the name "diamond". We must use uppercase letters instead.

Finally, the structure is initialized. The variable `textAttr&` is used for storage.

To avoid loading the desired font from disk, we check for it in RAM instead. When the routine `OpenFont` returns a pointer the desired font exists in memory. The height of the RAM font is stored in the variable `test.height%` for later comparison. The RAM font is then closed with `CloseFont`. This process is both important and necessary because, if the font already exists in RAM, we have already opened it with `OpenDiskFont`. Since we used `OpenFont`, we need to prevent the access counter from being incremented, so we close the font again. However, the font is not really closed; only our access entry for the `OpenFont` command is removed. When the RAM font isn't the one we are searching for, we have to close it anyway.

Now we compare the height of the found font with the height of the font for which we are looking. When they match, the pointer to the RAM font in `font.new&` remains and if they do not match, `font.new&` is deleted.

When `font.new&` is deleted, we look on the disk for the desired font. If it is found, `OpenDiskFont` loads the font into RAM. Since we

loaded a new font, we must be able to delete it at the end of the program. In order to do this, we must store the address of the font in a field of `storage&` and raise our array pointer.

To complete this process, we activate the new font. This happens only when `font.new&` is not equal to zero. When a font can't be found in RAM, ROM or on the disk, the variable `font.new&` is always equal to zero.

A call to `ClosePrgFont` must be at the end of your program. This reads through the fields of the `storage&` array and calls `CloseFont` for each font. Once this occurs, the memory is returned to the system.

5.4 The font menu

With the information and programs from the previous chapter, you are almost ready to work with Amiga fonts. So far you can load and use fonts but only if you already know the name and height of the desired font. Our next project will show you how to choose fonts from a menu.

You could put the names of all available fonts into the `DATA` statements of a program. However, this wouldn't be very useful because fonts on disk can be deleted or added. For this reason, there is the function `AvailFonts` in the Diskfont library. This routine assembles a list of the currently available fonts. The call to the `AvailFonts` routine looks like this:

```
status%=AvailFonts%(buffer&,buflen&,mode%)
```

`buffer&:`	Address to a free memory buffer
`buflen&:`	Size of buffer
`mode%:`	1=RAM/ROM
	2=DISK
	3=any source
`status%:`	0=all ok
	otherwise the number of bytes for the buffer is not enough

Depending on the mode variable, `AvailFonts` fills the buffer with the following information:

Offset	Type	Description
+ 000	Word	Number of following entries
(The next five entries are repeated as required)		
+ 002	Word	ID (1=RAM/ROM, 2=disk)
+ 004	Long	Pointer to name string
+ 008	Word	Height of font
+ 010	Byte	Style bits
+ 011	Byte	Preferences

This information is used by the `AvailFonts` to provide data on all found fonts.

The following program contains the SUB program `GenerateMenu`.

The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 5.4¶
'# Program: Menu Managed Fonts¶
'# Date:    04/10/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' automatically builds a menu for all useable¶
' Fonts and then controls the Menu¶
¶
PRINT "Searching for .bmap files..."¶
¶
'EXEC-Library¶
DECLARE FUNCTION AllocMem& LIBRARY¶
'FreeMem()¶
¶
'DISKFONT-Library¶
DECLARE FUNCTION OpenDiskFont& LIBRARY¶
DECLARE FUNCTION AvailFonts% LIBRARY¶
¶
'GRAPHICS-Library¶
DECLARE FUNCTION OpenFont& LIBRARY¶
'CloseFont()¶
'SetFont()¶
¶
LIBRARY "diskfont.library"¶
LIBRARY "graphics.library"¶
LIBRARY "exec.library"¶
¶
init:      '* Dimension Storage Fields¶
           DIM SHARED storage&(30) 'max. 30 different
Types¶
           DIM SHARED font.title$(19)¶
           DIM SHARED font.height%(19)¶
           CLS¶
           ¶
demo:      '* Here we go:¶
           MENU 5,0,1,"Font"¶
           MENU 5,1,1,"Load"¶
           MENU 6,0,1,"Service"¶
           MENU 6,1,1,"Quit    "¶
           ¶
           menu.old% = 1¶
           ¶
           ON MENU GOSUB menucheck¶
           MENU ON¶
           ¶
           PRINT "The Menu is now ready to use."¶
```

```
                PRINT "Pick a Font of your choice, or"¶
                PRINT "'LOAD', to choose from Menu."¶
                ¶
                WHILE forever=forever¶
                WEND¶
                ¶
menucheck:'* Menu Handling¶
                ¶
                menuId   = MENU(0)¶
                menuItem = MENU(1)¶
                ¶
                IF menuId=5 THEN¶
                  IF menu.old%=1 THEN¶
                    menu.old% = 0¶
                    menu.nr%  = 5¶
                    modeALL%  = 3¶
                    GenerateMenu menu.nr%,modeALL%¶
                    PRINT "Menu is ready!"¶
                  ELSE¶
                    ft$ = font.title$(menuItem-1)¶
                    fh% = font.height%(menuItem-1)¶
                     ¶
                    OpenPrgFont ft$,fh%¶
                  END IF¶
                ELSEIF menuId=6 THEN¶
                  GOTO endprg¶
                END IF¶
                ¶
                GOSUB ShowText¶
                ¶
                RETURN¶
                ¶
ShowText: '* Display Text¶
                PRINT ft$;fh%;" Points - TEXT SAMPLE *** text
sample"¶
                RETURN¶
                                ¶
                                 ¶
endprg:   '* Activate normal Font¶
                '* Delete all others from RAM¶
                 ClosePrgFont¶
                            ¶
                 LIBRARY CLOSE¶
                 END¶
¶
SUB GenerateMenu(menu.nr%,mode%) STATIC¶
      mem.opt&        = 2^0+2^16¶
      buffer.size&    = 3000¶
      buffer.add&     = AllocMem&(buffer.size&,mem.opt&)¶
      IF buffer.add&<>0 THEN¶
        status% =
AvailFonts%(buffer.add&,buffer.size&,mode%)¶
        IF status% = 0 THEN¶
          entry%   = PEEKW(buffer.add&)¶
          IF entry%>18 THEN entry%=18¶
          FOR loop% = 0 TO entry%-1¶
```

```
                counter%     = loop%*10¶
                font.name&   = PEEKL(buffer.add&+4+counter%)¶
                font.height% = PEEKW(buffer.add&+8+counter%)¶
                font.name$   = ""¶
                check%       = PEEK(font.name&)¶
                WHILE check%<>ASC(".")¶
                  font.name$ = font.name$+CHR$(check%)¶
                  font.name& = font.name&+1¶
                  check%     = PEEK(font.name&)¶
                WEND¶
                font.title$(loop%)  = font.name$¶
                font.height%(loop%) = font.height%¶
                menu.name$          = UCASE$(font.name$+STR$(
font.height%))¶
                MENU CSNG(menu.nr%),CSNG(loop%+1),
1,menu.name$¶
              NEXT loop%¶
              CALL FreeMem(buffer.add&,buffer.size&)¶
            END IF¶
          ELSE¶
            BEEP¶
          END IF¶
END SUB           ¶
                ¶
SUB OpenPrgFont(n$,height%) STATIC¶
          SHARED count%,mode%,font.original&¶
          ¶
          IF mode%=0 THEN¶
            mode%          = 1¶
            font.original& = PEEKL(WINDOW(8)+52)¶
          END IF¶
            ¶
          font.name$  = n$+".font"+CHR$(0)¶
          part2$      = RIGHT$(font.name$,LEN(font.name$)-1)¶
          part1%      = ASC(LEFT$(font.name$,1))¶
          part1%      = part1% OR 32¶
¶
          font.height% = height%¶
          font.stil%  = 0¶
          font.prefs% = 0¶
¶
         '* Fill TextAttr Structure¶
          textAttr&(0) = SADD(font.name$)¶
          textAttr&(1) = font.height%*2^16+font.stil%*2^4+
font.prefs%¶
          ¶
         '* New Font in RAM?¶
          font.new& = OpenFont&(VARPTR(textAttr&(0)))¶
          IF font.new&<>0 THEN¶
           '* yes, a font by that name is in RAM¶
            test.height% = PEEKW(font.new&+20)¶
            CALL CloseFont(font.new&)¶
            IF test.height%<>font.height% THEN¶
             '* but it's not as tall as the one searched for¶
             '* so search again¶
              font.new& = 0¶
```

```
        END IF¶
      END IF¶
        ¶
    '* Open New Font¶
      IF font.new& = 0 THEN¶
       '* Look on Disk (last chance...)¶
        font.new&  = OpenDiskFont&(VARPTR(textAttr&(0)))¶
        IF font.new&<>0 THEN¶
         '* found!¶
          count%           = count%+1¶
          storage&(count%) = font.new&¶
        END IF ¶
      END IF¶
      IF font.new&<>0 THEN¶
       '* New Font to be Activated¶
        CALL SetFont(WINDOW(8),font.new&)¶
      END IF        ¶
END SUB         ¶
      ¶
SUB ClosePrgFont STATIC¶
      SHARED count%,font.original&¶
      ¶
      FOR loop%=1 TO count%¶
        IF storage&(loop%)<>0 THEN¶
          CALL CloseFont(storage&(loop%))¶
        ELSE¶
          ERROR 255¶
        END IF¶
        storage&(loop%) = NULL¶
      NEXT loop%¶
      ¶
      IF font.original&<>0 THEN¶
        CALL SetFont(WINDOW(8),font.original&)¶
      END IF¶
END SUB¶
```

The call to this SUB program looks like this:

```
GenerateMenu menu.nr%,mode%
```

menu.nr%:	Number of the menu to generate
mode%:	1 = RAM/ROM font
	2 = Disk font
	3 = All fonts

After calling this SUB program two things happen:

a) A menu is created. This menu contains the names and height of the (depending on the mode selected) available fonts within your Y measurement.

b) Two data fields are initialized: `font.title$` contains the names of the fonts and `font.height%` contains their Y measurement.

By using the menu, you can select one of the available fonts. You do not have to know which fonts are on the disk. Once a font is selected, the name and height of the selected font are passed using the existing variables. `OpenPrgFont` then takes these variables and does the rest of the work.

5.5 Designing your own fonts

By now you should know enough about the standard Amiga fonts. We will now move on to the last and most difficult section: defining a completely original font.

In order to do this, you need to know about the construction of a font. At the beginning of this chapter we introduced you to the data structure named `TextFont`. Before continuing with this data structure, we will present some of the peculiarities of an Amiga font.

Basically, there are two different font forms for the Amiga:

a) Normal fonts

b) Proportional fonts

A typical font consists of characters that have equal height and width. Proportional font characters also have equal height but have individual widths.

You need a maximum of four memory blocks to define a font.

```
charData
charLoc
charSpace
charKern
```

`charData` contains the actual definition of a character in the font. This refers to the bit-packed character information. Since an Amiga character can be any pixel width that you select, it is wasteful to store the data for every character in byte form. The Amiga stores the character data as follows:

Let's start with the two characters below. A "." represents an unset pixel and a "*" represents a set pixel.

```
....*....      ****.
...*.*...      *...*
..*...*..      *...*
.*.....*.      ****.
*********      *...*
*.......*      *...*
*.......*      ****
```

These two characters have different widths and could be from a proportional font. The Amiga creates a bit-row, one after the other, of all the characters in a font. If our font contained only the characters above, then `charData` would look like this:

```
1. row:  ....*....****.
2. row:  ...*.*...*...*
3. row:  ..*...*..*...*
4. row:  .*.....*.****.
5. row:  **********...*
6. row:  *.......**...*
7. row:  *.......*****.
```

The individual rows are then written one after the other into the memory block:

```
charData: ....*....****....*.*...*...*..*...*..*...* etc.
```

This storage method is very efficient but has one problem. We need to get the characters out of the bits. This is why there is a memory block, called `charLoc`. This block contains two words (two byte fields) for every character in the font. The first word contains the bit count from the start of a row to the bit information for the character. The second word contains the bit count of the character. For the two characters in our example, it looks like this:

```
charLoc: 0,9, 9,5
```

The first character begins zero bits from the beginning of the data row and is nine bits (pixels) wide. The second character begins nine bits after the start of the data row and is five bits wide. This is the definition for all seven rows of our characters.

Another problem is getting from one `charData` row to the next. The field `modulo` from the `TextFont` structure is used to do this. The length, in bytes, of the data row is stored here. To get your next row, add this value to the current address of the data row.

The data field `charData` contains only the basic information for the current character. So, to separate the characters on the screen, we need to insert some space between them. The field `charSpace` contains the real character width, in pixels. For example:

```
charSpace: 11,7
```

The first character is eleven pixels wide and the second is seven.

There is one last problem. `charSpace` sets the actual character width. For the first character in our example, this is eleven pixels.

```
..........
..........
..........
..........
..........
..........
..........
```

This is a problem because the character itself is only nine pixels wide. So the position at which this field should begin displaying the character is still unknown. The block `charKern` contains, for every character, a word that defines, in bits, the distance from the left edge of the field where the character should be displayed. Again an example:

```
charKern: 1,2
```

This displays our characters on the screen like this:

```
.....*.....      ..****.
....*.*....      ..*...*
...*...*...      ..*...*
..*.....*..      ..****.
.*********.      ..*...*
.*.......*.      ..*...*
.*.......*.      ..****.
```

Space = 11 Space = 7
Kern = 1 Kern = 2
Width = 9 Width = 5

5.5.1 Reading the font generator

By using the information provided above, we can access an existing font and read it. You can also get the data for a specific character and display it.

The address of the `TextFont` structure for a font that is currently open is found in the RastPort.

```
font&=PEEKL(WINDOW(8)+52)
```

You will find the required pointer to the memory block there (see Section 5.1):

Here is the font reader program. The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 5.5.1 ¶
'# Program: Font Reader¶
'# Date:    04/11/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' Reads the currently active font and displays the¶
' decoded information in different sized¶
' variations.¶
¶
PRINT "Searching for .bmap file..."¶
¶
'GRAPHICS-Bibliothek¶
DECLARE FUNCTION OpenFont& LIBRARY¶
'SetFont()¶
'CloseFont()¶
¶
LIBRARY "graphics.library"¶
¶
init:       '* Variables¶
            DIM SHARED char$(256)¶
            g% = 12  'Box size¶
            CLS¶
            ¶
            FOR demo%=32 TO 255¶
              Matrix demo%¶
              CLS¶
              TopazON¶
              PRINT "Character: ASCII ";demo%;" =
";CHR$(demo%)¶
              FOR show% = 1 TO height%¶
                LOCATE show%+2,1¶
                PRINT char$(show%)¶
                FOR ex% = 1 TO LEN(char$(show%))¶
                  z$ = MID$(char$(show%),ex%,1)¶
                  IF z$ = "*" THEN¶
                    kolor% = 2¶
                  ELSEIF z$ = "." THEN¶
                    kolor% = 1¶
                  ELSEIF z$ = "," THEN¶
                    kolor% = 3¶
                  END IF¶
                  LINE (300+ex%*g%,show%*g%)-
(300+ex%*g%+g%,show%*g%+g%),kolor%,bf¶
                  LINE (500+ex%*2,show%*2)-
(500+ex%*2+2,show%*2+2),kolor%,bf¶
              NEXT ex%¶
```

```
                NEXT show%¶
                TopazOFF¶
              NEXT demo%          ¶
                ¶
              END¶
¶
SUB Matrix(code%) STATIC¶
              SHARED height%¶
              ¶
              f.1%       = 0¶
              f.2%       = 0¶
              font&      = PEEKL(WINDOW(8)+52)¶
              charData&  = PEEKL(font&+34)¶
              charLoc&   = PEEKL(font&+40)¶
              charSpace& = PEEKL(font&+44)¶
              charKern&  = PEEKL(font&+48)¶
              modulo%    = PEEKW(font&+38)¶
              ¶
              IF charSpace& = 0 THEN f.1%=1¶
              IF charKern&  = 0 THEN f.2%=1¶
              ¶
              height%   = PEEKW(font&+20)¶
              ¶
              loASCII% = PEEK(font&+32)¶
              hiASCII% = PEEK(font&+33)¶
              ¶
              IF code%<loASCII% OR code%>hiASCII% THEN¶
                PRINT "ASCII Code";code%;" not in Font"¶
              END IF¶
              ¶
             '* Decoding information¶
              offset%       = code%-loASCII%¶
              offset.bit&   = PEEKW(charLoc&+4*offset%)¶
              offset.byte%  = INT(offset.bit&/8)¶
              offset.bit%   = offset.bit&-(8*offset.byte%)¶
              char.width%   = PEEKW(charLoc&+4*offset%+2)¶
              IF f.1% = 0 THEN¶
                char.space%=PEEKW(charSpace&+2*offset%)¶
              END IF¶
              IF f.2% = 0 THEN¶
                char.kern% = PEEKW(charKern&+2*offset%)¶
              END IF¶
              ¶
             '* Read Data¶
              FOR loop1% = 1 TO height%¶
                char$(loop1%) = ""¶
                IF f.2% = 0 THEN¶
                  IF char.kern%>0 THEN¶
                    char$(loop1%)=STRING$(char.kern%,",")¶
                  END IF¶
                END IF¶
                linedata&  = PEEK(charData&+offset.byte%)¶
                counter%   = 7-offset.bit%¶
                FOR loop2% = 1 TO char.width%¶
                  IF (linedata& AND 2^counter%)<>0 THEN¶
                    linedata&     = linedata&-2^counter%¶
```

```
                    char$(loop1%) = char$(loop1%)+"*"¶
                  ELSE¶
                    char$(loop1%) = char$(loop1%)+"."¶
                  END IF¶
                  counter%        = counter%-1¶
                  IF counter%<0 THEN¶
                    offset.long% = offset.long%+1¶
                    linedata&    = PEEK(charData&+
offset.byte%+offset.long%)¶
                    counter%     = 7¶
                  END IF¶
                NEXT loop2%¶
                offset.long% = 0¶
                charData&    = charData&+modulo%¶
                IF f.2%=0 THEN¶
                  char.diff% = char.space%-char.width%-
char.kern%¶
                ELSEIF f.2%=0 THEN¶
                  char.diff% = char.space%-char.width%¶
                END IF¶
                IF char.diff%>0 THEN¶
                  char$(loop1%)=char$(loop1%)+
STRING$(char.diff%,",")¶
                END IF¶
              NEXT loop1%  ¶
END SUB¶
             ¶
SUB TopazON STATIC¶
      SHARED font&,font.old&¶
      font$         = "topaz.font"+CHR$(0)¶
      textAttr&(0) = SADD(font$)¶
      font.old&     = PEEKL(WINDOW(8)+52)¶
      font&         = OpenFont&(VARPTR(textAttr&(0)))¶
      CALL SetFont(WINDOW(8),font&)¶
END SUB¶
¶
SUB TopazOFF STATIC¶
      SHARED font&,font.old&¶
      CALL CloseFont(font&)¶
      CALL SetFont(WINDOW(8),font.old&)¶
END SUB¶
```

This program reads the currently active font. In most cases, this is one of the two ROM fonts. To see something really interesting, first load one of the disk fonts like sapphire.

All the possible characters of the font are represented on the screen in three ways: 1.) using "*" and ".", 2.) as large graphics, 3.) as small graphics.

The graphics use three different colors. The data area defined by `charData` is white and all the set pixels are black. The additional pixels defined by `charSpace` and `charKern` are displayed in orange.

However, proportional fonts will not display any orange because they do not use charSpace and charKern.

The following is information about the program:

The heart of the program is the SUB program matrix. This routine handles the difficult task of reading the font but can also be used for other purposes. Here is the call:

```
Matrix ascii.code%
```

ascii.code%: ASCII code of character (0=255)

or

```
Matrix CINT(ASC(z$))
```

z$: The desired character

In addition, there are the SUB programs TopazON and TopazOFF, which switch the currently active system font on and off. By using them, you are able to display comments between the font data that are independent of the active font being read and displayed.

An explanation of how the SUB Matrix program functions is found in Section 5.5.

5.5.2 Big Text: enlarging text

This application demonstrates the adaptability of the read routine matrix, from our previous example. Matrix helps the SUB BigText create text of any desired size on the screen.

This is the call:

```
BigText text$,size%,kolor%
```

text$: Text to display
size%: Enlarging factor (1-...)
kolor%: Text color

The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 5.5.2¶
'# Program: Enlarge Text¶
'# Date:    04/11/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' Enlarges any desired text. The Text is enlarged¶
' in the style of the currently active Font.¶
¶
init:       '* Variable¶
            DIM SHARED char$(256)¶
            BigText "Hello",15,2¶
            BigText "Commodore AMIGA",4,3 ¶
            LOCATE 3,1¶
            BigText "small",1,1¶
            BigText "larger",2,1¶
            BigText "Larger Yet!",3,1¶
            BigText "GIGANTIC!",8,3¶
            END¶
            ¶
SUB BigText(text$,size%,kolor%) STATIC¶
            SHARED height%,char.kern%,char.width%¶
            SHARED char.space%¶
¶
            o.xo% = 0¶
            z.x%  = PEEKW(WINDOW(8)+58)¶
            z.y%  = PEEKW(WINDOW(8)+58)¶
            y%    = CSRLIN*z.y%¶
            x%    = POS(0)*z.x%¶
            ¶
            FOR loop1%=1 TO LEN(text$)¶
              z$    = MID$(text$,loop1%,1)¶
              Matrix CINT(ASC(z$))¶
              o.xo% = o.xo%+char.kern%*size%¶
              FOR loop2%=1 TO height%¶
                FOR loop3%=1 TO LEN(char$(loop2%))¶
                  m$=MID$(char$(loop2%),loop3%,1)¶
                  IF m$="*" THEN¶
                    o.x% = x%+o.xo%+loop3%*size%¶
                    o.y% = y%+loop2%*size%¶
                    LINE (o.x%,o.y%)-
(o.x%+size%,o.y%+size%),kolor%,bf¶
                  END IF¶
                NEXT loop3%¶
              NEXT loop2%¶
              rest% = char.space%-char.width%-char.kern%¶
              IF rest%<0 THEN rest%=0¶
              o.xo% = o.xo%+char.width%*size%+
rest%*size%¶
            NEXT loop1%¶
            PRINT¶
END SUB                    ¶
```

```
                                ¶
SUB Matrix(code%) STATIC¶
              SHARED height%,char.kern%,char.width%¶
              SHARED char.space%¶
              f.1%       = 0¶
              f.2%       = 0¶
              font&      = PEEKL(WINDOW(8)+52)¶
              charData&  = PEEKL(font&+34)¶
              charLoc&   = PEEKL(font&+40)¶
              charSpace& = PEEKL(font&+44)¶
              charKern&  = PEEKL(font&+48)¶
              modulo%    = PEEKW(font&+38)¶
              ¶
              IF charSpace& = 0 THEN f.1%=1¶
              IF charKern&  = 0 THEN f.2%=1¶
              ¶
              height%  = PEEKW(font&+20)¶
              ¶
              loASCII% = PEEK(font&+32)¶
              hiASCII% = PEEK(font&+33)¶
              ¶
              IF code%<loASCII% OR code%>hiASCII% THEN¶
                PRINT "ASCII Code";code%;" not in Font"¶
              END IF¶
              ¶
             '* Decoding information¶
              offset%        = code%-loASCII%¶
              offset.bit&    = PEEKW(charLoc&+4*offset%)¶
              offset.byte%   = INT(offset.bit&/8)¶
              offset.bit%    = offset.bit&-
(8*offset.byte%)¶
              char.width%=PEEKW(charLoc&+4*offset%+2)¶
              z.b%           = char.width%¶
              IF f.1% = 0 THEN¶
                char.space% = PEEKW(charSpace&+2*offset%)¶
                z.b%        = char.space%¶
              END IF¶
              IF f.2% = 0 THEN¶
                char.kern%  = PEEKW(charKern&+2*offset%)¶
              END IF¶
              ¶
             '* Read¶
              FOR loop1% = 1 TO height%¶
                char$(loop1%) = ""¶
                IF f.2% = 0 THEN¶
                  IF char.kern%>0 THEN¶
                    char$(loop1%)=STRING$(char.kern%,",")¶
                  END IF¶
                END IF¶
                linedata&  = PEEK(charData&+offset.byte%)¶
                counter%   = 7-offset.bit% ¶
                FOR loop2% = 1 TO char.width%¶
                  IF (linedata& AND 2^counter%)<>0 THEN¶
                    linedata&     = linedata&-2^counter%¶
                    char$(loop1%) = char$(loop1%)+"*"¶
                  ELSE¶
```

```
                    char$(loop1%) = char$(loop1%)+"."¶
                  END IF¶
                  counter% = counter%-1¶
                  IF counter%<0 THEN¶
                    offset.long% = offset.long%+1¶
                    linedata&    = PEEK(charData&+
offset.byte%+offset.long%)¶
                    counter%     = 7¶
                  END IF¶
                NEXT loop2%¶
                offset.long% = 0¶
                charData&    = charData&+modulo%¶
                IF f.2% = 0 THEN¶
                  char.diff% = char.space%-char.width%-
char.kern%¶
                ELSEIF f.2%=0 THEN¶
                  char.diff% = char.space%-char.width%¶
                END IF¶
                IF char.diff%>0 THEN¶
                  char$(loop1%) =
char$(loop1%)+STRING$(char.diff%,",")¶
                END IF¶
              NEXT loop1%  ¶
END SUB¶
```

The matrix routine had to be changed slightly so that `BigText` could position the text properly. In order to do this, we added a few more parameters to the `SHARED` assignment at the beginning of the SUB.

5.5.3 A fixed width font generator

Now that you are more comfortable with the pointers and contents of a font, we will proceed to our main project, a character generator.

You have seen how difficult it is to define and manage a proportional font. Because of this, our first character generator is a fixed width generator which creates characters with identical widths.

To further simplify this process, our characters will have a set size of 8x8 pixels like the ROM font "topaz 8". Since this size has a one byte offset, it's easy to store and handle the character data in `charData` .

Before we discuss the details of the program, here is the program listing. The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 5.5.3¶
'# Program: Fixed-Width Font Generator¶
'# Date:    04/12/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' This program makes possible the creation of a¶
' different Font. Every character has a set size¶
' of 8x8 pixels. Each character can be freely defined as¶
' desired. All undefined Characters will default to the¶
' standard ROM Font (topaz 8). Unavailable characters¶
' will be indicated by the 'unprintable
character'symbol.¶
¶
PRINT "Searching for .bmap file..."¶
¶
'GRAPHICS-Library¶
DECLARE FUNCTION OpenFont& LIBRARY¶
'CloseFont()¶
'SetFont()¶
'AddFont()¶
¶
'EXEC-Library¶
DECLARE FUNCTION AllocMem& LIBRARY¶
'FreeMem()¶
'CopyMem()¶
¶
LIBRARY "graphics.library"¶
LIBRARY "exec.library"¶
¶
init:     CLS¶
         '* Generate character set¶
         '* Call:¶
         '* MakePrgFont "name",asciiLo%,asciiHi%¶
          ¶
          MakePrgFont "tobi",22,200¶
          MakePrgFont "ralfi",60,122¶
          ¶
         '* Call:¶
         '* ActivateFont "name"¶
          ¶
          ActivateFont "tobi"¶
          ¶
         '* Define new Font¶
         '* Call:¶
         '* NewD "char",row%,"definition"¶
         '* row%: 0...7 definition: *=set pixel¶
         ¶
          NewD "A",0,"......*."¶
          NewD "A",1,".....**."¶
          NewD "A",2,"....***."¶
          NewD "A",3,"...*.**."¶
```

```
          NewD "A",4,"..*****."¶
          NewD "A",5,".*....*."¶
          NewD "A",6,"***..***"¶
          NewD "A",7,""¶
          ¶
          ActivateFont "ralfi"¶
         '* second character using byte value method
(faster)¶
          NewB "@",0,126¶
          NewB "@",1,129¶
          NewB "@",2,157¶
          NewB "@",3,161¶
          NewB "@",4,161¶
          NewB "@",5,157¶
          NewB "@",6,129¶
          NewB "@",7,126¶
          ¶
         '* Sample Text¶
          ActivateFont "tobi"¶
          PRINT "@ 1989 Abacus - Amiga Graphics Inside &
Out"¶
          PRINT "       ^         ^"¶
          ActivateFont "ralfi"¶
          PRINT "@ 1989 Abacus - Amiga Graphics Inside &
Out"¶
          PRINT "^"¶
          ¶
         '* Delete Font¶
         '* Call:¶
         '* DeleteFont "name"¶
          ¶
          DeleteFont "tobi"¶
          DeleteFont "ralfi"¶
          END¶
¶
SUB ActivateFont(z.n$) STATIC¶
      z.name$   = UCASE$(z.n$+".font"+CHR$(0))¶
      t&(0)     = SADD(z.name$)¶
      t&(1)     = 8*2^16¶
      font&     = OpenFont&(VARPTR(t&(0)))¶
      IF font&  = 0 THEN BEEP:EXIT SUB¶
      CALL CloseFont(font&)¶
      CALL SetFont(WINDOW(8),font&)¶
END SUB      ¶
¶
SUB NewB(char$,row%,value%) STATIC¶
      n.font&   = PEEKL(WINDOW(8)+52)¶
      n.data&   = PEEKL(n.font&+34)¶
      n.ascii%  = ASC(char$)¶
      n.lo%     = PEEK(n.font&+32)¶
      n.hi%     = PEEK(n.font&+33)¶
      n.modulo% = PEEKW(n.font&+38)¶
      n.offset% = (n.ascii%-n.lo%)+row%*n.modulo%¶
      n.data%   = 0¶
      ¶
      IF n.ascii%<n.lo% OR n.ascii%>n.hi% THEN¶
```

```
        PRINT "Character not in Font!"¶
        ERROR 255¶
      END IF¶
¶
      POKE n.data&+n.offset%,value%¶
END SUB      ¶
      ¶
SUB NewD(char$,row%,bit$) STATIC¶
      n.font&   = PEEKL(WINDOW(8)+52)¶
      n.data&   = PEEKL(n.font&+34)¶
      n.ascii%  = ASC(char$)¶
      n.lo%     = PEEK(n.font&+32)¶
      n.hi%     = PEEK(n.font&+33)¶
      n.modulo% = PEEKW(n.font&+38)¶
      n.offset% = (n.ascii%-n.lo%)+row%*n.modulo%¶
      n.data%   = 0¶
      ¶
      IF n.ascii%<n.lo% OR n.ascii%>n.hi% THEN¶
        PRINT "Character not in Font!"¶
        ERROR 255¶
      END IF¶
      ¶
     '* 8 Bit Alignment¶
      IF LEN(bit$)<>8 THEN¶
        IF LEN(bit$)>8 THEN bit$ = LEFT$(bit$,8)¶
        IF LEN(bit$)<8 THEN bit$ = bit$+soace$(8-
LEN(bit$))¶
      END IF¶
      ¶
     '* Write data in charData¶
      FOR loop1%=7 TO 0 STEP -1¶
        n.check$  = MID$(bit$,8-loop1%,1)¶
        IF n.check$="*" THEN¶
          n.data% = n.data%+2^loop1%¶
        END IF¶
      NEXT loop1%¶
      POKE n.data&+n.offset%,n.data%¶
END SUB              ¶
      ¶
SUB DeleteFont(z.n$) STATIC¶
      z.name$ = UCASE$(z.n$+".font"+CHR$(0))¶
      t&(0)   = SADD(z.name$)¶
      t&(1)   = 8*2^16¶
      font&   = OpenFont&(VARPTR(t&(0)))¶
      IF font&=0 THEN ERROR 255¶
      ¶
      z.size& = PEEKL(font&-4)¶
      IF z.size&<100 OR z.size&>4000 THEN ERROR 255¶
      ¶
     '* Remove from System List¶
      z.1& = PEEKL(font&)¶
      z.2& = PEEKL(font&+4)¶
      POKEL z.1&+4,z.2&¶
      POKEL z.2&,z.1&¶
     ¶
     '* Release RAM¶
```

```
        font& = font&-4¶
        CALL FreeMem(font&,z.size&)¶
        ¶
       '* Load Standard Font¶
        standard$ = "topaz.font"+CHR$(0)¶
        t&(0)     = SADD(standard$)¶
        font&     = OpenFont&(VARPTR(t&(0)))¶
        IF font&  = 0 THEN ERROR 255¶
        CALL SetFont(WINDOW(8),font&) ¶
END SUB¶
        ¶
SUB MakePrgFont(z.n$,ascii.lo%,ascii.hi%) STATIC¶
        z.name$    = UCASE$(z.n$+".font"+CHR$(0))¶
        z.count%   = ascii.hi%-ascii.lo%+2¶
        z.modulo%  = z.count%¶
        z.size&    = z.count%*8+z.count%*4+110¶
        z.offset%  = ascii.lo%-32¶
        z.begin%   = 0¶
        ¶
        mem.opt&   = 2^0+2^16¶
        z.add&     = AllocMem&(z.size&,mem.opt&)¶
        IF z.add&  = 0 THEN ERROR 7¶
        POKEL z.add&,z.size&¶
        z.add&     = z.add&+4¶
        ¶
        z.data&    = z.add&+100¶
        z.loc&     = z.data&+z.count%*8¶
        z.name&    = z.add&+65¶
        ¶
        POKEL z.add&+10,z.name&¶
        POKEW z.add&+18,z.size&-4¶
        POKEW z.add&+20,8¶
        POKE  z.add&+23,64¶
        POKEW z.add&+24,8¶
        POKEW z.add&+26,6¶
        POKE  z.add&+32,ascii.lo%¶
        POKE  z.add&+33,ascii.hi%¶
        POKEL z.add&+34,z.data&¶
        POKEW z.add&+38,z.modulo%¶
        POKEL z.add&+40,z.loc&¶
        ¶
       '* Fill Name Field¶
        FOR loop1%=1 TO LEN(z.name$)¶
          POKE z.name&+loop1%-
1,ASC(MID$(z.name$,loop1%,1))¶
        NEXT loop1%¶
        ¶
       '* charLoc Field¶
        FOR loop1%=0 TO z.count%-1¶
          POKEW z.loc&+(4*loop1%)+0,loop1%*8¶
          POKEW z.loc&+(4*loop1%)+2,8¶
        NEXT loop1%¶
        ¶
       '* charData Field¶
        sample$ = "topaz.font"+CHR$(0)¶
        t&(0)   = SADD(sample$)¶
```

```
        t&(1)    = 8*2^16¶
        sample& = OpenFont&(VARPTR(t&(0)))¶
        IF sample&=0 THEN¶
          PRINT "ROM-Fonts weg???!"¶
          ERROR 255¶
        END IF¶
        s.char&   = PEEKL(sample&+34)¶
        s.modulo% = PEEKW(sample&+38)¶
        CALL CloseFont(sample&)¶
        ¶
        IF z.offset%<0 THEN¶
          z.count%  = z.count%+z.offset%¶
          z.begin%  = ABS(z.offset%)¶
          z.offset% = 0¶
        END IF¶
        ¶
        FOR loop1%=0 TO 7¶
          CALL CopyMem(s.char&+z.offset%+loop1%*s.modulo%,
z.data&+z.begin%+loop1%*z.modulo%,z.count%-1)¶
        NEXT loop1%¶
      ¶
      '* unprintable Character¶
       POKE z.data&+z.modulo%-1+0*z.modulo%,224¶
       POKE z.data&+z.modulo%-1+1*z.modulo%,64¶
       POKE z.data&+z.modulo%-1+2*z.modulo%,64¶
       POKE z.data&+z.modulo%-1+3*z.modulo%,64¶
       POKE z.data&+z.modulo%-1+4*z.modulo%,73¶
       POKE z.data&+z.modulo%-1+5*z.modulo%,73¶
       POKE z.data&+z.modulo%-1+6*z.modulo%,77¶
       POKE z.data&+z.modulo%-1+7*z.modulo%,74¶
       ¶
      '* link¶
       CALL AddFont(z.add&)¶
       t&(0)         = SADD(z.name$)¶
       font.new&     = OpenFont&(VARPTR(t&(0)))¶
       IF font.new& = 0 THEN ERROR 255¶
       ¶
       CALL SetFont(WINDOW(8),font.new&)¶
END SUB¶
```

Altogether, this program provides you all with five SUB programs:

- MakePrgFont
- DeleteFont
- NewD
- NewB
- ActivateFont

MakePrgFont This SUB program allows you to create a completely new font which carries the name assigned by you. Here is the call:

```
MakePrgFont name$,lo%,hi%
```

name$:	Name of the new font
lo%:	ASCII value of first character
hi%:	ASCII value of last character

You determine the number of characters in your font. Every character has an ASCII value that can be determined by using the ASC function. Select the low and high ASCII limits.

```
LINE INPUT "Character: ";z$
PRINT ASC(z$)
```

The codes 0 to 255 are available.

After the font is prepared, it contains no character definitions. Basically, it is "empty"; all the current characters equal nothing. Because of this, we fill the new font with data from the ROM font topaz 8.

After this call, the new font is ready for your commands. All the characters within your ASCII limits will be displayed with "topaz8" characters. Any characters outside these limits will be displayed as an "unprintable character" with the small TW character.

ActivateFont

This SUB program is useless to you if you only want to work with a single font. However, if you want to work with many fonts with different names, you can select any of them with this command.

```
ActivateFont name$
```

name$:	The name of a previously generated font with MakePrgFont

NewD

Our goal was to define our own characters. This is accomplished with NewD. Each character of our font is 8 pixels wide and 8 pixels high. By using NewD, we can define any one of the eight rows of any character.

```
NewD char$,nr%,bit$
```

char$:	The character, that you want to define
nr%:	The row of the character (0-7)
bit$:	The new data row (*=set pixels, "."=unset pixels)

Note:

NewD defines a character from the last generated (or active) font. Obviously, the character must exist in the font in order to be redefined.

NewB

This is a variation of the NewD. When using NewD, the character data for the new character row is displayed as astericks and points in binary form. This binary data must first be converted to decimal. However, if you are familiar with the binary decimal conversion, you can use the decimal values directly. In order to do this, NewB is used.

```
NewB char$,nr%,value%
```

char$,nr%:	Same as with NewD
value%:	Decimal value for row (0-255)

DeleteFont

When you no longer need one of your open fonts, you must close it again. This is accomplished with the command:

```
DeleteFont name$
```

name$:	Name of the font opened using MakePrgFont.

At the end of your program.you must close all the fonts that were opened with MakePrgFont. This returns the assigned memory to the system.

Those of you who want or need more information on making your own fonts will find detailed explanations on the following pages.

MakePrgFont

We fill the TextFont structure (from Section 5.5) with all the required paramenters.Then we initialize the field charLoc with its required parameters. Because the characters of the font have a standard width value of 8 pixels, the offset value is a multiple of 8. The width value is always equal to 8 (see Section 5.5).

The charData field is supposed to be filled with the user defined characters. Since we can assume that not all the characters will be redefined, we fill the font with the ROM font topaz 8. After opening topaz 8, we save the pointer to it in the variable sample&. The modulo is also read. Now we can close topaz again because the charData is in ROM and cannot be lost.

Next, we must initialize two variables z.offset% and z.begin%. Your font won't always have the same contents as the ROM type. The number of the character that the new font will later begin with is contained in the z offset%. The ROM font always starts with ASCII code 32. If the first character in your font has a larger value, for example,"A" (code = 65), then z.offset% contains 65 -32 =33. The opposite is accomplished with z.begin%. If the ASCII code of the first character in your new font is smaller than 32, then z.begin% contains the difference between the two values.

Now we can copy the ROM data to the RAM buffer. For this we use a function of the exec.library:

```
CALL CopyMem(o.data&,z.data&,bytes&)
```

o.data&:	Original data
z.data&:	Target data
bytes&:	Number of bytes to copy

This routine was first availabe with the Kickstart Version 1.2. Users of older versions must either revise these commands into PEEKs and POKEs or completely leave out the loop. Otherwise you have to define all the characters of the new font before working with it.

After all the characters are copied, the shape of the unprintable character is defined as a small "TW". Whenever a user requests a character that doesn't exist in the font, the "TW" character will appear. The data for this unprintable character is stored after the data for all the other characters.

Now the new font is completely functional. To add this font to the system, we use the function AddFont. From this point on, other programs can also access your font (for example, the NOTEPAD). Finally we use OpenFont& to open the new font. The address font.new& must match the address z.add&.

Right now it is necessary to take a closer look at the beginning of the TextFont structure. The message structure looks like this:

Data Structure Message/exec/20 Bytes

Offset	Type	Description
+ 000	Long	Pointer to next font
+ 004	Long	Pointer to previous font
+ 008	Byte	Node type
+ 009	Byte	Priority
+ 010	Long	Pointer to name of font
+ 014	Long	Pointer to Replyport
+ 018	Word	Length of message

Before we call the AddFont routine, we must store the name field and length of the message, which equals the length of the font. After AddFont, the rest of the pointers are initialized, which means that the first two pointers point at different fonts.

This step is very important when you try to delete your font from the system again. This will be explained further in DeleteFont.

ActivateFont Here we search for the font with the requested name. This SUB can only be used to look for fonts that are created with `MakePrgFont`. The reason for this is that the font is only opened for a short time, in order to receive all the necessary pointers, and then closed again. We do not have to keep it open because it has already been opened with `MakePrgFont`.

SetFont This SUB program activates the font.

NewD, NewB We read all the required font data from the RastPort. `NewD` converts the bit definitions into a decimal number, `NewB` works with decimal numbers and therefore, is faster. The values are poked to the locations set by your parameters.

DeleteFont The named font is opened here also. Fonts must always be removed by whatever opened them. ROM fonts are never removed and disk fonts are loaded and handled by AmigaDOS. You are responsible for the cleanup of your own fonts. When reserving the memory, `MakePrgFont` stored the length of the font in the last four bytes before the font. We read out this value. Before we delete the font with `FreeMem`, we must correct the `Systemlist` because `AddFont` integrated our font with the `Systemlist`. The required fields are restored and our font evaporates.

Now we can release the RAM allocated for the font and return this memory to the system. To avoid being left without a font, we activate the ROM font again.

5.5.4 A proportional font

Now we will discuss the complex proportional font. It is almost impossible, in this type of font, to redefine existing characters. Every time we redefine one character, hundreds of bytes would have to be shifted one way or the other to adjust for the new character size. A solution to this problem is to provide a number for the maximum width of any single character. The program then reserves enough memory for a font containing characters of this size. Although this method isn't memory efficient, it is the only practical solution.

Again, our character generator demonstrates its user friendliness. You can easily create characters without being required to use any complex numbers and parameters. We use six SUB programs:

```
MakePrgFont
```

```
DeleteFont
NewB
NewD
ActivateFont
Set
```

You are already familiar with these names. The way these SUB programs work is very similar to those in the fixed width character generator from the previous chapter.

Again, you can create as many fonts as you like. To do this, use the following command:

```
MakePrgFont name$,lo%,hi%,width%,height%,base%
```

`name$:`	Name of font
`lo%:`	Lower ASCII limit (see chapter 5.5.3)
`hi%:`	Upper ASCII limit
`width%:`	Maximum width in pixels
`height%:`	Uniform height in pixels
`base%:`	Baseline (height without underline)

Note: Baseline must be at least one pixel smaller than `height%`, otherwise, while using algorithmic managed fonts (special italic), the system buffer can be overwritten.

After this call, the Amiga generates a font with the above parameters; no other information is required. This font is exactly the opposite from those generated with the fixed width generator because it is empty with no previously defined characters. Now it is your responsibility to define each character in the font.

Before you design a particular character using the familiar SUB programs `NewD` and `NewB`, you must specify an individual size for this character. This is accomplished by using the "`set`" command:

```
Set char$,spacing%,kerning%
```

`char$:`	The character for which this value applies.
`spacing%:`	Width of character (cannot exceed the maximum width of the font generated).
`kerning%:`	Number of pixels to skip before the character you define actually appears.

Additional information can be found in Section 5.5.

All the other SUB programs have similar functions to those in the fixed width generator. However, they are not identical. The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 5.5.4 ¶
'# Program: Proportional Font Generator¶
'# Date:    04/12/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' This program makes it possible to create differently¶
' named proportional fonts and every character¶
' can have its own individual width. Undefined¶
' characters will not have character data, and¶
' will appear only after being successfully¶
' defined.¶
¶
'GRAPHICS-Library¶
DECLARE FUNCTION OpenFont& LIBRARY¶
'CloseFont()¶
'SetFont()¶
'AddFont()¶
¶
'EXEC-Library¶
DECLARE FUNCTION AllocMem& LIBRARY¶
'FreeMem()¶
¶
LIBRARY "graphics.library"¶
LIBRARY "exec.library"¶
¶
init:    '* Generate Fonts¶
          MakePrgFont "tobi",32,65,9,10,7¶
         ¶
         'Set Format:¶
         'Set "Characters",Space,Kern¶
          ¶
          Set  "@",20,3¶
          NewD "@",0,"...***..."¶
          NewD "@",1,"..*...*.."¶
          NewD "@",2,"......*.."¶
          NewD "@",3,".....*..."¶
          NewD "@",4,"....*...."¶
          NewD "@",5,"....*...."¶
          NewD "@",6,"........."¶
          NewD "@",7,"*********"¶
          NewD "@",8,".*******."¶
          NewD "@",9,"..*****.."¶
          ¶
         '* second character using byte method (faster)¶
```

```
          Set  "A",11,1¶
          NewB "A",0,8,126¶
          NewB "A",1,8,129¶
          NewB "A",2,8,157¶
          NewB "A",3,8,161¶
          NewB "A",4,8,161¶
          NewB "A",5,8,157¶
          NewB "A",6,8,129¶
          NewB "A",7,8,126¶
          ¶
         '* Sample Text¶
          PRINT STRING$(40,"A")¶
          PRINT STRING$(40,"@")¶
¶
         '* Delete Font¶
          DeleteFont "tobi"¶
¶
          END¶
¶
SUB ActivateFont(z.n$) STATIC¶
      z.name$  = UCASE$(z.n$+".font"+CHR$(0))¶
      t&(0)    = SADD(z.name$)¶
      t&(1)    = 8*2^16¶
      font&    = OpenFont&(VARPTR(t&(0)))¶
      IF font& = 0 THEN BEEP:EXIT SUB¶
      CALL CloseFont(font&)¶
      CALL SetFont(WINDOW(8),font&)¶
END SUB      ¶
¶
SUB Set(char$,spacing%,kerning%) STATIC¶
      n.font&    = PEEKL(WINDOW(8)+52)¶
      n.space&   = PEEKL(n.font&+44)¶
      n.kern&    = PEEKL(n.font&+48)¶
      n.ascii%   = ASC(char$)¶
      n.lo%      = PEEK(n.font&+32)¶
      n.hi%      = PEEK(n.font&+33)¶
      n.number%  = n.ascii%-n.lo%¶
      IF n.ascii%<n.lo% OR n.ascii%>n.hi% THEN¶
        EXIT SUB¶
      END IF¶
      POKEW n.space&+(2*n.number%),spacing%¶
      POKEW n.kern&+(2*n.number%),kerning%¶
END SUB¶
¶
SUB NewB(char$,row%,bits%,value%) STATIC¶
      n.byte%   = 0¶
      n.bit%    = 0¶
      n.font&   = PEEKL(WINDOW(8)+52)¶
      n.data&   = PEEKL(n.font&+34)¶
      n.loc&    = PEEKL(n.font&+40)¶
      n.ascii%  = ASC(char$)¶
      n.lo%     = PEEK(n.font&+32)¶
      n.hi%     = PEEK(n.font&+33)¶
      n.modulo% = PEEKW(n.font&+38)¶
      n.offset% = row%*n.modulo%¶
      n.height% = PEEKW(n.font&+20)¶
```

```
      n.number% = n.ascii%-n.lo%¶
      n.width%  = PEEKW(PEEKL(n.font&+40)+
(4*n.number%)+2)¶
      n.offset& = PEEKW(PEEKL(n.font&+40)+(4*n.number%))¶
      n.byte%   = INT(n.offset&/8)¶
      n.bit%    = 7-(n.offset&-(n.byte%*8))¶
        ¶
      IF n.ascii%<n.lo% OR n.ascii%>n.hi% THEN¶
        EXIT SUB¶
      END IF¶
¶
      IF bits%>n.width% THEN¶
        bits% = n.width%¶
      END IF¶
      n.data&    = n.data&+n.offset%¶
      FOR loop1% = bits%-1 TO 0 STEP -1¶
        IF (value% AND 2^loop1%)<>0 THEN¶
          POKE n.data&+n.byte%,PEEK(n.data&+n.byte%) OR
(2^n.bit%)¶
        END IF¶
        n.bit% = n.bit%-1¶
        IF n.bit%<0 THEN¶
          n.bit%  = 7¶
          n.byte% = n.byte%+1¶
        END IF¶
      NEXT loop1%¶
¶
      POKEW n.loc&+(4*n.number%)+2,bits%¶
END SUB¶
¶
SUB NewD(char$,row%,bits$) STATIC¶
      bits%     = LEN(bits$)¶
      n.byte%   = 0¶
      n.bit%    = 0¶
      n.font&   = PEEKL(WINDOW(8)+52)¶
      n.data&   = PEEKL(n.font&+34)¶
      n.loc&    = PEEKL(n.font&+40)¶
      n.ascii%  = ASC(char$)¶
      n.lo%     = PEEK(n.font&+32)¶
      n.hi%     = PEEK(n.font&+33)¶
      n.modulo% = PEEKW(n.font&+38)¶
      n.offset% = row%*n.modulo%¶
      n.height% = PEEKW(n.font&+20)¶
      n.number% = n.ascii%-n.lo%¶
      n.width%  = PEEKW(PEEKL(n.font&+40)+
(4*n.number%)+2)¶
      n.offset& = PEEKW(PEEKL(n.font&+40)+(4*n.number%))¶
      n.byte%   = INT(n.offset&/8)¶
      n.bit%    = 7-(n.offset&-(n.byte%*8))¶
      ¶
      IF n.ascii%<n.lo% OR n.ascii%>n.hi% THEN¶
        EXIT SUB¶
      END IF¶
¶
      IF bits%>n.width% THEN¶
        bits% = n.width%¶
```

```
      END IF¶
      n.data& = n.data&+n.offset%¶
      FOR loop1%=bits%-1 TO 0 STEP -1¶
        c$ = MID$(bits$,bits%-loop1%,1)        ¶
        IF c$="*" THEN¶
          POKE n.data&+n.byte%,PEEK(n.data&+n.byte%) OR
(2^n.bit%)¶
        END IF¶
        n.bit% = n.bit%-1¶
        IF n.bit%<0 THEN¶
          n.bit%  = 7¶
          n.byte% = n.byte%+1¶
        END IF¶
      NEXT loop1%¶
¶
      POKEW n.loc&+(4*n.number%)+2,bits%¶
END SUB      ¶
      ¶
SUB DeleteFont(z.n$) STATIC¶
      z.name$  = UCASE$(z.n$+".font"+CHR$(0))¶
      t&(0)    = SADD(z.name$)¶
      t&(1)    = 8*2^16¶
      font&    = OpenFont&(VARPTR(t&(0)))¶
      IF font& = 0 THEN ERROR 255¶
      ¶
      z.size&  = PEEKL(font&-4)¶
      IF z.size&<100 OR z.size&>40000& THEN ERROR 255¶
      ¶
     '* Delete from System List¶
      z.1& = PEEKL(font&)¶
      z.2& = PEEKL(font&+4)¶
      POKEL z.1&+4,z.2&¶
      POKEL z.2&,z.1&¶
     ¶
     '* Release RAM¶
      font& = font&-4¶
      CALL FreeMem(font&,z.size&)¶
      ¶
     '* Load Standard Font¶
      standard$ = "topaz.font"+CHR$(0)¶
      t&(0)     = SADD(standard$)¶
      font&     = OpenFont&(VARPTR(t&(0)))¶
      IF font&  = 0 THEN ERROR 255¶
      CALL SetFont(WINDOW(8),font&) ¶
END SUB¶
       ¶
SUB MakePrgFont(z.n$,ascii.lo%,ascii.hi%,
z.maxX%,z.height%,z.baseline%) STATIC¶
     ¶
      z.name$    = UCASE$(z.n$+".font"+CHR$(0))¶
      z.number%  = ascii.hi%-ascii.lo%+2¶
      z.modulo%  = (z.number%*z.maxX%+4)/8¶
      IF (z.modulo% MOD 2)<>0 THEN ¶
        z.modulo%=z.modulo%+1¶
      END IF¶
        ¶
```

```
        z.size& = z.modulo%*z.height%+z.number%*8+110¶
        ¶
        IF z.baseline%>=z.height% THEN¶
          z.baseline% = z.height%-1¶
        END IF  ¶
        ¶
        mem.opt&  = 2^0+2^16¶
        z.add&    = AllocMem&(z.size&,mem.opt&)¶
        IF z.add& = 0 THEN ERROR 7¶
        POKEL z.add&,z.size&¶
        ¶
        z.add&    = z.add&+4¶
        z.data&   = z.add&+100¶
        z.loc&    = z.data&+z.modulo%*z.height%¶
        IF z.loc&/2<>INT(z.loc&/2) THEN¶
          z.loc&  = z.loc&+1¶
        END IF  ¶
        ¶
        z.kern&   = z.loc&+4*z.number%¶
        z.space&  = z.kern&+2*z.number%¶
        z.name&   = z.add&+65¶
      ¶
        POKEL z.add&+10,z.name&¶
        POKEW z.add&+18,z.size&-4¶
        POKEW z.add&+20,z.height%¶
        POKE  z.add&+23,64+32¶
        POKEW z.add&+24,z.maxX%¶
        POKEW z.add&+26,z.baseline%¶
        POKE  z.add&+32,ascii.lo%¶
        POKE  z.add&+33,ascii.hi%¶
        POKEL z.add&+34,z.data&¶
        POKEW z.add&+38,z.modulo%¶
        POKEL z.add&+40,z.loc&¶
        POKEL z.add&+44,z.space&¶
        POKEL z.add&+48,z.kern&¶
        ¶
      '* Fill Name Field¶
        FOR loop1%=1 TO LEN(z.name$)¶
          POKE z.name&+loop1%-
1,ASC(MID$(z.name$,loop1%,1))¶
        NEXT loop1%¶
      ¶
      '* charLoc Field ¶
        FOR loop1%=0 TO z.number%-1¶
          POKEW z.loc&+(4*loop1%)+0,loop1%*z.maxX%¶
          POKEW z.loc&+(4*loop1%)+2,z.maxX%¶
        NEXT loop1%¶
      ¶
      '* link¶
        CALL AddFont(z.add&)¶
        t&(0)       = SADD(z.name$)¶
        font.new&   = OpenFont&(VARPTR(t&(0)))¶
        IF font.new&=0 THEN ERROR 255¶
        ¶
        CALL SetFont(WINDOW(8),font.new&)¶
END SUB¶
```

6. Graphic hardcopy

Obviously, the graphics you have created aren't useful if they disappear the moment you turn off the power to your computer. You need a routine that enables you to print out your graphics to a printer.

Most computers require a boring and difficult machine language routine in order to do this. The Amiga, however, provides us with system software to produce "hardcopies". After you have selected your printer in Preferences, you don't have to do much more (if your printer is capable of printing graphics). The printer.device takes over all the work. It reads the graphic image, controls the printer and the patterns used for colors. Don't confuse the printer.device with the printer. The printer.device is a component of the Amiga operating system that controls your printer.

Before we can print graphics, we must find a way to make contact with the printer.device. We do this by using the Amiga's standard I/O (Input/Output), which is managed by the exec.library.

The data structure `IODRPReq` (I/O Dump Rastport Request), which means input/output request to print a RastPort, is specifically designed for this purpose. First you must fill it with the data that needs to be sent to the printer.device. The structure appears as follows:

Data structure `IODRPReq`/printer/62 Bytes

Offset	Type	Description
+ 000	Long	Next data structure
+ 004	Long	Previous data structure
+ 008	Byte	Node-type (5=MESSAGE)
+ 009	Byte	Priority (0=normal)
+ 010	Long	Pointer to name
+ 014	Long	Pointer to message port (reply port)
+ 018	Word	Length of message
+ 020	Long	ioDevice
+ 024	Long	ioUnit
+ 028	Word	Command (11=DumpRastPort())
+ 030	Byte	Flag (1=Quick I/O)
+ 031	Byte	Error-Nr. 1 = User canceled print event 2 = No graphic printer connected 3 = HAM cannot be inverted

Offset	Type	Description
		4 = Print coordinates are invalid
		5 = Print dimensions are invalid
		6 = No memory free for internal calculations
		7 = No memory free for print buffer
+ 032	Long	Address of RastPort
+ 036	Long	Address of colormap
+ 040	Long	Mode (of ViewPort)
+ 044	Word	RastPort: X begin (0=normal)
+ 046	Word	RastPort: Y begin (0=normal)
+ 048	Word	RastPort: width
+ 050	Word	RastPort: height
+ 052	Long	Printer: width
+ 056	Long	Printer: height
+ 060	Word	Special modes
		1 = Bit 0 = 1: print width in 1/1000 inch
		2 = Bit 1 = 1: print height in 1/1000 inch
		4 = Bit 2 = 1: print width large as possible
		8 = Bit 3 = 1: print height large as possible
		16 = Bit 4 = 1: print width part of FULL-X
		32 = Bit 5 = 1: print height part of FULL-Y
		128= Bit 7 = 1: X-Y ratio smoothing
		256= Bit 8 = 1: low resolution
		512= Bit 9 = 1: medium resolution
		768= Bits 8+9 = 1: high resolution
		1024= Bit 10 = 1: highest resolution

Before you can use this data structure, you must fill most of the data fields with the correct values. This includes the pointer to the ReplyPort which is an exec message port. The ReplyPort is a sender/receiver between related tasks and, as usual, is also a data structure.

Data structure Message Port/exec/34 Bytes

Offset	Type	Description
+ 000	Long	next data structure
+ 004	Long	previous data structure
+ 008	Byte	Type (4=MESSAGE PORT)
+ 009	Byte	Priority (0=normal)
+ 010	Long	Pointer to name
+ 014	Byte	Flags
		PA SIGNAL = 1
		PA SOFTINT = 2
		PA IGNORE = 4
+ 015	Byte	Signal bit

Offset	Type	Description
+ 016	Long	Pointer to task for signal
+ 020	Long	first data structure
+ 024	Long	last data structure
+ 028	Long	second to last data structure
+ 032	Byte	Type
+ 033	Byte	unused

Once you have correctly initialized both structures, you are ready to access the printer. The `OpenDevice` function is responsible for this:

```
status%=OpenDevice%(name$,unit%,io&,flags%)
```

`name$:` Pointer to zero terminated name string, here: `"printer.device"+CHR$(0)`.
`unit%:` Device number; not required now, so = 0.
`io&:` Address of the correctly initialized I/O data blocks, here: `IODRPReq`.
`flag%:` not required now, so = 0.

This function then returns a status report. If everything worked correctly, the returned value is equal to zero. If a value other than zero is returned, then we were unable to open a printer. The printer was either not connected properly, turned off or was being used by another task. Note: When you use `LPRINT` in your program, the hardcopy routine cannot access the printer.

When the printer is properly opened, the fields `ioDevice` and `ioUnit` in the `IODRPReq` structure are filled. The exec command `DoIO` starts the printing function:

```
error%=DoIO%(io&)
```

`io&` Address of `IODRPReq` block
`error%` all ok = 0
for error decoding see `IODRPReq` structure (above), Error field

Now that we have discussed the theory, we will present some examples.

6.1 A simple hardcopy routine

The following program is the basic foundation for a hardcopy routine. The SUB program `Hardcopy` prints the contents of the current output window (use `WINDOW OUTPUT` to set).

```
'########################################
'#
'# Section: 6.1
'# Program: Hardcopy I
'# Date:    04/13/87
'# Author:  tob
'# Version: 1.0
'#
'########################################
' Prints the contents of the current Output Window
' as graphic hardcopy on a graphic capable printer.
PRINT "Searching for .bmap file..."

'EXEC-Library
DECLARE FUNCTION AllocMem& LIBRARY
DECLARE FUNCTION DoIO% LIBRARY
DECLARE FUNCTION OpenDevice% LIBRARY
DECLARE FUNCTION AllocSignal% LIBRARY
DECLARE FUNCTION FindTask& LIBRARY
'FreeMem()
'CloseDevice()
'FreeSignal()
'AddPort()
'RemPort()

LIBRARY "exec.library"

init:     '* draw something
          CLS
          CIRCLE (300,100),100
          LINE   (10,10)-(200,100),2,bf

          Hardcopy

          END

SUB Hardcopy STATIC
      mem.opt& = 2^0+2^16
      p.io&    = AllocMem&(100,mem.opt&)
      p.port&  = p.io&+62
      IF p.io& = 0 THEN ERROR 7

      f.windo&     = WINDOW(7)
      f.rastport&  = PEEKL(f.windo&+50)
```

```
        f.height%    = PEEKW(f.windo&+114)
        f.screen&    = PEEKL(f.windo&+46)
        f.viewport&  = f.screen&+44
        f.colormap&  = PEEKL(f.viewport&+4)
        f.vp.mode%   = PEEKW(f.viewport&+32)

        p.sigBit% = AllocSignal%(-1)
        IF p.sigBit% = -1 THEN
          PRINT "No Signalbit free!"
          CALL FreeMem(p.io&,100)
          EXIT SUB
        END IF
        p.sigTask& = FindTask&(0)

        POKE  p.port&+8,4
        POKEL p.port&+10,p.port&+34
        POKE  p.port&+15,p.sigBit%
        POKEL p.port&+16,p.sigTask&
        POKEL p.port&+20,p.port&+24
        POKEL p.port&+28,p.port&+20
        POKE  p.port&+34,ASC("P")
        POKE  p.port&+35,ASC("R")
        POKE  p.port&+36,ASC("T")

        CALL AddPort(p.port&)

        POKE  p.io&+8,5
        POKEL p.io&+14,p.port&
        POKEW p.io&+28,11
        POKEL p.io&+32,f.rastport&
        POKEL p.io&+36,f.colormap&
        POKEL p.io&+40,f.vp.mode%
        POKEW p.io&+48,f.width%
        POKEW p.io&+50,f.height%
        POKEL p.io&+52,f.width%
        POKEL p.io&+56,f.height%
        POKEW p.io&+60,4

        d.name$ = "printer.device"+CHR$(0)
        status% = OpenDevice%(SADD(d.name$),0,p.io&,0)
        IF status%<>0 THEN
          PRINT "Printer is not free."
          CALL  FreeMem(p.io&,100)
          CALL  FreeSignal(p.sigBit%)
          EXIT SUB
        END IF

        ercond% = DoIO%(p.io&)

        CALL CloseDevice(p.io&)
        CALL RemPort(p.port&)
        CALL FreeMem(p.io&,100)
        CALL FreeSignal(p.sigBit%)
        PRINT "Error Code: ";ercond%
END SUB
```

About the program:

When you print using a black-and-white printer, the printer.device converts all the colors on the screen into patterns on the printout. Different patterns represent different colors. To achieve a white background you have to set the colors accordingly:

```
PALETTE 0,1,1,1
COLOR 1,0
```

Possible Errors:

No graphic print out. Here is a checklist:

Was an error code returned? Note: This can take up to 30 seconds.

Error codes

The error code provides information about the type of error. The following codes are possible:

Code 1: You Interrupted the Print Event
You have intentionally stopped the print. For example, you selected `Cancel` in the `PRINTER TROUBLE` requester instead of correcting the problem.

Code 2: Not a Graphic Printer
The printer you selected in preferences cannot print graphics (such as a daisy wheel printer, etc.).

Code 3: Hold and Modify (HAM)
You cannot invert HAM graphics because of the method used to create their colors. See Section 4.2.2.

Code 4: Invalid Print Coordinates
This error should not occur when your program entries are correct. When it does occur, you have probably made a typing error and the X and Y coordinates are outside the RastPort.

Code 5: Invalid Dimensions
See Code 4. The RastPort width and/or height is larger than the existing RastPort.

Code 6 and 7: Out of Memory
There is not enough available memory to handle the task.

No error code

If you received the `No Signalbit Free` error message, you have overloaded the multitasking system. You will have to wait until another program releases a signalbit.

If you received the `Printer not free` error message, the printer is currently being used by another task (`LPRINT` in your own program, for example). Another possibility is that a task opened the printer and did not close it again. The only solution to this is to reboot the system (reset).

6.2 Hardcopies: enlarging and shrinking

The previous program creates hardcopies that are suitable for most users' needs. However, we have barely utilized the many capabilities of the printer.device. The following hardcopy routine enables you to print a section of a window. When this section is sent to the printer, it can either be as large as the entire window or can be reduced or enlarged.

The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 6.2¶
'# Program: Hardcopy II¶
'# Date:    04/13/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' Allows you to print any portion of a window¶
' and enlarge or shrink the printed output.¶
¶
PRINT "Searching for .bmap file..."¶
¶
'EXEC-Library¶
DECLARE FUNCTION AllocMem& LIBRARY¶
DECLARE FUNCTION DoIO% LIBRARY¶
DECLARE FUNCTION OpenDevice% LIBRARY¶
DECLARE FUNCTION AllocSignal% LIBRARY¶
DECLARE FUNCTION FindTask& LIBRARY¶
'FreeMem()¶
'CloseDevice()¶
'FreeSignal()¶
'AddPort()¶
'RemPort()¶
¶
LIBRARY "exec.library"¶
¶
init:      '* draw somethin¶
           CLS¶
           CIRCLE (300,100),100¶
           LINE   (10,10)-(200,100),2,bf¶
           ¶
          '* Colors: black & white contrast¶
           PALETTE 0,1,1,1¶
           PALETTE 1,0,0,0¶
¶
```

```
                ParameterHardcopy 200,10,200,100,1.2,.5¶
                ¶
                END¶
                ¶
SUB ParameterHardcopy(x%,y%,p.width%,p.height%,f1,f2)
STATIC¶
        mem.opt& = 2^0+2^16¶
        p.io&    = AllocMem&(100,mem.opt&)¶
        p.port&  = p.io&+62¶
        IF p.io& = 0 THEN ERROR 7¶
¶
        f.windo&     = WINDOW(7)¶
        f.rastport&  = PEEKL(f.windo&+50)¶
        f.width%     = PEEKW(f.windo&+112)¶
        f.height%    = PEEKW(f.windo&+114)¶
        f.screen&    = PEEKL(f.windo&+46)¶
        f.viewport&  = f.screen&+44¶
        f.colormap&  = PEEKL(f.viewport&+4)¶
        f.vp.mode%   = PEEKW(f.viewport&+32)¶
        ¶
        ¶
        p.sigBit%    = AllocSignal%(-1)¶
        IF p.sigBit% = -1 THEN¶
          PRINT "No Signalbit free!"¶
          CALL FreeMem(p.io&,100)¶
          EXIT SUB¶
        END IF¶
        p.sigTask&   = FindTask&(0)¶
        ¶
        POKE  p.port&+8,4¶
        POKEL p.port&+10,p.port&+34¶
        POKE  p.port&+15,p.sigBit%¶
        POKEL p.port&+16,p.sigTask&¶
        POKEL p.port&+20,p.port&+24¶
        POKEL p.port&+28,p.port&+20¶
        POKE  p.port&+34,ASC("P")¶
        POKE  p.port&+35,ASC("R")¶
        POKE  p.port&+36,ASC("T")¶
        ¶
        CALL AddPort(p.port&)¶
        ¶
        POKE  p.io&+8,5¶
        POKEL p.io&+14,p.port&¶
        POKEW p.io&+28,11¶
        POKEL p.io&+32,f.rastport&¶
        POKEL p.io&+36,f.colormap&¶
        POKEL p.io&+40,f.vp.mode%¶
        POKEW p.io&+44,x%¶
        POKEW p.io&+46,y%¶
        POKEW p.io&+48,p.width%¶
        POKEW p.io&+50,p.height%¶
        POKEL p.io&+52,f.width%*f1¶
        POKEL p.io&+56,f.height%*f2¶
¶
        d.name$ = "printer.device"+CHR$(0)¶
        status% = OpenDevice%(SADD(d.name$),0,p.io&,0)¶
```

```
        IF status%<>0 THEN¶
          PRINT "Printer is not free."¶
          CALL FreeMem(p.io&,100)¶
          CALL FreeSignal(p.sigBit%)¶
          EXIT SUB¶
        END IF¶
        ¶
        ercond% = DoIO%(p.io&)¶
        ¶
        CALL CloseDevice(p.io&)¶
        CALL RemPort(p.port&)¶
        CALL FreeMem(p.io&,100)¶
        CALL FreeSignal(p.sigBit%)¶
        PRINT "Error Code: ";ercond%¶
END SUB¶
```

Call:

```
ParameterHardcopy x%,y%,width%,height%,f1,f2
```

`x%,y%:` Coordinates of the upper left hand corner of the window piece that you want to print.

`width%:` Width of the section in pixels.

`height%:` Height of the section in pixels.

`f1:` Enlargement factor horizontal.

`f2:` Enlargement factor vertical.

6.3 Printing selected windows

So far, we have only printed output from the AmigaBASIC windows. However, the printer.device can also print the contents of any window (preferences for example). In Section 3.1, we showed you how to access all of the windows in the system. We will use this knowledge in a slightly different way in our next program, which will print any selected window. To do this, only the name of the window is required. (Note: The Preference window is named "Preferences")

The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 6.3 ¶
'# Program: Hardcopy III¶
'# Date:    04/13/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' Prints the contents of any selected system window for¶
' which you know the name, including selected pieces,¶
' enlarging, and shrinking.¶
¶
PRINT "Searching for .bmap file..."¶
¶
'EXEC-Library¶
DECLARE FUNCTION AllocMem& LIBRARY¶
DECLARE FUNCTION DoIO% LIBRARY¶
DECLARE FUNCTION OpenDevice% LIBRARY¶
DECLARE FUNCTION AllocSignal% LIBRARY¶
DECLARE FUNCTION FindTask& LIBRARY¶
'FreeMem()¶
'CloseDevice()¶
'FreeSignal()¶
'AddPort()¶
'RemPort()¶
¶
LIBRARY "exec.library"¶
¶
init:     '* here we go¶
          CLS¶
          PALETTE 0,1,1,1¶
          PALETTE 1,0,0,0¶
¶
```

```
            LIST¶
            UniversalHardcopy "LIST",0,0,200,100,.8,.5¶
            ¶
            END¶
            ¶
SUB UniversalHardcopy(na$,x%,y%,p.width%,p.height%,f1,f2)
STATIC¶
      f.windo&    = WINDOW(7)¶
      f.reg&      = PEEKL(f.windo&+66)¶
      WHILE f.reg&>0¶
        f.windo&    = f.reg&¶
        f.reg&      = PEEKL(f.windo&+66)¶
      WEND¶
      ¶
      finder:¶
      f.title&      = PEEKL(f.windo&+32)¶
      check%        = PEEK(f.title&+count%)¶
      WHILE check%>0¶
        check$      = check$+CHR$(check%)¶
        count%      = count%+1¶
        check%      = PEEK(f.title&+count%)¶
      WEND¶
      found$        = check$:check$="":count%=0¶
      IF UCASE$(found$)<>UCASE$(na$) THEN¶
        f.windo&    = PEEKL(f.windo&+70)¶
        IF f.windo&>0 THEN¶
          GOTO finder¶
        ELSE¶
          PRINT "Window does not exist!"¶
          EXIT SUB¶
        END IF¶
      END IF¶
            ¶
      mem.opt&   = 2^0+2^16¶
      p.io&      = AllocMem&(100,mem.opt&)¶
      p.port&    = p.io&+62¶
      IF p.io&   = 0 THEN ERROR 7¶
¶
      f.rastport& = PEEKL(f.windo&+50)¶
      f.width%    = PEEKW(f.windo&+112)¶
      f.height%   = PEEKW(f.windo&+114)¶
      f.screen&   = PEEKL(f.windo&+46)¶
      f.viewport& = f.screen&+44¶
      f.colormap& = PEEKL(f.viewport&+4)¶
      f.vp.mode%  = PEEKW(f.viewport&+32)¶
      ¶
      ¶
      p.sigBit% = AllocSignal%(-1)¶
      IF p.sigBit% = -1 THEN¶
        PRINT "No Signalbit free!"¶
        CALL FreeMem(p.io&,100)¶
        EXIT SUB¶
      END IF¶
      p.sigTask&=FindTask&(0)¶
      ¶
      POKE  p.port&+8,4¶
```

```
        POKEL p.port&+10,p.port&+34¶
        POKE  p.port&+15,p.sigBit%¶
        POKEL p.port&+16,p.sigTask&¶
        POKEL p.port&+20,p.port&+24¶
        POKEL p.port&+28,p.port&+20¶
        POKE  p.port&+34,ASC("P")¶
        POKE  p.port&+35,ASC("R")¶
        POKE  p.port&+36,ASC("T")¶
        ¶
        CALL AddPort(p.port&)¶
        ¶
        POKE  p.io&+8,5¶
        POKEL p.io&+14,p.port&          ¶
        POKEW p.io&+28,11¶
        POKEL p.io&+32,f.rastport&¶
        POKEL p.io&+36,f.colormap&¶
        POKEL p.io&+40,f.vp.mode%¶
        POKEW p.io&+44,x%¶
        POKEW p.io&+46,y%¶
        POKEW p.io&+48,p.width%¶
        POKEW p.io&+50,p.height%¶
        POKEL p.io&+52,f.width%*f1¶
        POKEL p.io&+56,f.height%*f2¶
¶
        d.name$ = "printer.device"+CHR$(0)¶
        status% = OpenDevice%(SADD(d.name$),0,p.io&,0)¶
        IF status%<>0 THEN¶
          PRINT "Printer is not free."¶
          CALL FreeMem(p.io&,100)¶
          CALL FreeSignal(p.sigBit%)¶
          EXIT SUB¶
        END IF¶
        ¶
        ercond% = DoIO%(p.io&)¶
        ¶
        CALL CloseDevice(p.io&)¶
        CALL RemPort(p.port&)¶
        CALL FreeMem(p.io&,100)¶
        CALL FreeSignal(p.sigBit%)¶
        PRINT "Error Code: ";ercond%¶
END SUB¶
```

6.4 ScreenDump - a complete screen

It is also possible to print a screen since it has its own RastPort. The following program is very similar to the previous one. Instead of requiring the window name, the screen number is required (0=Workbench Screen). This program only functions when the output window is in the Workbench Screen.

The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 6.3 ¶
'# Program: Hardcopy III¶
'# Date:    04/13/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' Prints the contents of any selected system window for¶
' which you know the name, including selected pieces,¶
' enlarging, and shrinking.¶
¶
PRINT "Searching for .bmap file..."¶
¶
'EXEC-Library¶
DECLARE FUNCTION AllocMem& LIBRARY¶
DECLARE FUNCTION DoIO% LIBRARY¶
DECLARE FUNCTION OpenDevice% LIBRARY¶
DECLARE FUNCTION AllocSignal% LIBRARY¶
DECLARE FUNCTION FindTask& LIBRARY¶
'FreeMem()¶
'CloseDevice()¶
'FreeSignal()¶
'AddPort()¶
'RemPort()¶
¶
LIBRARY "exec.library"¶
¶
init:      '* here we go¶
           CLS¶
           PALETTE 0,1,1,1¶
           PALETTE 1,0,0,0¶
¶
           LIST¶
           UniversalHardcopy "LIST",0,0,200,100,.8,.5¶
```

```
          ¶
          END¶
          ¶
SUB UniversalHardcopy(na$,x%,y%,p.width%,p.height%,f1,f2)
STATIC¶
      f.windo&    = WINDOW(7)¶
      f.reg&      = PEEKL(f.windo&+66)¶
      WHILE f.reg&>0¶
        f.windo&    = f.reg&¶
        f.reg&      = PEEKL(f.windo&+66)¶
      WEND¶
      ¶
      finder:¶
      f.title&      = PEEKL(f.windo&+32)¶
      check%        = PEEK(f.title&+count%)¶
      WHILE check%>0¶
        check$      = check$+CHR$(check%)¶
        count%      = count%+1¶
        check%      = PEEK(f.title&+count%)¶
      WEND¶
      found$        = check$:check$="":count%=0¶
      IF UCASE$(found$)<>UCASE$(na$) THEN¶
        f.windo&    = PEEKL(f.windo&+70)¶
        IF f.windo&>0 THEN¶
          GOTO finder¶
        ELSE¶
          PRINT "Window does not exist!"¶
          EXIT SUB¶
        END IF¶
      END IF¶
           ¶
      mem.opt&   = 2^0+2^16¶
      p.io&      = AllocMem&(100,mem.opt&)¶
      p.port&    = p.io&+62¶
      IF p.io&   = 0 THEN ERROR 7¶
¶
      f.rastport& = PEEKL(f.windo&+50)¶
      f.width%    = PEEKW(f.windo&+112)¶
      f.height%   = PEEKW(f.windo&+114)¶
      f.screen&   = PEEKL(f.windo&+46)¶
      f.viewport& = f.screen&+44¶
      f.colormap& = PEEKL(f.viewport&+4)¶
      f.vp.mode%  = PEEKW(f.viewport&+32)¶
      ¶
      ¶
      p.sigBit% = AllocSignal%(-1)¶
      IF p.sigBit% = -1 THEN¶
        PRINT "No Signalbit free!"¶
        CALL FreeMem(p.io&,100)¶
        EXIT SUB¶
      END IF¶
      p.sigTask&=FindTask&(0)¶
      ¶
      POKE  p.port&+8,4¶
      POKEL p.port&+10,p.port&+34¶
      POKE  p.port&+15,p.sigBit%¶
```

```
        POKEL p.port&+16,p.sigTask&¶
        POKEL p.port&+20,p.port&+24¶
        POKEL p.port&+28,p.port&+20¶
        POKE  p.port&+34,ASC("P")¶
        POKE  p.port&+35,ASC("R")¶
        POKE  p.port&+36,ASC("T")¶
        ¶
        CALL AddPort(p.port&)¶
        ¶
        POKE  p.io&+8,5¶
        POKEL p.io&+14,p.port&          ¶
        POKEW p.io&+28,11¶
        POKEL p.io&+32,f.rastport&¶
        POKEL p.io&+36,f.colormap&¶
        POKEL p.io&+40,f.vp.mode%¶
        POKEW p.io&+44,x%¶
        POKEW p.io&+46,y%¶
        POKEW p.io&+48,p.width%¶
        POKEW p.io&+50,p.height%¶
        POKEL p.io&+52,f.width%*f1¶
        POKEL p.io&+56,f.height%*f2¶
¶
        d.name$ = "printer.device"+CHR$(0)¶
        status% = OpenDevice%(SADD(d.name$),0,p.io&,0)¶
        IF status%<>0 THEN¶
          PRINT "Printer is not free."¶
          CALL FreeMem(p.io&,100)¶
          CALL FreeSignal(p.sigBit%)¶
          EXIT SUB¶
        END IF¶
        ¶
        ercond% = DoIO%(p.io&)¶
        ¶
        CALL CloseDevice(p.io&)¶
        CALL RemPort(p.port&)¶
        CALL FreeMem(p.io&,100)¶
        CALL FreeSignal(p.sigBit%)¶
        PRINT "Error Code: ";ercond%¶
END SUB¶
```

6.5 Multitasking hardcopy

The Amiga can use two different types of I/O: synchronous and asynchronous. Until now, we have only worked with synchronous I/O, which means our program has to wait for the hardcopy to finish. This has both advantages and disadvantages. An advantage is that the program cannot change the drawing while it is being printed, which would cause problems. However, waiting for the printing to finish wastes a lot of time.

Asynchronous I/O can solve this problem by sending the data contained in the printer.device to the printer and immediately returning control to the BASIC interpreter. This means that the program does not have to wait for the printer to finish. Although this method is very useful, it requires extremely complex programming.

To demonstrate this programming method, we have rewritten the program from Section 6.1 for asynchronous I/O. Just as before, the `hardcopy` subprogram sends the contents of the output window to your printer. Then something remarkable happens. Your BASIC program no longer waits for the printer but, instead, immediately continues with the other tasks.

A call to the `Delete` subprogram must be at the end of your program. This routine releases the running I/O request, returns memory to the system and then closes the printer.

The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 6.5¶
'# Program: Hardcopy V (Multitasking)¶
'# Date:    04/13/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
' This is a Multi-Tasking Hardcopy Routine, which¶
' demonstrates all the abilities of the Amiga:¶
' The program does not have to wait on the printing and¶
' can continue working immediately after calling the¶
' print. The printing continues as a separate task.¶
' WARNING: The entire area being printed with these¶
```

```
' routines is stored temporarily. This require a lot¶
' of memory.¶
¶
PRINT "Searching for .bmap files..."¶
¶
'GRAPHICS-Library¶
DECLARE FUNCTION BltBitMap% LIBRARY¶
DECLARE FUNCTION AllocRaster& LIBRARY¶
'FreeRaster()¶
¶
'EXEC-Library¶
DECLARE FUNCTION AllocMem& LIBRARY¶
DECLARE FUNCTION OpenDevice% LIBRARY¶
DECLARE FUNCTION AllocSignal% LIBRARY¶
DECLARE FUNCTION FindTask& LIBRARY¶
DECLARE FUNCTION WaitIO% LIBRARY¶
DECLARE FUNCTION CheckIO% LIBRARY¶
'FreeMem()¶
'CloseDevice()¶
'FreeSignal()¶
'AddPort()¶
'RemPort()¶
'SendIO()¶
¶
LIBRARY "exec.library"¶
LIBRARY "graphics.library"¶
¶
init:      '* draw something¶
           CLS¶
           CIRCLE (300,100),100¶
           LINE   (10,10)-(200,100),2,bf¶
           ¶
           Hardcopy¶
           ¶
demo:      '* You could replace this with your program¶
           WHILE INKEY$=""¶
             PRINT "Instead of this wait loop you could"¶
             PRINT "have a graphic program that does"¶
             PRINT "not have to wait for printing"¶
             PRINT"to be completed!"¶
             PRINT¶
             PRINT "Press any key to abort."¶
           WEND  ¶
           ¶
          '* asynchronous I/O provisions¶
           iodelete¶
           END¶
           ¶
SUB Hardcopy STATIC¶
      SHARED p.io&,p.sigBit%,f.rastport&¶
      SHARED p.port&¶
      mem.opt& = 2^0+2^16¶
      p.io&    = AllocMem&(240,mem.opt&)¶
      p.port&  = p.io&+62¶
      p.rast&  = p.io&+100¶
      p.bmap&  = p.io&+200¶
```

```
        IF p.io&=0 THEN ERROR 7¶
¶
        f.windo&     = WINDOW(7)¶
        f.rastport&  = PEEKL(f.windo&+50)¶
        f.bitmap&    = PEEKL(f.rastport&+4)¶
        f.width%     = PEEKW(f.windo&+112)¶
        f.height%    = PEEKW(f.windo&+114)¶
        f.screen&    = PEEKL(f.windo&+46)¶
        f.viewport&  = f.screen&+44¶
        f.colormap&  = PEEKL(f.viewport&+4)¶
        f.vp.mode%   = PEEKW(f.viewport&+32)¶
        f.x%         = PEEKW(f.bitmap&)*8¶
        f.y%         = PEEKW(f.bitmap&+2)¶
        f.depth%    = PEEK(f.bitmap&+5)¶
¶
        CALL CopyMem(f.rastport&,p.rast&,100)¶
        CALL CopyMem(f.bitmap&,p.bmap&,40)¶
        ¶
        FOR loop1%=0 TO f.depth%-1¶
          p.plane&(loop1%) = AllocRaster&(f.x%,f.y%)¶
          IF p.plane&(loop1%)=0 THEN¶
            FOR loop2%=0 TO loop1%-1¶
              CALL FreeRaster(p.plane&(loop2%),f.x%,f.y%)¶
            NEXT loop2%¶
            CALL FreeMem(p.io&,240)¶
            PRINT "Out Of Memory!"¶
            EXIT SUB¶
          END IF    ¶
          POKEL p.bmap&+8+loop1%*4,p.plane&(loop1%)¶
        NEXT loop1%¶
        tempA$ = SPACE$(f.x%/8)¶
        pc%    = BltBitMap%(f.bitmap&,0,0,p.bmap&,0,0,f.x%,
f.y%,200,255,SADD(tempA$))¶
        IF pc%<>f.depth% THEN ERROR 255¶
        ¶
        POKEL p.rast&+4,p.bmap&¶
        f.rastport& = p.rast&¶
        ¶
        p.sigBit% = AllocSignal%(-1)¶
        IF p.sigBit%=-1 THEN¶
          PRINT "No Signalbit free!"¶
          CALL FreeMem(p.io&,240)¶
          EXIT SUB¶
        END IF¶
        p.sigTask& = FindTask&(0)¶
        ¶
        POKE  p.port&+8,4¶
        POKEL p.port&+10,p.port&+34¶
        POKE  p.port&+15,p.sigBit%¶
        POKEL p.port&+16,p.sigTask&¶
        POKEL p.port&+20,p.port&+24¶
        POKEL p.port&+28,p.port&+20¶
        POKE  p.port&+34,ASC("P")¶
        POKE  p.port&+35,ASC("R")¶
        POKE  p.port&+36,ASC("T")¶
        ¶
```

```
        CALL AddPort(p.port&)¶
        ¶
        POKE  p.io&+8,5¶
        POKEL p.io&+14,p.port&        ¶
        POKEW p.io&+28,11¶
        POKEL p.io&+32,f.rastport&¶
        POKEL p.io&+36,f.colormap&¶
        POKEL p.io&+40,f.vp.mode%¶
        POKEW p.io&+48,f.width%¶
        POKEW p.io&+50,f.height%¶
        POKEL p.io&+52,f.width%¶
        POKEL p.io&+56,f.height%¶
        POKEW p.io&+60,4¶
        ¶
        d.name$ = "printer.device"+CHR$(0)¶
        status% = OpenDevice%(SADD(d.name$),0,p.io&,0)¶
        IF status%<>0 THEN¶
          PRINT "Drucker is not free."¶
          CALL FreeMem(p.io&,240)¶
          CALL FreeSignal(p.sigBit%)¶
          EXIT SUB¶
        END IF¶
        ¶
        CALL SendIO(p.io&)¶
        IF PEEK(p.io&+31)<>0 THEN: iodelete¶
END SUB¶
¶
SUB iodelete STATIC¶
        SHARED p.io&,p.sigBit%,f.rastport&¶
        SHARED p.port&¶
        status%    = CheckIO%(p.io&)¶
        IF status% = 0 THEN¶
          PRINT "Printer still in use!"¶
          PRINT "Please wait!"¶
        END IF  ¶
        ercond%    = WaitIO%(p.io&)¶
        l.bitmap&  = PEEKL(f.rastport&+4)¶
        l.x%       = PEEKW(l.bitmap&)*8¶
        l.y%       = PEEKW(l.bitmap&+2)¶
        l.depth%   = PEEK(l.bitmap&+5)¶
        FOR loop1%=1 TO l.depth%¶
          l.plane& = PEEKL(l.bitmap&+4+4*loop1%)¶
          CALL FreeRaster(l.plane&,l.x%,l.y%)¶
        NEXT loop1%  ¶
        ¶
        CALL CloseDevice(p.io&)¶
        CALL RemPort(p.port&)¶
        CALL FreeMem(p.io&,240)¶
        CALL FreeSignal(p.sigBit%)¶
        PRINT "Error Code: ";ercond%¶
END SUB¶
```

Since there is not enough room in this book for a step by step program explantion (this would be found in a book about the exec operating system), the following is a brief explanation of what the program does:

Because your program continues to work after calling `hardcopy`, the following things happen: the RastPort and bit-map, including all bit-planes, are copied to a memory buffer. This ensures that the printing will not be disturbed by anything else the program does. Most of this work is performed by the exec function `CopyMem` and the graphic routine `BltBitMap` (Note: Use Kickstart Version 1.2).

I/O block and ReplyPort are setup as required. We activate the printing with `SendIO`.

The SUB program `delete` is responsible for checking the printer to see if its finished. `CheckIO%` performs this function. If we receive a value of zero, the printer is still printing. `WaitIO` waits internally for the printing to finish and then sends the ReplyMessage. `WaitIO` also reads the error field of the I/O block and returns this value to the program.

Now we return the bit-planes and system structures to the system, close the printer and release the `Signalbit`.

7. The IFF-ILBM standard

"IFF" is the abbreviation for "Interchange File Format". This format evolved from a agreement between two companies, Electronic Arts and Commodore when the Amiga was just being introduced. These companies decided that it would not be practical for users or programmers if each graphic program saved graphics in a different format. If this happened, then it is possible that Graphicraft® pictures would only work with Graphicraft® or Aegis® pictures would only work with Aegis® software. Without a standard format, exchanging pictures between the programs would be impossible. This is the reason why the standard format ILBM (Interleaved Bitmap) was created. An ILBM file consists of many components. The following are the most important:

"BMHD" = BitmapHeader

Identification: BMHDnnnn

Offset	Type	Description
+ 000	Word	Width of graphic
+ 002	Word	Height of graphic
+ 004	Word	X position this graphic
+ 006	Word	Y position this graphic
+ 008	Byte	Number of bit-planes (depth)
+ 009	Byte	Masking
		0 = No masking
		1 = Masking
		2 = Transparent
		3 = Lasso
+ 010	Byte	Data compression
		0 = No compression
		1 = ByteRun1 algorithms
+ 011	Byte	unused
+ 012	Word	"transparent" color
+ 014	Byte	X aspect
+ 015	Byte	Y aspect
+ 016	Word	Width of source page
+ 018	Word	Height of source page

"CMAP" = ColorMap

Identification: CMAPnnnn

Offset	Type	Description	
+ 001	Byte	red	(0-255)
+ 002	Byte	green	(0-255)
+ 003	Byte	blue	(0-255)

"BODY" = Bit-planes

Identification: BODYnnnn

Bit-plane 1
Bit-plane 2
(...)
Bit-plane n

"CAMG" = Amiga ViewPort Mode (Hi-Res, Lo-Res, Lace, etc.)

Identification: CAMGnnnn

Offset	Type	Description
+ 000	Long	ViewPort mode (see Chapter 4)

Same as the colorcycle information from Graphicraft:

"CCRT" - Graphicraft Colorcycle Data

Identification: "CCRTnnnn"

Offset	Type	Description
+ 000	Word	Direction 0=backwards 1=forwards
+ 002	Byte	Start (number of color register: 0-31)
+ 003	Byte	End (number of color register: 0-31)
+ 004	Long	Seconds
+ 008	Long	Micro seconds

These important data blocks represent the ILBM format file for every graphic saved using this method.

The following program demonstrates how to load and display this type of file. It is possible for you to load any ILBM graphic that you want. For example, you can load a graphic that was created with your drawing program or an ILBM graphic from any other type of program.

The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 7¶
'# Program: Load ILBM Picture from disk¶
'# Date:    01/15/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'# Loads pictures of any mode, including¶
'# HAM, HalfBrite and Graphicraft Color¶
'# Cycle.¶
'#¶
'# Based on the ILBM IFF Interleaved¶
'# Bitmap Standard published November¶
'# 15, 1985 by Jerry Morrison (from¶
'# Electronic Arts) im the "Commodore¶
'# Amiga ROM Kernel Manual Volume 2", ¶
'# CBM Prod.-Nr. 327271-02 rev 2 from¶
'# September 12, 1985, Page H-25.¶
'#¶
'########################################¶
¶
¶
PRINT "Searching for .bmap files..."¶
¶
'DOS-Library¶
DECLARE FUNCTION xOpen& LIBRARY¶
DECLARE FUNCTION xRead& LIBRARY¶
DECLARE FUNCTION Seek& LIBRARY¶
'xClose()¶
'Delay()¶
¶
'EXEC-Library¶
¶
DECLARE FUNCTION AllocMem& LIBRARY¶
DECLARE FUNCTION DoIO% LIBRARY¶
DECLARE FUNCTION OpenDevice% LIBRARY¶
DECLARE FUNCTION AllocSignal% LIBRARY¶
DECLARE FUNCTION FindTask& LIBRARY¶
'FreeMem()¶
¶
'GRAPHICS-Library¶
DECLARE FUNCTION AllocRaster& LIBRARY¶
'SetRast()¶
'LoadRGB4()¶
¶
LIBRARY "dos.library"¶
LIBRARY "exec.library"¶
LIBRARY "graphics.library"¶
¶
main:       '* Load¶
```

```
            CLS¶
            PRINT "ILBM Loader"¶
            PRINT¶
            PRINT "Loads Standard IFF Graphic files
into"¶
            PRINT "memory and displays the picture."¶
            PRINT¶
            PRINT "The IFF Loader supports Graficraft
Color-Cycling,"¶
            PRINT "the display of Hold-And-Modify (4096
Colors)"¶
            PRINT "and Halfbrite (64 Colors)."¶
            PRINT¶
            PRINT "In accordance with ILBM standards it
decodes packed"¶
            PRINT "graphic files that use the ByteRun1
method and"¶
            PRINT "displays them."¶
            PRINT¶
            PRINT "If desired you can print the picture
on a graphics"¶
            PRINT "capable printer."¶
            ¶
            LINE INPUT "Name of ILBM File: ";ilbm.file$¶
            LINE INPUT "Do you want the picture printed?
(y/n) ";yn$¶
            ¶
            LoadILBM ilbm.file$¶
            ColorCycle -1¶
            ¶
            IF yn$="y" THEN¶
              WINDOW OUTPUT 2¶
              Hardcopy¶
            END IF¶
              ¶
            WHILE INKEY$="":WEND¶
            ILBMend¶
            ¶
            LIBRARY CLOSE¶
¶
¶
SUB LoadILBM(ilbm.name$) STATIC¶
      SHARED disk.handle&,buf.read&¶
      SHARED buf.color&,buf.rgb&¶
      SHARED ilbm.error$,signal%     ¶
      SHARED screen.kolor%,amiga.viewport&¶
      SHARED ilbm.vp.mode&¶
      SHARED amiga.rastport&¶
      ¶
      disk.modeOldFile% = 1005      ¶
      disk.name$        = ilbm.name$+CHR$(0)¶
¶
     '* Open ILBM file¶
      disk.handle&=xOpen&(SADD(disk.name$),1005)¶
      IF disk.handle&=0 THEN¶
```

```
        ilbm.error$="ILBM File "+ilbm.name$+" not on
specified disk and drawer."¶
        GOTO ilbm.error.A¶
      END IF¶
      ¶
     '* Setup disk read buffer¶
      mem.opt&=2^0+2^16¶
      buf.size& = 240¶
      buf.add&  = AllocMem&(buf.size&,mem.opt&)¶
      IF buf.add&=0 THEN¶
        ilbm.error$="Not enough buffer memory free."¶
        GOTO ilbm.error.A¶
      END IF¶
      ¶
     '* Section Buffer for Chunk Areas¶
      buf.read&  = buf.add&+0*120¶
      buf.rgb&   = buf.add&+1*120¶
      ¶
     '* Is it an ILBM file?¶
      disk.wasread&=xRead&(disk.handle&,buf.read&,12)¶
      ilbm.ID.$   = ""¶
      FOR loop1%  = 8 TO 11¶
        ilbm.ID.$ = ilbm.ID.$+CHR$(PEEK(buf.read&+
loop1%))¶
      NEXT loop1%¶
      ¶
      IF ilbm.ID.$<>"ILBM" THEN¶
        ilbm.error$="File "+ilbm.name$+" is not an ILBM
file."¶
        GOTO ilbm.error.A¶
      END IF¶
      ¶
     '* read Data Chunks from ILBM¶
      WHILE (signal%<>1) AND (ilbm.error$="")¶
        ReadChunk¶
      WEND¶
      ¶
     '* Error?¶
      ilbm.error.A:¶
      IF ilbm.error$<>"" THEN¶
        WINDOW CLOSE 2¶
        SCREEN CLOSE 1¶
        PRINT ilbm.error$¶
        EXIT SUB¶
      END IF¶
      ¶
     '* all ok!¶
      CALL LoadRGB4(amiga.viewport&,buf.rgb&,
screen.kolor%)¶
¶
     '* Close ILBM file?¶
      IF disk.handle&<>0 THEN¶
        CALL xClose(disk.handle&)¶
      END IF¶
      IF buf.add&<>0 THEN¶
        CALL FreeMem(buf.add&,buf.size&)¶
```

```
        buf.add&=0¶
      END IF¶
¶
     '* Set Viewmodes¶
      POKEW amiga.viewport&+32,ilbm.vp.mode&¶
END SUB¶
¶
SUB ILBMend STATIC¶
      WINDOW CLOSE 2¶
      SCREEN CLOSE 1¶
END SUB¶
        ¶
SUB ColorCycle(mode%) STATIC¶
      SHARED ccrt.direction%¶
      SHARED ccrt.start%,amiga.colortable& ¶
      SHARED ccrt.end%,screen.kolor%¶
      SHARED ccrt.secs&,status%     ¶
      SHARED ccrt.mics&,amiga.viewport&¶
     ¶
     '* as intended?¶
      IF (status% AND 2^4)<>2^4 THEN¶
        EXIT SUB¶
      END IF¶
     ¶
     '* Setup Variable fields¶
      DIM kolor.original%(screen.kolor%-1)¶
      DIM kolor.actual%(screen.kolor%-1)¶
      ¶
     '* all ok, save old color from Viewport¶
      FOR loop1%=0 TO screen.kolor%-1¶
        kolor.original%(loop1%)=
PEEKW(amiga.colortable&+2*loop1%)¶
        kolor.actual%(loop1%)=kolor.original%(loop1%)¶
      NEXT loop1%¶
      ¶
     '* Color Cycling¶
      WHILE mode%<>0¶
       '* Mode?¶
        IF mode%<0 THEN¶
          in$=INKEY$¶
          IF in$<>"" THEN mode%=0¶
        ELSE¶
          mode%=mode%-1¶
        END IF¶
        ¶
       '* forwards?¶
        IF ccrt.direction%=1 THEN¶
          ccrt.backup%=kolor.actual%(ccrt.start%)¶
          FOR loop1%=ccrt.start%+1 TO ccrt.end%¶
            kolor.actual%(loop1%-
1)=kolor.actual%(loop1%)¶
          NEXT loop1%¶
          kolor.actual%(ccrt.end%)=ccrt.backup%¶
       ¶
       '* backwards?¶
        ELSE¶
```

```
          ccrt.backup%=kolor.actual%(ccrt.end%)¶
          FOR loop1%=ccrt.start%-1 TO ccrt.end% STEP -1¶
            kolor.actual%(loop1%+1)=
kolor.actual%(loop1%)¶
          NEXT loop1%¶
          kolor.actual%(ccrt.start%)=ccrt.backup%¶
                ¶
        END IF¶
        CALL LoadRGB4(amiga.viewport&,VARPTR
(kolor.actual%(0)),screen.kolor%)¶
        timeout&=50*(ccrt.secs&+(ccrt.mics&/1000000&))¶
        CALL Delay(timeout&)¶
      WEND¶
      ¶
     '* Restore original colors¶
      CALL LoadRGB4(amiga.viewport&,VARPTR
(kolor.original%(0)),screen.kolor%)¶
¶
     '* Return fields¶
      ERASE kolor.original%¶
      ERASE kolor.actual%¶
END SUB¶
      ¶
SUB ReadChunk STATIC¶
      SHARED disk.handle&,buf.read&¶
      SHARED buf.color&,buf.rgb&¶
      SHARED ilbm.error$,signal%    ¶
      SHARED screen.kolor%,amiga.viewport&¶
      SHARED ilbm.vp.mode&,status%¶
      SHARED amiga.rastport&¶
      SHARED ccrt.direction%¶
      SHARED ccrt.start%,amiga.colortable& ¶
      SHARED ccrt.end%,screen.kolor%¶
      SHARED ccrt.secs&    ¶
      SHARED ccrt.mics&¶
      ¶
     '* Read Chunk Header¶
      disk.wasread& = xRead&(disk.handle&,buf.read&,8)¶
      ilbm.chunk&   = PEEKL(buf.read&+4)¶
      ilbm.ID.$   = ""¶
      FOR loop1%  = 0 TO 3¶
        ilbm.ID.$ =
ilbm.ID.$+CHR$(PEEK(buf.read&+loop1%))¶
      NEXT loop1%¶
¶
     '* The BitMap Header (BMHD) ?¶
      IF ilbm.ID.$="BMHD" THEN¶
       '* Read Chunk contents¶
        disk.wasread&=xRead&(disk.handle&,
buf.read&,ilbm.chunk&)¶
     ¶
        status%=status% OR 2^0¶
        ilbm.width%   = PEEKW(buf.read&+0)¶
        ilbm.height%  = PEEKW(buf.read&+2)¶
        ilbm.depth%   = PEEK (buf.read&+8)¶
        ilbm.mode%    = PEEK (buf.read&+10)¶
```

```
        screen.width%    = PEEKW(buf.read&+16)¶
        screen.height%   = PEEKW(buf.read&+18)¶
       ¶
       '* Build display parameters¶
        ilbm.bytes%   = ilbm.width%/8¶
        screen.bytes% = screen.width%/8¶
        screen.kolor% = 2^(ilbm.depth%)¶
        ¶
       '* prepare all for display¶
       ¶
       '* HiRes (High Resolution?)¶
        IF screen.width%>320 THEN¶
          screen.mode%=2¶
        ELSE¶
          screen.mode%=1¶
        END IF¶
                    ¶
       '* Interlace (y=0 - 399)¶
        IF screen.height%>200 THEN
screen.mode%=screen.mode%+2¶
        ¶
       '* ilbm.depth%=6 -> HAM/Halfbrite?¶
        IF ilbm.depth%=6 THEN¶
          ilbm.reg%=-1¶
        END IF¶
        ¶
        depth%=ilbm.depth%+ilbm.reg%¶
        SCREEN 1,screen.width%,screen.height%,
depth%,screen.mode%¶
        WINDOW 2,,,0,1¶
¶
       '* System Parameters¶
        amiga.windo&      = WINDOW(7)¶
        amiga.screen&     = PEEKL(amiga.windo&+46)¶
        amiga.viewport&   = amiga.screen&+44¶
        amiga.rastport&   = amiga.screen&+84¶
        amiga.colormap&   = PEEKL(amiga.viewport&+4)¶
        amiga.colortable& = PEEKL(amiga.colormap&+4)¶
        amiga.bitmap&     = PEEKL(amiga.rastport&+4)¶
        FOR loop1%=0 TO ilbm.depth%-1¶
          amiga.plane&(loop1%)=
PEEKL(amiga.bitmap&+8+loop1%*4)¶
        NEXT loop1%¶
        ¶
       '* for HAM/Halfbrite prepare 6 bitplanes¶
        IF ilbm.reg%=-1 THEN¶
          ilbm.reg%=0¶
          newplane&=AllocRaster&(screen.width%,
screen.height%)¶
          IF newplane&=0 THEN¶
            ilbm.error$="Not enough memory free for 6
bitplanes!"¶
          ELSE¶
            POKE  amiga.bitmap&+5,6¶
            POKEL amiga.bitmap&+28,newplane&¶
          END IF¶
```

```
        END IF¶
                ¶
     '* Color Table (CMAP) ?¶
      ELSEIF ilbm.ID.$="CMAP" THEN¶
       '* Read Chunk contents¶
        disk.wasread&=xRead&(disk.handle&,
buf.read&,ilbm.chunk&)¶
     ¶
        status%=status% OR 2^1  ¶
¶
       '* Calculate RGB Table¶
        FOR loop1% = 0 TO screen.kolor% -1¶
          kolor.red%   = PEEK(buf.read&+loop1%*3+0)¶
          kolor.green% = PEEK(buf.read&+loop1%*3+1)¶
          kolor.blue%  = PEEK(buf.read&+loop1%*3+2)¶
          kolor.rgb%   =
kolor.green%+16*kolor.red%+1/16*kolor.blue%¶
          POKEW buf.rgb&+2*loop1%,kolor.rgb%¶
        NEXT loop1%¶
        ¶
       '* Alignment¶
        IF (ilbm.chunk OR 1)=ilbm.chunk THEN¶
          disk.wasread&=xRead&(disk.handle&,buf.read&,1)¶
        END IF¶
           ¶
     '* Viewport Mode (CAMG) AMIGA ?¶
      ELSEIF ilbm.ID.$="CAMG" THEN¶
       '* Read Chunk contents¶
        disk.wasread&=xRead&(disk.handle&,
buf.read&,ilbm.chunk&)¶
     ¶
        status%=status% OR 2^3¶
        ilbm.vp.mode& = PEEKL(buf.read&)¶
        ¶
     '* Color-Cycle Data (CCRT) ?¶
      ELSEIF ilbm.ID.$="CCRT" THEN¶
       '* Read Chunk contents¶
        disk.wasread&=xRead&(disk.handle&,
buf.read&,ilbm.chunk&)¶
 ¶
        status%=status% OR 2^4¶
        ccrt.direction% = PEEKW(buf.read&+0)¶
        ccrt.start%     = PEEK (buf.read&+2)¶
        ccrt.end%       = PEEK (buf.read&+3)¶
        ccrt.secs&      = PEEKL(buf.read&+4)¶
        ccrt.mics&      = PEEKL(buf.read&+8)¶
        ¶
     '* Bitplanes (BODY) ?¶
      ELSEIF ilbm.ID.$="BODY" THEN¶
        status%=status% OR 2^2¶
        ¶
       '* not compressed data¶
        IF ilbm.mode%=0 THEN¶
          FOR loop1%=0 TO ilbm.height%-1¶
            FOR loop2%=0 TO ilbm.depth%-1¶
```

```
                screen.buffer&=amiga.plane&(loop2%)+
(loop1%*screen.bytes%)¶
                disk.wasread&=xRead&(disk.handle&,
screen.buffer&,ilbm.bytes%)¶
              NEXT loop2%¶
            NEXT loop1%¶
                ¶
          '* compressed Data (ByteRun1-Encoding)¶
           ELSEIF ilbm.mode%=1 THEN¶
             FOR loop1%=0 TO ilbm.height%-1¶
               FOR loop2%=0 TO ilbm.depth%-1¶
                 screen.buffer&=amiga.plane&(loop2%)+
(loop1%*screen.bytes%)¶
                 counter%=0¶
                 ¶
                '* Decoding¶
                 WHILE counter%<ilbm.bytes%¶
                   disk.wasread& =
xRead&(disk.handle&,buf.read&,1)¶
                   code%          = PEEK(buf.read&)¶
                  '* Code 1: read n Bytes uncoded¶
                   IF code%<128 THEN¶
                     disk.wasread& =
xRead&(disk.handle&,screen.buffer&+counter%,code%+1)¶
                     counter%       = counter%+code%+1¶
                  '* Code 2: repeat next Byte (257-n) times¶
                   ELSEIF code%>128 THEN¶
                     disk.wasread& =
xRead&(disk.handle&,buf.read&,1)¶
                     disk.byte%     = PEEK(buf.read&)¶
                     FOR loop3%=counter% TO counter%+257-
code%¶
                       POKE
screen.buffer&+loop3%,disk.byte%¶
                     NEXT loop3%¶
                     counter%=counter%+257-code%¶
                  '* Code 3: no operation¶
                   ELSE¶
                     '* no operation¶
                   END IF¶
                 WEND¶
               NEXT loop2%¶
             NEXT loop1%¶
          ¶
          '* different Decoding method¶
           ELSE¶
             ilbm.error$="Data Compression algorithm
unknown."¶
           END IF¶
        ¶
        '* process unimportant Chunks (GRAB, DEST, SPRT,
etc.)¶
         ELSE¶
          '* read straight count bytes¶
          IF (ilbm.chunk% OR 1)=ilbm.chunk% THEN¶
            ilbm.chunk%=ilb.chunk%+1¶
```

```
        END IF¶
        ¶
       '* Move Disk Cursor¶
        mode.current%=0¶
        stat&=Seek&(disk.handle&,ilbm.chunk%,0)¶
        IF stat&=-1 THEN¶
          ilbm.error$="DOS Error. Seek() failed."¶
        END IF¶
      ¶
      END IF¶
      ¶
     '* Error Check¶
      IF disk.wasread&<0 THEN¶
        ilbm.error$="DOS Error. Read() failed."¶
     '* EOF (End-Of-File) reached?¶
      ELSEIF disk.wasread&=0  AND ((status% AND 7)<>7
THEN¶
        ilbm.error$="ILBM Data Chunks not present."¶
        signal%=1¶
      ELSEIF (status% AND 7)=7 THEN¶
        signal%=1  ¶
      END IF¶
END SUB                                                ¶
¶
'* This is the Hardcopy I program from this book,¶
'* integrated with the ILBM program.¶
¶
SUB Hardcopy STATIC¶
      mem.opt& = 2^0+2^16¶
      p.io&    = AllocMem&(100,mem.opt&)¶
      p.port&  = p.io&+62¶
      IF p.io& = 0 THEN ERROR 7¶
¶
      f.windo&     = WINDOW(7)¶
      f.rastport&  = PEEKL(f.windo&+50)¶
      f.width%     = PEEKW(f.windo&+112)¶
      f.height%    = PEEKW(f.windo&+114)¶
      f.screen&    = PEEKL(f.windo&+46)¶
      f.viewport&  = f.screen&+44¶
      f.colormap&  = PEEKL(f.viewport&+4)¶
      f.vp.mode%   = PEEKW(f.viewport&+32)¶
      ¶
      ¶
      p.sigBit% = AllocSignal%(-1)¶
      IF p.sigBit% = -1 THEN¶
        PRINT "No Signalbit free!"¶
        CALL FreeMem(p.io&,100)¶
        EXIT SUB¶
      END IF¶
      p.sigTask& = FindTask&(0)¶
      ¶
      POKE  p.port&+8,4¶
      POKEL p.port&+10,p.port&+34¶
      POKE  p.port&+15,p.sigBit%¶
      POKEL p.port&+16,p.sigTask&¶
      POKEL p.port&+20,p.port&+24¶
```

```
        POKEL p.port&+28,p.port&+20¶
        POKE  p.port&+34,ASC("P")¶
        POKE  p.port&+35,ASC("R")¶
        POKE  p.port&+36,ASC("T")¶
        ¶
        CALL AddPort(p.port&)¶
        ¶
        POKE  p.io&+8,5¶
        POKEL p.io&+14,p.port&          ¶
        POKEW p.io&+28,11¶
        POKEL p.io&+32,f.rastport&¶
        POKEL p.io&+36,f.colormap&¶
        POKEL p.io&+40,f.vp.mode%¶
        POKEW p.io&+48,f.width%¶
        POKEW p.io&+50,f.height%¶
        POKEL p.io&+52,f.width%¶
        POKEL p.io&+56,f.height%¶
        POKEW p.io&+60,4¶
        ¶
        d.name$ = "printer.device"+CHR$(0)¶
        status% = OpenDevice%(SADD(d.name$),0,p.io&,0)¶
        IF status%<>0 THEN¶
          PRINT "Printer is not free."¶
          CALL  FreeMem(p.io&,100)¶
          CALL  FreeSignal(p.sigBit%)¶
          EXIT SUB¶
        END IF¶
        ¶
        ercond% = DoIO%(p.io&)¶
        ¶
        CALL CloseDevice(p.io&)¶
        CALL RemPort(p.port&)¶
        CALL FreeMem(p.io&,100)¶
        CALL FreeSignal(p.sigBit%)¶
        PRINT "Error Code: ";ercond%¶
END SUB¶
```

Program Description

The SUB program `LoadILBM` requires a string containing the name of the ILBM picture you want to load from disk. The picture must be in the active disk directory in order for the program to find it (`CHDIR` directory). The Extras diskette for Workbench 1.2 contains the `Heart.ILBM` file, which is a good example picture to use This SUB program loads and displays the picture. Now you could call the SUB program `ColorCycle`. When this SUB program finds a "CCRT ColorCycle" data block, it activates the cycling, which creates a motion-like effect on the screen. This SUB program requires an integer argument. If the argument is negative, the Amiga cycles the colors until you press a key. If the argument is positive, the Amiga cycles the colors for a time period determined by your argument. The SUB program `ILBMend` stops the display and closes both the screen and window.

The DOS library routines are new in our program. You cannot use the built in OPEN/INPUT#/CLOSE commands when working with ILBM files because these commands place zeros in the data. Here is a short explanation of the DOS routines used:

```
name$=name$+CHR$(0)
disk.handle&=xOpen&(SADD(name$),1005)
```

name$:	Name of file to be opened.
1005:	ModeOldFile - file already exists.
1006:	ModeNewFile - make file with this name.
disk.handle&:	BPTR (Pointer/4) in handler data block when 0, then xOpen failed.

```
wasread&=xRead(disk.handle&,buffer&,bytes&)
```

disk.handle&:	Address from xOpen call.
buffer&:	Address of a free memory area.
bytes&:	Number of bytes to be read from the actual disk cursor position that have to fit in the buffer!
wasread&:	Number of bytes read. =0: EOF (End Of File) smaller than 0: read error

```
oldpos&=Seek(disk.handle&,offset%,mode%)
```

disk.handle&:	Address from xOpen call
offset%:	Number of bytes to move the disk cursor
mode%:	0 = from current position -1 = from file begin 1 = from file end

```
CALL xClose(disk.handle&)
```

disk.handle&:	Handle from xOpen command; closes file

```
CALL delay(ticks)
```

tick	= 1/50 second
Microseconds	= 1/1000000 second

Waits for the specified time (however, this does not mean "busy-waiting", the system has additional computing time free).

Special features of the program:

This program not only supports the AmigaBASIC display modes hi-res and interlace, but also the ILBM graphics in halfbrite (64 colors) and HAM mode (4096 colors). Both modes use six bit-planes. Whenever one of these modes is encountered, a sixth bit-plane is added to the screen. However we do not delete this bit-plane using `FreeRaster`. Instead, when the `SCREEN CLOSE` statement closes the new screen, all the bit-planes, including the sixth bit-plane that was added, are automatically deleted.

The program is capable of decoding compacted bit-planes by using the "`ByteRun1`" method. This method uses two control codes. When the first byte read is smaller than 128, then the following byte is simply loaded and used. When the first byte is a value greater than 128, the second byte is repeated 257 times (normally signed bytes from -127 to +128 are used for a more complicated compression algorithm). When the byte is equal to 128 nothing happens because it is a NOP (No Operation).

8. A 1024x1024 paint program

In the previous chapters, you learned about the Amiga's various system components and how to program them. To conclude the BASIC portion of our book, we have included a complete paint program that uses superbitmap layers. The program also includes the capability of using the Amiga fonts, including their various modes and styles. Here is a brief description of the program and what it can do:

- Full mouse and menu control
- Up to 1024x1024 pixel sized drawings
- Soft scrolling over the entire drawing area
- Circles, lines, rectangles in true rubberband technique
- Freehand drawing
- Text output in JAM1, JAM2, complement and inverse
- Up to 19 different fonts at the same time
- Outline, italic, bold and underline text
- Extensive hardcopy functions:
- Printing of the entire 1024x1024 pixel graphic
- Selected piece with enlargement/reduction
- Distortion
- Fill areas
- Drawing grid
- Block erase
- Copy a piece of the drawing
- Self defined brushes and patterns

Since the superbitmap planes are so large, this drawing program uses only one bit-plane. Because of this, your drawings are limited to black and white. We designed the program to create drawings that you can print. The large drawing area makes it easy to create very detailed graphics that you can then print on a graphics-capable printer in their original or a reduced size.

The ¶ characters in the following program listing signify the end of a BASIC program line. Some lines were split when the program was formatted for this book.

```
'########################################¶
'#¶
'# Section: 8 ¶
'# Program: Superbitmap Paint Program¶
'# Date:    04/16/87¶
'# Author:  tob¶
'# Version: 1.0¶
'#¶
'########################################¶
¶
PRINT "Searching for .bmap files..."¶
¶
'GRAPHICS-Library¶
DECLARE FUNCTION AskSoftStyle& LIBRARY¶
DECLARE FUNCTION SetSoftStyle& LIBRARY¶
DECLARE FUNCTION OpenFont& LIBRARY¶
DECLARE FUNCTION AllocRaster& LIBRARY¶
¶
'EXEC-Library¶
DECLARE FUNCTION DoIO& LIBRARY¶
DECLARE FUNCTION OpenDevice& LIBRARY¶
DECLARE FUNCTION AllocSignal& LIBRARY¶
DECLARE FUNCTION FindTask& LIBRARY¶
DECLARE FUNCTION AllocMem& LIBRARY¶
¶
'DISKFONT-Library¶
DECLARE FUNCTION OpenDiskFont& LIBRARY¶
DECLARE FUNCTION AvailFonts& LIBRARY¶
¶
'LAYERS-Library¶
DECLARE FUNCTION CreateBehindLayer& LIBRARY¶
DECLARE FUNCTION UpFrontLayer& LIBRARY¶
DECLARE FUNCTION BehindLayer& LIBRARY¶
¶
LIBRARY "layers.library"¶
LIBRARY "graphics.library"¶
LIBRARY "exec.library"¶
LIBRARY "intuition.library"¶
LIBRARY "diskfont.library"¶
¶
setup:     '* Here we go:¶
           PRINT "Paint-1024 Drawing Program"¶
           PRINT "--------------------------"¶
           PRINT¶
           PRINT "Do you want to work with a LoRes(1) or
HiRes(2) Screen"¶
           PRINT "(has no effect on the size of the drawing
area) ?"¶
           PRINT¶
           LINE INPUT "Enter Choice (1 or 2) --> ";yn$¶
           IF yn$="2" THEN¶
             scrWidth% = 640¶
             scrMode%  =   2¶
           ELSE¶
             scrWidth% = 320¶
             scrMode%  =   1¶
           END IF¶
           ¶
initPar:   '* Screen Parameter¶
           scrHeight%   = 200¶
           scrDepth%    =   1¶
           scrNr%       =   1¶
```

```
                WBenchScrNr% =  -1¶
                ¶
               '* Window Parameters¶
                windWidth%  = scrWidth%-9¶
                windHeight% = scrHeight%-26¶
                windNr%     = 1¶
                windTitle$  = "Working Area"¶
                windMode%   = 16¶
                ¶
               '* Window Gadgets¶
                Xoffset%      = 15¶
                GadX%         = windWidth%-Xoffset%+3¶
                GadY%         = 11¶
                GadSX%        = Xoffset%-3¶
                GadSY%        = GadSX%-1¶
                GadNumber%    = 5¶
                GadTolerance% = 1¶
                Gad$(0)       = "^"¶
                Gad$(1)       = "v"¶
                Gad$(2)       = "<"¶
                Gad$(3)       = ">"¶
                Gad$(4)       = "H"¶
                ¶
               '* CAD Super Bitmap¶
                superWidth%  = 800¶
                superHeight% = 400¶
                superFlag%   =   4¶
                ¶
               '* Layer Size¶
                layMinX% =  3¶
                layMinY% = 11¶
                layMaxX% = windWidth%-8-Xoffset%¶
                layMaxY% = windHeight%¶
               ¶
               '* Drawing Mode¶
                draw%       = 4¶
                mode$       = "FREEHAND"¶
                drMd%       = 0¶
                style%      = 0¶
                swapper%    = 1¶
                kl%         = 1¶
                grid1%      = 1¶
                grid2%      = 1¶
                fontHeight% = 8¶
                DIM get.array%(1)¶
                ¶
               '* Printer Parameters¶
                printX0%   = 0¶
                printY0%   = 0¶
                printX1%   = superWidth%¶
                printY1%   = superHeight%¶
                printSpec% = 4 ¶
                    ¶
initDisp:      '* Open Screen and Window¶
                SCREEN scrNr%,scrWidth%,scrHeight%,
scrDepth%,scrMode%¶
                WINDOW windNr%,windTitle$,(0,0)-
(windWidth%,windHeight%),windMode%,scrNr%¶
                WINDOW OUTPUT windNr%¶
                PALETTE 1,0,0,0¶
                PALETTE 0,1,1,1¶
                ¶
```

```
                DIM area.pat%(3):DIM full%(1)¶
                area.pat%(0) = &H1111¶
                area.pat%(1) = &H2222¶
                area.pat%(2) = &H4444¶
                area.pat%(3) = &H8888¶
                PATTERN ,area.pat%¶
                PAINT (100,50),1,1¶
                full%(0)=&HFFFF¶
                full%(1)=full%(0)¶
                PATTERN ,full%¶
               '* Prepare TmpRas¶
               ¶
                buffersize& = superWidth%*superHeight%/8¶
                buffer&     = PEEKL(WINDOW(8)+12)¶
                IF buffer&<>0 THEN¶
                  fillflag% = 1¶
                  mem&  = PEEKL(buffer&)¶
                  size& = PEEKL(buffer&+4)¶
                  CALL FreeMem(mem&,size&)          ¶
                  opt&  = 2^0+2^1+2^16¶
                  buf&  = AllocMem&(buffersize&,opt&)¶
                  IF buf&=0 THEN ¶
                    fillflag%=0¶
                    POKEL WINDOW(8)+12,0¶
                  ELSE¶
                    POKEL buffer&,buf&¶
                    POKEL buffer&+4,buffersize&¶
                  END IF    ¶
                ELSE¶
                  fillflag%=0¶
                END IF¶
                ¶
                 ¶
initSys:       '* Read System Parameters¶
                windAdd&      = WINDOW(7)¶
                scrAdd&       = PEEKL(windAdd&+46)¶
                scrViewPort&  = scrAdd&+44¶
                scrColMap&    = PEEKL(scrViewPort&+4)¶
                scrBitMap&    = scrAdd&+184¶
                scrLayerInfo& = scrAdd&+224¶
                scrMode%      = PEEKW(scrViewPort&+32)¶
                font&         = PEEKL(WINDOW(8)+52)¶
                ¶
initSBMap:     '* Create Superbitmap¶
                opt&         = 2^0+2^1+2^16¶
                superBitmap& = AllocMem&(40,opt&)¶
                IF superBitmap&=0 THEN¶
                  PRINT "Hm. Not even 40 Bytes?"¶
                  ERROR 7¶
                END IF¶
                ¶
               '* Activate Superbitmap¶
                CALL InitBitMap(superBitmap&,scrDepth%,
superWidth%,superHeight%)¶
                superPlane& =
AllocRaster&(superWidth%,superHeight%)¶
                IF superPlane& = 0 THEN¶
                  PRINT "No Room!"¶
                  CALL FreeMem(superBitmap&,40)¶
                  ERROR 7¶
                END IF¶
                POKEL superBitmap&+8,superPlane&¶
```

```
          ¶
          '* Open Superbitmap Layer¶
          SuperLayer& = CreateBehindLayer&(scrLayerInfo&,
scrBitMap&,layMinX%,layMinY%,layMaxX%,layMaxY%,superFlag%,superB
itmap&)¶
          IF SuperLayer& = 0 THEN¶
            PRINT "Unable to create Layer!"¶
            CALL FreeRaster(superPlane&,superWidth%,
superHeight%)¶
            CALL FreeMem(superBitmap&,40)¶
            ERROR 7¶
          END IF¶
          ¶
          '* New RastPort¶
          SuperRast& = PEEKL(SuperLayer&+12)¶
¶
initPrint: '* Printer initialization¶
          opt& = 2^0+2^16¶
          pio& = AllocMem&(100,opt&)¶
          IF pio&<>0 THEN¶
            port&   = pio&+62¶
            sigBit% = AllocSignal&(-1)¶
            IF sigBit%<>-1 THEN¶
              sigTask& = FindTask&(0)¶
              POKE  port&+8,4¶
              POKEL port&+10,port&+34¶
              POKE  port&+15,sigBit%¶
              POKEL port&+16,sigTask&¶
              POKEL port&+20,port&+24¶
              POKEL port&+28,port&+20¶
              POKEL port&+34,1347572736&¶
              CALL AddPort(port&)¶
              POKE  pio&+8,5¶
              POKEL pio&+14,port&¶
              POKEW pio&+18,12¶
              POKEW pio&+28,11¶
              POKEL pio&+32,SuperRast&¶
              POKEL pio&+36,scrColMap&¶
              POKEL pio&+40,scrMode%¶
              POKEW pio&+48,superWidth%¶
              POKEW pio&+50,superHeight%¶
              POKEL pio&+52,superWidth%¶
              POKEL pio&+56,superHeight%¶
              POKEW pio&+60,4¶
            ELSE¶
              printflag% = 1¶
              CALL FreeMem(pio&,100)¶
            END IF¶
          ELSE¶
            printflag% = 1¶
          END IF¶
          ¶
prepare:  '* Draw Move Gadgets¶
          CALL SetDrMd(WINDOW(8),5)¶
          FOR loop% = 0 TO GadNumber%-1¶
            LINE  (GadX%,GadY%+(GadSY%+5)*loop%)-
(GadX%+GadSX%,GadY%+GadSY%+(GadSY%+5)*loop%),1,bf¶
            gadMouse%(loop%) = GadY%+(GadSY%+5)*loop%-4-
GadTolerance%¶
            CALL  Move(WINDOW(8),GadX%+3,GadY%+
((GadSY%+5)*loop%)+8)¶
            PRINT Gad$(loop%)¶
```

```
              NEXT loop%¶
              CALL SetDrMd(WINDOW(8),1)¶
¶
             '* Prepare Drawing Area¶
              CALL SetRast(SuperRast&,0)¶
             ¶
             '* Draw Calibrations¶
              FOR loop% = 0 TO windWidth%-Xoffset% STEP 3¶
                IF loop%/15 = INT(loop%/15) THEN¶
                  LINE (loop%,windHeight%)-(loop%,windHeight%-10)¶
                ELSE¶
                  LINE (loop%,windHeight%)-(loop%,windHeight%-5)¶
                END IF¶
              NEXT loop%¶
              FOR loop% = 0 TO windHeight% STEP 2¶
                IF loop%/10 = INT(loop%/10) THEN¶
                  LINE (windWidth%-Xoffset%,loop%)-(windWidth%-10-
Xoffset%,loop%)¶
                ELSE¶
                  LINE (windWidth%-Xoffset%,loop%)-(windWidth%-5-
Xoffset%,loop%)¶
                END IF¶
              NEXT loop%¶
              ¶
             '* Display Layer¶
              e& = UpFrontLayer&(scrLayerInfo&,SuperLayer&)¶
              GOSUB newpointer¶
                                        ¶
             '* Protect and Integrate Layer¶
              POKEL SuperLayer&+40,windAdd&    ¶
              backup.rast&  = PEEKL(SuperLayer&+12)¶
              backup.layer& = PEEKL(WINDOW(8))¶
              POKEL SuperLayer&+12,WINDOW(8)¶
              POKEL WINDOW(8),SuperLayer&¶
              SuperRast&=WINDOW(8)¶
              ¶
              GOSUB coord¶
              ¶
             '* Menu Control¶
              MENU 1,0,1,"Service"¶
                MENU 1,1,1,"Clear Screen    "¶
                MENU 1,2,1,"Coordinates On  "¶
                MENU 1,3,1,"----------------"¶
                MENU 1,4,1,"Transparent     "¶
                MENU 1,5,1,"JAM 2           "¶
                MENU 1,6,1,"Complement      "¶
                MENU 1,7,1,"Inverse         "¶
                MENU 1,8,1,"----------------"¶
                MENU 1,9,1,"normal/reset    "¶
                MENU 1,10,1,"italic          "¶
                MENU 1,11,1,"bold            "¶
                MENU 1,12,1,"underline       "¶
                MENU 1,13,1,"outline         "¶
                MENU 1,14,1,"----------------"¶
                MENU 1,15,1,"s/w -> w/s      "¶
                MENU 1,16,1,"Title Bar On/Off"¶
                MENU 1,17,1,"QUIT            "¶
              MENU 2,0,1,"Draw"¶
                MENU 2,1,1,"Circle     "¶
                MENU 2,2,1,"Square     "¶
                MENU 2,3,1,"Lines      "¶
                MENU 2,4,1,"Freehand   "¶
```

```
                    MENU 2,5,1,"Text       "¶
                    MENU 2,6,1,"Erase      "¶
                    MENU 2,7,fillflag%,"Fill       "¶
                    MENU 2,8,1,"Raster/Grid"¶
                    MENU 2,9,1,"Grid Reset "¶
                    MENU 2,10,1,"Get Area   "¶
                    MENU 2,11,1,"Paint Area "¶
                  MENU 3,0,1,"Font"¶
                    MENU 3,1,1,"Load Fonts"¶
                  MENU 4,0,1,"I/O"¶
                    MENU 4,1,1,"Print  "¶
                    MENU 4,2,1,"Param. "¶
¶
                  ON MENU GOSUB MenuCtrl¶
                  MENU ON¶
                       ¶
mcp:              '* Master Control Program¶
                  WHILE forever=forever¶
                    test%=MOUSE(0)¶
                    mx%=MOUSE(1)¶
                    my%=MOUSE(2)¶
                    GOSUB updateDisp¶
                    CALL SetDrMd(SuperRast&,drMd%)¶
                    enable%=AskSoftStyle&(SuperRast&)¶
                    n&=SetSoftStyle&(SuperRast&,style%,enable%)
¶
                    '* drawing!¶
                    IF mx%>layMinX% AND mx%<layMaxX% AND test%<0 THEN¶
                      IF draw%=4 THEN¶
                        GOSUB freedraw¶
                      ELSEIF draw%=10 THEN¶
                        GOSUB paintdraw  ¶
                      ELSEIF draw%=5 THEN¶
                        GOSUB drawtext¶
                      ELSEIF draw%=7 THEN¶
                        GOSUB filler  ¶
                      ELSE   ¶
                        GOSUB drawit¶
                        IF draw%=3 AND fetch%=1 THEN¶
                          printX0% = cX%+subox%¶
                          printX1% = 1+cX%+subox%+oldrcX%¶
                          printY0% = cY%+suboy%¶
                          printY1% = 1+cY%+suboy%+oldrcY%¶
                          GOTO continue¶
                        ELSEIF draw%=3 AND grid%=1 THEN¶
                          x1%=cX%+subox%¶
                          y1%=cY%+suboy%¶
                          x2%=cX%+subox%+oldrcX%¶
                          y2%=cY%+suboy%+oldrcY%¶
                          IF x1%>x2% THEN SWAP x1%,x2%¶
                          IF y1%>y2% THEN SWAP y1%,y2%¶
                          g.width%  = x2%-x1%¶
                          g.height% = y2%-y1%¶
                          IF copy%=0 THEN          ¶
                            grid1%=g.width%¶
                            grid2%=g.height%¶
                          ELSE¶
                            g.size&=6+(g.height%+1)*2*
INT((g.width%+16)/16)¶
                            IF g.size&>(FRE(0)-1000) THEN¶
                              BEEP¶
                            ELSE¶
```

```
                              ERASE get.array%¶
                              DIM get.array%(g.size&/2)¶
                              GET (x1%,y1%)-(x2%,y2%),get.array%¶
                            END IF¶
                          END IF¶
                        END IF¶
                      END IF¶
                    ELSEIF (test%<0 AND mx%>layMaxX%) THEN¶
                    '* Scroll Gadgets in use?¶
                      IF my%>gadMouse%(4) THEN¶
                        GOSUB ScrollHome¶
                      ELSEIF my%>gadMouse%(3) THEN '<-¶
                        GOSUB ScrollLeft¶
                      ELSEIF my%>gadMouse%(2) THEN '->¶
                        GOSUB ScrollRight¶
                      ELSEIF my%>gadMouse%(1) THEN 'up¶
                        GOSUB ScrollUp¶
                      ELSEIF my%>gadMouse%(0) THEN 'down¶
                        GOSUB ScrollDown¶
                      END IF¶
                    END IF¶
                  WEND¶
                  ¶
deleteSys:  '* remove System¶
                  buf&=PEEKL(WINDOW(8)+12)¶
                  IF buf&<>0 THEN¶
                    buffer&=PEEKL(buf&)¶
                    size&=PEEKL(buf&+4)¶
                    CALL FreeMem(buffer&,size&)¶
                    POKEL WINDOW(8)+12,0¶
                  END IF¶
                  IF ptr&<>0 THEN¶
                    CALL ClearPointer(WINDOW(7))¶
                    CALL FreeMem(ptr&,20)¶
                  END IF  ¶
                  POKEL SuperLayer&+12,backup.rast&¶
                  POKEL WINDOW(8),backup.layer&¶
                  POKEL SuperLayer&+40,0¶
                  ¶
                  CALL DeleteLayer(scrLayerInfo&,SuperLayer&)¶
                  CALL FreeRaster(superPlane&,superWidth%,
superHeight%)¶
                  CALL FreeMem(superBitmap&,40)¶
                  ¶
                  SCREEN CLOSE scrNr%¶
                  WINDOW windNr%,"hi!",,,WBenchScrNr%¶
                  ¶
                  IF printflag%<>1 THEN¶
                    CALL RemPort(port&)¶
                    CALL FreeSignal(sigBit%)¶
                    CALL FreeMem(pio&,100)¶
                  END IF¶
                  ¶
                  IF oldFont&<>0 THEN CALL CloseFont(oldFont&)  ¶
                  LIBRARY CLOSE¶
                  END¶
¶
'*** That was it. Here are the important Subroutines. ***¶
¶
MenuCtrl:  '* Menu in use¶
                  menuId   = MENU(0)¶
                  menuItem = MENU(1)¶
```

```
¶
                IF menuId=1 THEN¶
                  IF menuItem     =  1 THEN¶
                    CALL SetRast(SuperRast&,0)¶
                  ELSEIF menuItem =  2 THEN¶
                    GOSUB coord¶
                  ELSEIF menuItem =  4 THEN¶
                    drMd%=0¶
                  ELSEIF menuItem =  5 THEN¶
                    drMd%=drMd% OR 1¶
                  ELSEIF menuItem =  6 THEN¶
                    drMd%=drMd% OR 2¶
                  ELSEIF menuItem =  7 THEN¶
                    drMd%=drMd% OR 4¶
                  ELSEIF menuItem =  9 THEN¶
                    style%=0:drMd%=0:outline%=0¶
                  ELSEIF menuItem = 10 THEN¶
                    style%=style% OR 4¶
                  ELSEIF menuItem = 11 THEN¶
                    style%=style% OR 2¶
                  ELSEIF menuItem = 12 THEN¶
                    style%=style% OR 1¶
                  ELSEIF menuItem = 13 THEN¶
                    outline%=1¶
                  ELSEIF menuItem = 15 THEN¶
                    GOSUB swapcol¶
                  ELSEIF menuItem = 16 THEN¶
                    IF kl%=0 THEN¶
                      kl%=1¶
                    ELSE¶
                      kl%=0¶
                    END IF¶
                  ELSEIF menuItem = 17 THEN¶
                    GOTO deleteSys
¶
                  END IF¶
                ELSEIF menuId = 2 THEN¶
                  grid% = 0¶
                  copy% = 0¶
                  IF menuItem     = 1 THEN¶
                    mode$  = "CIRCLE"¶
                    draw%  = 1¶
                  ELSEIF menuItem = 2 THEN¶
                    mode$  = "SQUARE"¶
                    draw%  = 3¶
                  ELSEIF menuItem = 3 THEN¶
                    mode$  = "LINES"¶
                    draw%  = 2¶
                  ELSEIF menuItem = 4 THEN¶
                    mode$  = "FREEHAND"¶
                    draw%  = 4    ¶
                  ELSEIF menuItem = 5 THEN¶
                    mode$  = "TEXT"¶
                    draw%  = 5¶
                  ELSEIF menuItem = 6 THEN¶
                    mode$  = "DELETE"¶
                    draw%  = 6¶
                  ELSEIF menuItem = 7 THEN¶
                    mode$  = "FILL"¶
                    draw%  = 7¶
                  ELSEIF menuItem = 8 THEN¶
                    mode$  = "GRID"¶
```

```
                    grid%  = 1¶
                    draw%  = 3¶
                  ELSEIF menuItem = 9 THEN¶
                    grid1% = 1¶
                    grid2% = 1¶
                  ELSEIF menuItem = 10 THEN¶
                    mode$  = "GET AREA"¶
                    draw%  = 3¶
                    grid%  = 1¶
                    copy%  = 1¶
                  ELSEIF menuItem = 11 THEN¶
                    mode$  = "PAINT"¶
                    draw%  = 10¶
                  END IF¶
                ELSEIF menuId = 3 THEN¶
                  IF fontflag% = 0 THEN¶
                    GOSUB loadFonts¶
                  ELSE¶
                    GOSUB loadFont¶
                  END IF¶
                ELSEIF menuId=4 THEN¶
                  IF menuItem=1 THEN¶
                    IF printflag%<>1 THEN¶
                      GOSUB hardcopy¶
                    ELSE¶
                      BEEP¶
                    END IF¶
                  ELSEIF menuItem=2 THEN¶
                    GOSUB changePrint¶
                  END IF¶
                END IF¶
¶
                IF kl%=1 THEN¶
                  tboff$ = mode$+" / TitleBar Off"+CHR$(0)¶
                  CALL WaitTOF¶
                  CALL SetWindowTitles(windAdd&,SADD(tboff$),-1)¶
                END IF¶
                ¶
                RETURN¶
                                    ¶
coord:          '* Show Coordinates Intersection¶
                CALL SetDrMd(WINDOW(8),2)¶
                POKEW SuperRast&+34,&HAAAA ¶
                FOR loop%=0 TO superWidth% STEP 50¶
                  LINE (loop%,0)-(loop%,superHeight%)¶
                NEXT loop%¶
                FOR loop%=0 TO superHeight% STEP 50¶
                  LINE (0,loop%)-(superWidth%,loop%)¶
                NEXT loop%¶
                POKEW SuperRast&+34,&HFFFF¶
                CALL SetDrMd(WINDOW(8),drMd%)¶
                RETURN¶
                ¶
drawit:          '* Multi Function Drawing with Rubberbanding¶
                  cX%=MOUSE(1)¶
                  cY%=MOUSE(2)¶
                  test%=MOUSE(0)¶
                  mx%=1:my%=1¶
                  ccX%=0:ccY%=0¶
                  oldcX%=0:oldcY%=0¶
                  rcX%=0:rcY%=0¶
                  oldrcX%=0:oldrcY%=0¶
```

```
                CALL SetDrMd(SuperRast&,2)¶
                subox%=ox%¶
                suboy%=oy%¶
                loopflag%=0¶
                IF (cX% MOD grid1%)>(.5*grid1%) THEN
cX%=cX%+grid1%¶
                IF (cY% MOD grid2%)>(.5*grid2%) THEN
cY%=cY%+grid2%¶
                cX% = cX%-(cX% MOD grid1%)¶
                cY% = cY%-(cY% MOD grid2%)¶
                GOSUB oldpos¶
                ¶
                WHILE test%<0¶
                  test%=MOUSE(0)¶
                  oldx%=mx%¶
                  oldy%=my%¶
                  oldcX%=ccX%¶
                  oldcY%=ccY%¶
                  oldrcX%=rcX%¶
                  oldrcY%=rcY%¶
                  mx%=MOUSE(1)¶
                  my%=MOUSE(2)¶
                  IF (mx% MOD grid1%)>(.5*grid1%) THEN
mx%=mx%+grid1%¶
                  IF (my% MOD grid2%)>(.5*grid2%) THEN
my%=my%+grid2%¶
                  mx%=mx%-(mx% MOD grid1%)¶
                  my%=my%-(my% MOD grid2%)¶
                  IF mx%=oldx% AND my%=oldy% AND s.fl%=0 THEN¶
                    rep.flag%=1¶
                  ELSE¶
                    rep.flag%=0¶
                    s.fl%=0¶
                  END IF¶
                  IF rep.flag%=0 THEN¶
                    GOSUB oldpos¶
                  END IF¶
                  IF mx%<layMinX%+5 THEN GOSUB ScrollRight¶
                  IF mx%>layMaxX%-15 THEN GOSUB ScrollLeft¶
                  IF my%<layMinY%+5 THEN GOSUB ScrollDown¶
                  IF my%>layMaxY%-20 THEN GOSUB ScrollUp¶
                  GOSUB updateDisp¶
¶
                  ccX%=ABS(mx%-cX%)+ABS(ox%-subox%)¶
                  ccY%=ABS(my%-cY%)+ABS(oy%-suboy%)¶
                  rcY%=my%-cY%+(oy%-suboy%)¶
                  rcX%=mx%-cX%+(ox%-subox%)¶
                  IF rep.flag%=0 THEN¶
                    GOSUB newpos¶
                  END IF¶
¶
                WEND¶
                GOSUB newpos¶
                CALL SetDrMd(SuperRast&,1)¶
                IF draw%=6 THEN¶
                    x1%=cX%+subox%¶
                    y1%=cY%+suboy%¶
                    x2%=cX%+subox%+oldrcX%¶
                    y2%=cY%+suboy%+oldrcY%¶
                    IF x2%<x1% THEN SWAP x1%,x2%¶
                    IF y2%<y1% THEN SWAP y1%,y2%¶
                    x1%=x1%+1:y1%=y1%+1¶
```

```
                    x2%=x2%-1:y2%=y2%-1¶
                    CALL SetAPen(WINDOW(8),0)¶
                    CALL RectFill(WINDOW(8),x1%,y1%,x2%,y2%) ¶
                    CALL SetAPen(WINDOW(8),1)¶
                ELSEIF (draw%=3 AND (fetch%<>0 OR grid%<>0)) THEN¶
                    REM nothing¶
                ELSE¶
                    GOSUB newpos¶
                END IF¶
                ¶
            RETURN¶
¶
newpos:         IF draw%=1 THEN¶
                    CALL DrawEllipse(SuperRast&,cX%+subox%,
cY%+suboy%,ccX%,ccY%)¶
                ELSEIF draw%=2 THEN ¶
                    LINE (cX%+subox%,cY%+suboy%)-
(cX%+subox%+rcX%,cY%+suboy%+rcY%),swapper%¶
                ELSEIF draw%=3 OR draw%=6 THEN¶
                    LINE (cX%+subox%,cY%+suboy%)-
(cX%+subox%+rcX%,cY%+suboy%+rcY%),swapper%,b ¶
                END IF¶
                RETURN¶
                ¶
oldpos:         IF draw%=1 THEN¶
                     CALL
DrawEllipse(SuperRast&,cX%+subox%,cY%+suboy%,oldcX%,oldcY%)¶
                 ELSEIF draw%=2 THEN¶
                     LINE (cX%+subox%,cY%+suboy%)-
(cX%+subox%+oldrcX%,cY%+suboy%+oldrcY%),swapper%¶
                 ELSEIF draw%=3 OR draw%=6 THEN¶
                     LINE (cX%+subox%,cY%+suboy%)-
(cX%+subox%+oldrcX%,cY%+suboy%+oldrcY%),swapper%,b¶
                 END IF¶
                 RETURN¶
filler:        '* Fill Routine¶
               test%=MOUSE(0)¶
               oldx%=MOUSE(1)¶
               oldy%=MOUSE(2)¶
               PAINT (ox%+oldx%,oy%+oldy%),1,1¶
               RETURN¶
               ¶
freedraw:      '* Freehand Drawing¶
               test% = MOUSE(0)¶
               oldx%   = MOUSE(1)¶
               oldy%   = MOUSE(2)¶
               WHILE test%<0¶
                 oldx% = mx%¶
                 oldy% = my%¶
                 mx%   = MOUSE(1)¶
                 my%   = MOUSE(2)¶
                 IF mx%<layMinX%+10 THEN GOSUB ScrollRight¶
                 IF mx%>layMaxX%-20 THEN GOSUB ScrollLeft¶
                 IF my%<layMinY%+10 THEN GOSUB ScrollDown¶
                 IF my%>layMaxY%-25 THEN GOSUB ScrollUp¶
                 LINE (ox%+oldx%,oy%+oldy%)-
(ox%+mx%,oy%+my%),swapper%¶
                 GOSUB updateDisp¶
                 test% = MOUSE(0)¶
               WEND¶
               RETURN¶
¶
```

```
paintdraw:      '* Draw with Image¶
                 test%=MOUSE(0)¶
                 WHILE test%<0¶
                   mx%   = MOUSE(1)¶
                   my%   = MOUSE(2)¶
                   IF mx%<layMinX%+10 THEN GOSUB ScrollRight¶
                   IF mx%>layMaxX%-20 THEN GOSUB ScrollLeft¶
                   IF my%<layMinY%+10 THEN GOSUB ScrollDown¶
                   IF my%>layMaxY%-25 THEN GOSUB ScrollUp¶
                   mx%   = mx%-(mx% MOD grid1%)¶
                   my%   = my%-(my% MOD grid2%)¶
                   IF mx%<layMinX%+10 THEN GOSUB ScrollRight¶
                   IF mx%>layMaxX%-20 THEN GOSUB ScrollLeft¶
                   IF my%<layMinY%+10 THEN GOSUB ScrollDown¶
                   IF my%>layMaxY%-25 THEN GOSUB ScrollUp¶
                   ¶
                   test% = MOUSE(0)¶
                   PUT (mx%+ox%,my%+oy%),get.array%,OR¶
                 WEND¶
                 RETURN¶
                   ¶
ScrollHome:     x%=-ox%¶
                y%=-oy%¶
                ox%=0¶
                oy%=0¶
                GOSUB ScrollDisplay¶
                RETURN¶
                ¶
ScrollRight:    IF ox%>grid1%-1 THEN¶
                  x%  = -grid1%¶
                  ox% = ox%-grid1%¶
                  GOSUB ScrollDisplay¶
                END IF¶
                RETURN¶
                ¶
ScrollLeft:     IF ox%<(superWidth%-layMaxX%+layMinX%-grid1%)
THEN¶
                  x% = grid1%¶
                  IF textWidth%<>0 THEN¶
                    IF ox%+textWidth%<(superWidth%-
layMaxX%+layMin%) THEN¶
                      x% = textWidth%¶
                    END IF¶
                    textWidth% = 0¶
                  END IF  ¶
                  ox% = ox%+x%¶
                  GOSUB ScrollDisplay¶
                END IF¶
                RETURN¶
                ¶
ScrollUp:       IF oy%<(superHeight%-layMaxY%+layMinY%-grid2%)
THEN¶
                  y%  = grid2%¶
                  oy% = oy%+grid2%¶
                  GOSUB ScrollDisplay¶
                END IF¶
                RETURN¶
                ¶
ScrollDown:     IF oy%>grid2%-1 THEN¶
                  y%  = -grid2%¶
                  oy% = oy%-grid2%¶
                  GOSUB ScrollDisplay¶
```

```
                END IF¶
                RETURN¶
                ¶
ScrollDisplay: '* scroll it¶
                CALL ScrollLayer(scrLayerInfo&,
SuperLayer&,x%,y%)¶
                x% = 0¶
                y% = 0¶
                s.fl% = 1¶
                RETURN  ¶
                ¶
updateDisp:     IF kl%=0 THEN¶
                  actu$="> "+mode$+" [F]="+STR$(fontHeight%)+"
[X]="+STR$(ox%+mx%)+"  [Y]="+STR$(oy%+my%)+CHR$(0)¶
                  CALL WaitTOF¶
                  CALL SetWindowTitles(windAdd&,SADD(actu$),-1)¶
                END IF¶
                RETURN  ¶
                                ¶
loadFonts:     '* Load Disk Fonts¶
                sp$     = mode$¶
                mode$   = "LOAD FONTS"¶
                GOSUB updateDisp¶
                opt&    = 2^0+2^16¶
                bufLen& = 3000¶
                buffer& = AllocMem&(bufLen&,opt&)¶
                IF buffer&<>0 THEN¶
                  er& = AvailFonts&(buffer&,bufLen&,3)¶
                  IF er& = 0 THEN¶
                    fontcnt% = PEEKW(buffer&)¶
                    IF fontcnt%>19 THEN fontcnt% = 19¶
                    DIM textAttr&(fontcnt%*2)¶
                    DIM textName$(fontcnt%)¶
                    FOR loop%=0 TO fontcnt%-1¶
                      counter%    = loop%*10¶
                      fontTitle&  = PEEKL(buffer&+4+counter%)¶
                      fontH%      = PEEKW(buffer&+counter%+8)¶
                      textAttr&(loop%*2+1)=
PEEKL(buffer&+counter%+8)¶
                      fontTitle$  = ""¶
                      check%=PEEK(fontTitle&)¶
                      WHILE check%<>ASC(".")¶
                        fontTitle$ = fontTitle$+CHR$(check%)¶
                        fontTitle& = fontTitle&+1¶
                        check%     = PEEK(fontTitle&)¶
                      WEND¶
                      textName$(loop%) =
fontTitle$+".font"+CHR$(0)¶
                      fontName$        = fontTitle$+STR$(fontH%)¶
                      fontcounter      = fontcounter+1¶
                      MENU 3,fontcounter,1,fontName$¶
                    NEXT loop%¶
                    CALL FreeMem(buffer&,bufLen&)¶
                    fontflag% = 1¶
                  END IF¶
                ELSE¶
                  BEEP¶
                END IF¶
                mode$  = sp$¶
                RETURN   ¶
                         ¶
loadFont:      '* Load Font¶
```

```
                sp$     = mode$¶
                mode$   = "LOAD FONT"¶
                GOSUB updateDisp¶
                textBase%             = (menuItem-1)*2¶
                textAttr&(textBase%)  = SADD(textName$(menuItem-
1))¶
                newFont& = OpenDiskFont&(VARPTR(textAttr&
(textBase%)))¶
                IF newFont& = 0 THEN¶
                  newFont& = OpenFont&(VARPTR(textAttr&
(textBase%)))¶
                END IF¶
                IF newFont&<>0 THEN¶
                  IF oldFont&<>0 THEN¶
                    CALL CloseFont(oldFont&)¶
                  END IF¶
                  CALL SetFont(SuperRast&,newFont&)¶
                  oldFont&    = newFont&¶
                  fontHeight% = INT(textAttr&(textBase%+1)/2^16)¶
                ELSE¶
                  BEEP  ¶
                END IF¶
                mode$  = sp$¶
                RETURN¶
                ¶
drawtext:       '* Write Text in Graphic Bitmap¶
                IF (mx% MOD grid1%)>(.5*grid1%) THEN
mx%=mx%+grid1%¶
                IF (my% MOD grid2%)>(.5*grid2%) THEN
my%=my%+grid2%¶
                my%=my%-(my% MOD grid2%)¶
                mx%=mx%-(mx% MOD grid1%)¶
                CALL Move(SuperRast&,mx%+ox%,my%+oy%+
fontHeight%)¶
                mode$ = "ENTRY"+CHR$(0)¶
                CALL WaitTOF¶
                CALL SetWindowTitles(windAdd&,SADD(mode$),-1)¶
¶
                mode$  = "TEXT"¶
                in$    = ""¶
                WHILE in$<>CHR$(13)¶
                  IF in$<>"" THEN¶
                    CALL SetDrMd(SuperRast&,drMd%)¶
                    enable% = AskSoftStyle&(SuperRast&)¶
                    n&      = SetSoftStyle&(SuperRast&,style%,
enable%)¶
                    tempX%  = PEEKW(SuperRast&+36)¶
                    tempY%  = PEEKW(SuperRast&+38)¶
                    rand%   = tempX%-ox%¶
                    IF rand%>layMaxX%-20 THEN¶
                      textWidth% = PEEKW(SuperRast&+60)¶
                      GOSUB ScrollLeft¶
                    END IF  ¶
                    IF outline% = 0 THEN¶
                      CALL Text(SuperRast&,SADD(in$),1)¶
                    ELSE¶
                      CALL SetDrMd(SuperRast&,0)¶
                      FOR loop1%=-1 TO 1¶
                        FOR loop2%=-1 TO 1¶
                          CALL Move(SuperRast&,tempX%+loop2%,
tempY%+loop1%)¶
                          CALL Text(SuperRast&,SADD(in$),1)¶
```

```
                    NEXT loop2%¶
                  NEXT loop1%¶
                  CALL SetDrMd(SuperRast&,2)¶
                  CALL Move(SuperRast&,tempX%,tempY%)¶
                  CALL Text(SuperRast&,SADD(in$),1)¶
                  tempW% = 0¶
              END IF¶
            END IF¶
            in$ = INKEY$¶
           '* Function Key Assignments¶
           '* These lines could be used to assign characters¶
           '* which are normally unaccessable to the ¶
           '* function keys for use in TEXT mode¶
             IF in$ = CHR$(129) THEN in$ = CHR$(49)¶
             IF in$ = CHR$(130) THEN in$ = CHR$(50)¶
             IF in$ = CHR$(131) THEN in$ = CHR$(51)¶
             IF in$ = CHR$(132) THEN in$ = CHR$(52)¶
             IF in$ = CHR$(133) THEN in$ = CHR$(53)¶
             IF in$ = CHR$(134) THEN in$ = CHR$(54)¶
             IF in$ = CHR$(135) THEN in$ = CHR$(55)¶
             IF in$ = CHR$(136) THEN in$ = CHR$(56)¶
             IF in$ = CHR$(137) THEN in$ = CHR$(57)¶
             IF in$ = CHR$(138) THEN in$ = CHR$(48)¶
           WEND¶
           m$ = "TEXT"+CHR$(0)¶
           CALL WaitTOF¶
           CALL SetWindowTitles(windAdd&,SADD(m$),-1)¶
¶
           mx% = 0¶
           my% = 0¶
           RETURN    ¶
¶
newpointer:  '* Define Drawing pointer¶
           opt&=2^1+2^16¶
           ptr&=AllocMem&(20,opt&)¶
           IF ptr&<>0 THEN¶
             POKEW ptr&+4,256¶
             POKEW ptr&+8,640¶
             POKEW ptr&+12,256¶
             CALL SetPointer(WINDOW(7),ptr&,3,16,-8,-1)¶
           END IF¶
           RETURN¶
               ¶
hardcopy:    '* Print Bitmap¶
           sp$    = mode$¶
           mode$  = "HARDCOPY"¶
           GOSUB updateDisp¶
           dev$ = "printer.device"+CHR$(0)¶
           er&  = OpenDevice&(SADD(dev$),0,pio&,0)¶
           IF er&=0 THEN¶
             er& = DoIO&(pio&)¶
             IF er&<>0 THEN BEEP:BEEP¶
             CALL CloseDevice(pio&)¶
           ELSE¶
             BEEP¶
           END IF¶
           mode$  = sp$¶
           RETURN¶
           ¶
swapcol:     IF swapper%=0 THEN¶
             swapper%=1¶
           ELSE¶
```

```
                swapper%=0¶
              END IF¶
                ¶
              RETURN¶
¶
changePrint: '* Change Printer Parameters¶
             '* Output to your Window¶
              backup.font&=PEEKL(WINDOW(8)+52)¶
              CALL SetFont(WINDOW(8),font&) ¶
              POKEL SuperLayer&+12,backup.rast&¶
              POKEL WINDOW(8),backup.layer&¶
              e& = BehindLayer&(scrLayerInfo&,SuperLayer&)¶
              ¶
              CALL SetDrMd(WINDOW(8),1) ¶
              LINE (0,0)-(windWidth%-8-offset%-20,windHeight%-
15),1,bf¶
              LINE (20,10)-(windWidth%-8-offset%-40,windHeight%-
25),0,bf¶
              LOCATE 3,1¶
              PRINT TAB(4);"PRINT PARAMETER SETTINGS"¶
              PRINT TAB(4);"------------------------"¶
              PRINT¶
              PRINT TAB(4);"Outline area to be printed"¶
              PRINT TAB(4);"with the frame."¶
              PRINT¶
              FOR t=1 TO 10000:NEXT t¶
              repeat:¶
              fetch% = 1¶
              draw%  = 3¶
              mode$  = "FETCH"¶
¶
             '* Output Back to the Layer¶
              e&=UpFrontLayer&(scrLayerInfo&,SuperLayer&)¶
              POKEL SuperLayer&+12,WINDOW(8)¶
              POKEL WINDOW(8),SuperLayer&¶
              ¶
              GOTO mcp¶
              ¶
              continue:¶
              fetch%=0¶
              mode$="SQUARE"¶
             '* Output on your Window¶
              e& = BehindLayer&(scrLayerInfo&,SuperLayer&)¶
              POKEL SuperLayer&+12,backup.rast&¶
              POKEL WINDOW(8),backup.layer&¶
                LOCATE 9,1¶
                PRINT TAB(4);USING "New Starting
X:####";printX0%¶
                PRINT TAB(4);USING "New Starting
Y:####";printY0%¶
                PRINT TAB(4);USING "New Ending
X:####";printX1%¶
                printX1P%=printX1%-printX0%¶
                PRINT TAB(4);USING "New Ending
Y:####";printY1%¶
                printY1P%=printY1%-printY0%¶
              ¶
                LOCATE 15,1¶
                PRINT TAB(4) SPACE$(26)¶
                LOCATE 15,1¶
                PRINT TAB(4);¶
                INPUT "Are the values OK (y/n) ";yn$¶
```

```
    IF yn$="n" THEN GOTO repeat¶
¶
 PRINT TAB(4);¶
 INPUT "[1] Normal [2] Resized ";nv%¶
 IF nv%=2 THEN¶
   PRINT TAB(4);¶
   INPUT "Absolute X Expansion";printX%¶
   PRINT TAB(4);¶
   INPUT "Absolute Y Expansion";printY%¶
   printSpec% = 0¶
 ELSE¶
   printSpec% = 4¶
 END IF¶
¶
 POKEW pio&+44,printX0%¶
 POKEW pio&+46,printY0%¶
 POKEW pio&+48,printX1P%¶
 POKEW pio&+50,printY1P%¶
 POKEL pio&+52,printX%¶
 POKEL pio&+56,printY%¶
 POKEW pio&+60,printSpec%¶
 ¶
 '* Output Back to the Layer¶
 e&=UpFrontLayer&(scrLayerInfo&,SuperLayer&)¶
 POKEL SuperLayer&+12,WINDOW(8)¶
 POKEL WINDOW(8),SuperLayer&¶
 CALL SetFont(WINDOW(8),backup.font&)¶
 CALL SetDrMd(WINDOW(8),drMd%)¶
 ¶
 RETURN¶
```

8.1 Paint-1024 program instructions

Before you start the program, take a good look at the variable definitions at the beginning. You can use either lo-res or hi-res for your working screen. Since this has no effect on the size of the drawing, we recommend you use the lower resolution.

You can choose the size of the superbitmap which means that you will be choosing the overall size of the drawing. The current program settings define a 400x800 pixel drawing area. When you have enough memory, you can expand this to the full 1024x1024 CAD standard.

Starting the Program: You start Paint-1024 with "RUN". If you have two disk drives, put your program disk in one drive and the Workbench disk in the second drive. For use with single drives, start the program and then replace the program disk with the Workbench disk. If you receive the message "Please put disk XXX in drive..." you are returned to the Workbench screen. To get back to the program screen, press the left Amiga <A> key and <M> key simultaneously. Or you can simply slide the Workbench screen down.

First Drawing: Now you are in the drawing program. The drawing area is the biggest portion of the screen. The title bar is switched off as soon as the program begins. Press the right mouse button once and the menu bar appears. The SERVICE menu contains:

```
SERVICE          DRAW        FONT        I/O
Clear Screen
Coordinates On
--------------
Transparent
JAM 2
Complement
Inverse
--------------
normal/reset
italic
bold
underline
outline
--------------
s/w -> w/s
TitleBar
On/Off
QUIT
```

The first menu selection clears the entire drawing. The second selection switches on a coordinate grid. If you select this item a second time, the grid is removed.

The next nine items determine the method of text display, which we will discuss shortly. The `s/w` and `w/s` switch exchanges the back and foreground colors. This can be very helpful when you want to delete only part of your drawing.

Try selecting "`TitleBar on/off`" once. The actual mouse position will be displayed in the title bar along with the active drawing mode and the height of the current font. The title bar requires a lot of calculation time to keep it updated. Also, the scrolling and drawing functions will slow down when the title bar displays the mouse coordinates. So, when you can do without the title bar, leave it switched off.

When the program starts you will be in the FREEHAND drawing mode. As soon as you press the left mouse button, you will be drawing. When you get close to the right or lower border, the screen begins scrolling. Additional portions of the superbitmap become visible if you are not already at a border of the plane.

In the right screen border you will find five small icons. Move onto one of them with the mouse and press the left button. The screen scrolls in the direction indicated by the arrow if there is anything to move. The H symbol stands for Home and, with incredible speed, will return the drawing to its original position.

Circles, Lines, Squares

Under the menu item `DRAW` are the different drawing functions. As long as you hold down the left mouse button you can set the size and direction of the selected drawing function. Releasing the button draws the object.

Please remember that the `Circle` command only functions with Kickstart Version 1.2 or a later version.

Erase a portion of the drawing:

To delete only a portion of your drawing, select `Erase`. You can then select a rectangular portion of your drawing that will be replaced with the background color.

Text output:

Use this option to add text to your drawing. After you activate the `Text` mode you can still move around with the mouse. When you press the left mouse button the Amiga expects text to be entered through the keyboard. To end text entry, press the <Enter> key. The function keys are currently assigned the number 0 thru 9. They can be reassigned in the program to contain special characters of the Amiga fonts.

The SERVICE menu provides you with many text modes:

Transparent:	Text does not delete graphics.
JAM2:	Text overwrites graphics.
Complement:	What is black becomes white and vice versa.
Inverse:	Back and foreground colors are swapped (only functions in normal text mode).
Normal:	All text styles are switched off.
Italic:	Slanted text
Bold:	Bold text
Underline:	Text is underlined.
Outline:	Silhouette text

The text styles can be mixed by selecting more than one style. It is also possible to change styles while you are in text output mode.

Pressing <Enter> ends the text output mode.

If you want to enter more than one line of text and align the entire text, switch on the GRID. Select the X direction width of one character of the font and the Y direction height (more information about GRID follows). End each line with <Enter>. To start the next line, move the mouse to the desired position and press the left button.

Using different Fonts:

The FONT menu item allows you to use disk fonts for your text output. The first time you use this menu item you will see Load Fonts. Before you select this menu item you should insert the Workbench disk that contains your fonts into one of your drives. When Load Fonts is selected it will search for all the usable fonts (19 is the maximum number the menu will handle). The next time you click this menu item, it will contain a list of available fonts. You can select any font from this list. The Text item of the Draw menu uses the last selected font.

Graphic Print:

If you have a graphics-capable printer, you can print your drawing. The I/O menu contains the Print and Param options. To print the entire drawing plane, simply select Print. If you want to print only a portion of the drawing, select Param. This menu item allows you to select a section of your drawing by using a rectangular-rubberband method that frames the area you want printed with a rectangle. The selected area is displayed and you can make any needed corrections. Once the section is correct, you can choose between (1) normal print and (2) distorted print. Normal print produces the graphic in its normal proportions. Distorted print allows you to specify the number of pixels, in X and Y directions, that the printed output should use. If you specify more pixels in the horizontal direction than your printer is able to print, then nothing will be printed.

During printing all drawing functions are switched off.

Raster/Grid: Often, it is necessary to create symmetrical drawings or diagrams. The `Raster/Grid` menu item allows you to create a drawing grid of any size. Simply select the grid square size that you need. From this point on, all drawing operations will function according to this grid. You can change this grid size at any time. `Grid Reset` returns the grid to a lxl pixel size which is the default drawing mode.

Area fill: The `Fill` function lets you fill areas of any size. First the area to be filled must be framed completely with a black line that has no gaps. Place the mouse in the middle of the area to be filled and press the left mouse button. When filling in very large areas, and when working with large objects in the drawing area, this operation can take over a minute to complete.

User Defined Brushes: You can select a piece of your drawing to use as a brush. The size of the brush is limited only by your available memory. To do this, select `Get Area` from the `DRAW` menu. Now you can frame the desired area and "grab" it. You can draw with this area by using `Paint Area`.

User Defined Patterns: The way patterns work is quite similar to brush. You can define any portion of your graphic as a pattern. To do this, draw a small sample of the pattern and grab it by using `Get Area`. Now grab the same piece again using the `Raster/Grid` menu item and then select `Paint Area`. Now you can use your pattern as a brush to paint on the screen.

9. Graphic Programming in C

You have probably noticed that although BASIC can create fantastic graphics, it is quite slow. It is obvious that BASIC just doesn't have the speed we need to create fast graphics. However, if we use machine language, we could gain more speed, but machine language can be difficult to program.

The high level language "C" is perfect for this type of programming. This language offers the speed usually attained only with machine language and the simplicity of a programming language like BASIC.

Since we do not want to repeat the information provided in the first section of this book in this chapter, we will only discuss the C characteristics for graphic programming.

The programs in this section were written using three compilers, Aztec C V3.6a, Lattice C V5 and one using the Lattice C V3.10 compiler. Minor changes may have to be made to compile the programs on different compilers, see your compiler documentation for more information. Please remember to raise the system stack to 1024 bytes (CLI Command: 'stack 1024'). Now we will present the required elements for graphic programming in C.

9.1 The Amiga libraries

Since learning a new computer language can be difficult, we will concentrate only on graphic programming with C and not on learning the C language. If certains terms, such as `cast` and `struct` are confusing, please refer to the Abacus book **Amiga C for Beginners** or another book on Amiga C.

Before you can use C's graphic functions, which are not part of the standard C functions, first you have to learn about the Amiga libraries.

Each library contains a series of jump addresses for the individual graphic ROM routines. Since there are different ROM versions (Kickstarts) for the Amiga, the library concept is very useful. Every Kickstart has its own memory section that places a routine at a specific address. This means that a routine can be at a different address in each Kickstart version. These addresses are found in the libraries. This allows a program to work with Kickstart 1.2 and Kickstart 1.3 even though the routines are in different locations.

There are libraries for many different purposes. For us the most important library is the graphics library. You open the graphics library like this:

```
GfxBase = (struct GfxBase*)OpenLibrary("graphics.library",0)
```

`GfxBase` is the structure that contains all the important library information. The `OpenLibrary` function passes a pointer for this information to an initialized `GfxBase` structure that must be named `GfxBase`. At the same time, the string `"graphics library"` is the name of the graphics library. The zero in the parameter list for `OpenLibrary` specifies the current version of the library, which is the one we want to open.

Now that we have opened the graphics.library we can return to graphic programming. First we must create an area where we can display and save our graphics.

Let's compare our operations to those of a large factory.

9.2 The boss: the View

View should already be familiar to you from the first two sections of this book. However, since View is so important, we will briefly discuss it again.

View is the link between our creativity and the computer. The structure `struct View` contains all the necessary View data. In order to set up a View structure, you must use `struct View View`. Use `InitView(&View)` to initialize the View structure for all the individual variables.

9.3 The foreman: the ViewPort

This is where you determine the size of the creativity area, the display mode and the available colors. All of this data is sent to the ViewPort This data is later used to create the Copper list that the special Copper coprocessor will use to construct your screen.

Before you can send data to the ViewPort, you must first specify the position on the screen at which data should appear. The variables `View.DxOffset` and `View.Dyoffset` establish this position on the screen: One sets the X-coordinate on the screen, while the other sets the Y-coordinate. `InitView` initializes these variables so that your View position matches the positioning set in Preferences (i.e., the corner angle in your Preferences screen which helps center the screen).

If you enter a zero value for `ViewPort.DxOffset` and `ViewPort.DyOffset` the upper left corner of the ViewPort will be positioned at exactly the same location as the View. Different values (larger than zero) will place your ViewPort somewhere in the middle of the screen.

`ViewPort.DWidth` and `ViewPort.DHeight` set the size of the ViewPort. When setting these variables you must remember which resolution you selected. If you selected 640 pixels horizontal resolution (`ViewPort.Mode=HIRES`), you must specify a `ViewPort.DWidth` of 640 (instead of 320 normal mode). This also applies to `ViewPort.Mode=LACE` (Interlace-Mode), which must have a `ViewPort.DHeight` of 400 instead of 200 (PAL 512 instead of 256).

A color map (or palette) is required for the color resolution. Since the different color entries require memory, we must assign them.

```
ColorMap = (struct ColorMap*) GetColorMap
(Number_of_Colors)
```

sets up an entire structure called the `ColorMap` structure. This structure stores the pointers to the color memory and the number of available colors for the ViewPort.

After you initialize the color map you must make it available to the ViewPort. You can send a pointer for the color map to the ViewPort like this:

```
ViewPort.ColorMap = ColorMap
```

or make a direct call like this:

```
ViewPort.ColorMap = (struct ColorMap *) GetColorMap(Number_of_Co
lors)
```

It is important that you use the `Cast ('...(struct ColorMap*)')` to prevent warnings (pointers do not point to same object).

You can determine the returned values of the function `GetColorMap` (and all other functions) before the program sets them:

```
extern struct ColorMap *GetColorMap()

...

Main()
```

Now the compiler will not display the warning messages when you call `ViewPort.ColorMap = GetColorMap (Number_of_Colors).`

To prevent the ViewPort structure from containing any random values that would confuse the system, we also have an initialize command:

```
InitVPort(&ViewPort)
```

You should always use this command before writing any values to the ViewPort or using the ViewPort.

Before we can use of the bit-map we have to make a connection between the View and the ViewPort:

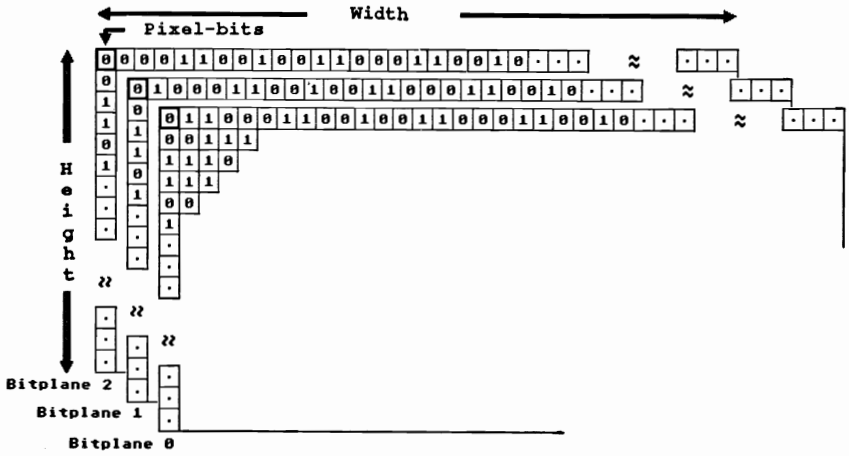
```
View.ViewPort = &ViewPort
```

9.4 The worker: the Bit-map

At some time and in some place, our graphic must be entered into memory. The bit-map controls both when and where this will occur. Then this information is divided among different bit-planes. The number of bit-planes sets how many colors are located in the Viewport and displayed by the bit-map. The more bit-planes there are, the more colors. The actual number of colors is calculated below:

Number_of_Colors = 2^Number_of_BitPlanes

This calculation is based on the way the planes are graphically layered on top of each other in the display (in memory they are actually stored one after another).



Each pixel is represented by one bit in every bit-plane. The combination of all set and unset bits of a pixel in all the bit-planes determines the number of the color register (see Chapter 14). The pixel is then displayed in that color.

The `AllocRaster` function (a subset of the `AllocMem` function) allocates memory (bytes) for the individual bit-planes. This function also makes sure that the assigned memory area begins at an even word address. This is necessary since the system can only access even word addresses.

You use `BitMap.Planes[i] = AllocRaster (Width, Height)` to assign the required memory for a bit-plane of the bit-map.

Width sets the width in pixels and height specifies the height in rows. You must make sure that the pointer `BitMap.Planes[i]` never equals to zero. If this happens, you will be unable to reserve enough memory. (The same thing can happen when you use a value of zero with the routines `OpenLibrary` or `GetColorMap.`) Most of the time this results in insufficient memory problems. When this happens, try to have your program be the only active one (other than the CLI or Workbench) in your Amiga. Test the return values of the functions and have them print an attention message if they receive a value of zero. This will let all users know which memory areas cannot be used. For example:

```
printf (" No more Memory available for ColorMap/n");
exit (0);
```

Now back to the number of bit-planes. The system provides us with eight `BitMap.Planes[1..8]` pointers. At the moment, (Kickstart 1.2, Amiga 2000) only a maximum of six of these can be used. Later, by expanding the system it may be possible to use all of them.

Six bit-planes can provide up to $2^6 = 64$ displayed colors. There are actually only 32 possible colors based on the pixel-bits (the Amiga 2000/1000/500 only has 32 color registers).

We only require that five bit-planes have 32 colors. The sixth bit-plane is used with HAM or Halfbrite modes (see Chapter 17).

There is still another limitation to discuss. You can only work with four bit-planes in hi-res mode or 640 pixel horizontal resolution. The reason for this is that we have more data to display on the screen. The four bit-planes use up the limited time the Amiga has available to display all this data.

Before we continue with the bit-maps we will show you how to initialize the bit-planes (or memory).

Since we cannot guarantee that the memory is clear, we must clear it ourselves (otherwise we may have a mess on the screen).

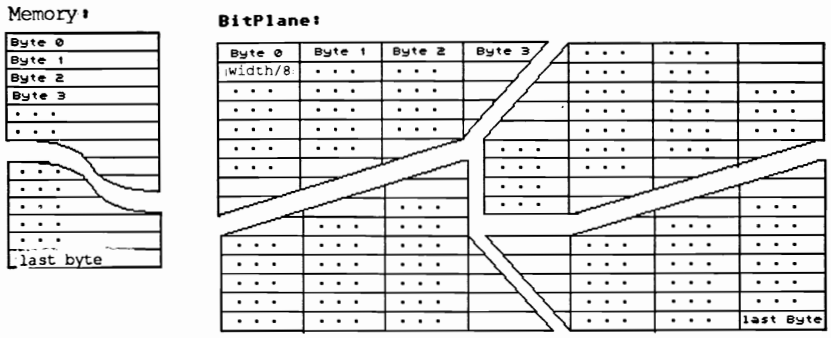
`BltClear (BitMap.Planes[i], Number_of_Bytes,Flags)` is used to clear memory. To calculate the number of bytes to clear use the following:

```
Number_of_Bytes = Width*Height/8
```

The macro `RASSIZE(Width,Height)` calculates the same value (For the parameter flags please see Appendix B. Normally flags are set equal to zero).

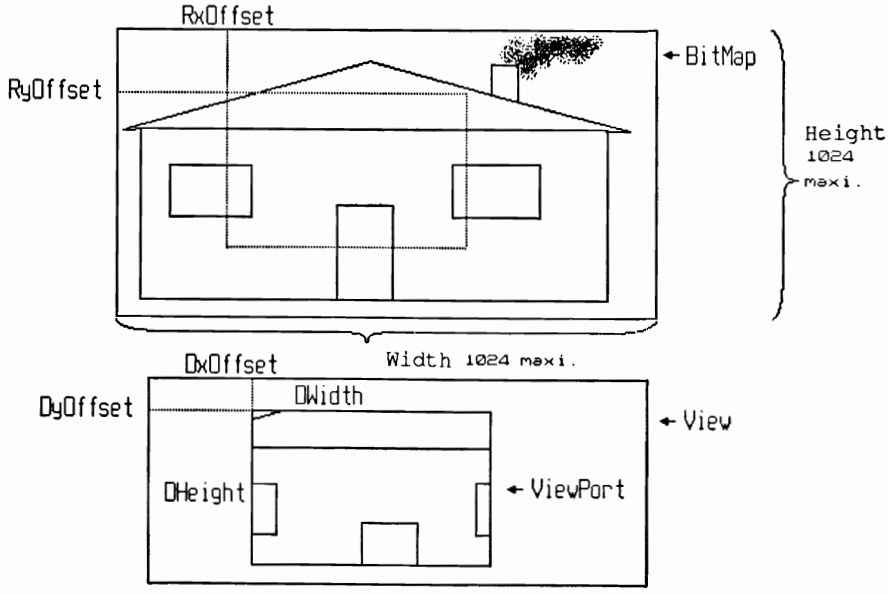
The memory area information that you receive from `AllocRaster` is distributed in an individual bit-plane row like this:

The bytes '0 to width/8-1' represent the memory for the first row of the bit-plane. The bytes 'width/8' to '2*Width/8-1' represent the second row, and this pattern continues to the last byte.



Now that we have closely studied the bit-planes, we will move to the bit-map (the bit-plane factory):

We set the size of the bit-map and the depth (number of bit-planes) in the bit-map structure. The size of the bit-map and of the ViewPort can be different:



The only thing you need to remember is that the bit-map size must be within the 1024 * 1024 row/pixel limit. You initialize the bit-map with `InitBitMap` (&BitMap, Depth, Width, Height).

9.5 The messenger: RasInfo

The bit-map and the ViewPort are still isolated from each other. We can connect them by using the `RasInfo` structure. This structure isn't established with an initialize procedure. You must program this structure, element by element, by writing the necessary values into it.

The actual connection between the ViewPort and bit-map to the `RasInfo` structure is made as follows:

```
ViewPort.RasInfo = &RasInfo
RasInfo.BitMap   = &BitMap
```

The bit-map pointer of the `RasInfo` structure (raster information) is the only common element. The only remaining variables to be initialized are the `RxOffSet` and `RyOffset`. These specify the pixel coordinates for the upper left corner of the ViewPorts and are normally initialized to zero.

The last `RasInfo` variable, the `RasInfo.Next` pointer, is used only for special display modes (see Chapter 17) and is normally set to zero.

9.6 The laborer: RastPort

We can compare the RastPort, which means the RastPort structure, to a heavy laborer. We process almost all of the graphic output through the RastPort because it contains the actual color of the pixels, the drawing mode (see Chapter 11) and much more (see Appendix A).

`InitRastPort(&RastPort)` initializes the RastPort. This initialization sets up default values that can be changed later by using the proper commands.

After initializing the RastPort, you need to supply the address of the bit-map where your graphic commands will become visible:

```
RastPort.BitMap = &BitMap
```

Now we have initialized all the required structures and linked them together (The RastPort does not require a link to the ViewPort because it only provides information for the graphic commands).

Until now, nothing has appeared on the screen. Finally, we can start creating something that will be shown on the screen.

The reason why there hasn't been anything displayed on the screen is because the Copper list created by special programs and used by the Copper coprocessor doesn't exist yet. The hardware registers are especially affected by the Copper list which is required to control the graphic display. See Appendix C, register `bltcon0/1`, `bplcon0/1/2`, etc.

The Copper list is created by using `MakeVPort (&View,&ViewPort)` and `MrgCop (&View)` which use the data initialized in View, ViewPort, etc.

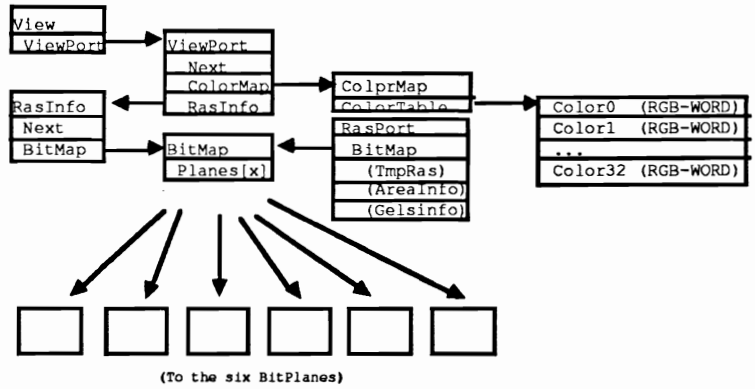
First we use `MakeVPort` to create an intermediate Copper list for every ViewPort (There can be more than one ViewPort in a View but you must make sure that a difference of at least one pixel row exists between the ViewPorts where they overlap). The Copper itself is unable to do anything with these intermediate lists. Once all the ViewPort Copper lists are created, we use `MgrCop` to make a final Copper list that the Copper is able to use. To make this list active, use `LoadView(&View)`.

Finally, the Copper works on the Copper list and you can see the results on the screen. Before activating the Copper list, you should save

the existing Copper list in a variable (`oldview = GfxBase->ActiView`). The `GfxBase` structure helps us do this because it always contains a pointer to the actual View.

If, just one time, you forget to save your old View by calling `LoadView(oldview)` before running your program, you will be lost in the Amiga jungle. Since we start most programs from the CLI or Workbench, you won't be able to return to the Workbench screen where the CLI window is also located.

The following illustration shows how View, ViewPort, RastPort, bitmap, RasInfo, etc., are linked together:



9.7 Finish the day

In our programs, "finish the day" means we still have more work to do. First we have to return the reserved memory back to the system. Then we have to close all the opened libraries using `CloseLibrary (base pointer)`. The base pointer for the graphic library must be `GfxBase`.

When you are returning memory, you must remember that the Copper list you created with `MakeVPort` and `MrgCop` uses memory also.

To release the ViewPort intermediate lists, use `FreeVPortCopLists (&ViewPort)`. Then use `FreeCprList (View LOFCprList)` and `FreeCprList (View SHFCprList)` to release the actual Copper list (see Appendix A for `View LOF/SHFCprList`).

`FreeRaster (BitMap Planes [i], Width, Height)` is used to release the memory for every bit-plane used.

To release the reserved memory for the color map, use `FreeColorMap (&ColorMap)` and `FreeColorMap (ViewPort.ColorMap)`.

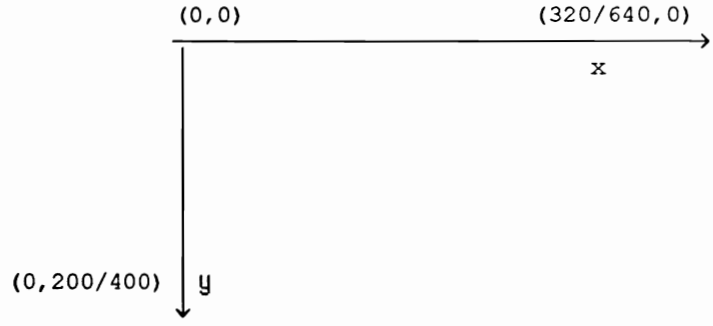
10. Lines and pixels

Now we can begin our discussion of our first graphic commands. The smallest element in all computer graphics is the pixel.

10.1 Pixels set with WritePixel

You can set a single pixel in the bit-map with the `WritePixel (&RastPort, x, y)` function. The RastPort, which is one of the required parameters, contains the color used to write the pixel to the bit-map.

The x and y parameters specify the two dimensional coordinates for the pixel. Please remember that the coordinates begin from the upper-left corner. The Y coordinate value increases as you move towards the bottom of the screen. The X coordinate increases as you move towards the right side of the screen:



The following program opens a View, ViewPort, etc. It uses `WritePixel` to create a string of pixels. We set each pixel in the string to a different color. After the drawing is complete, we rotate the contents of the color registers with the `ColorMap` function (see Chapter 14). This creates the familiar color cycling effect used by Deluxe Paint® and Graphicraft®.

```
/*******************************************************/
/*                     Strings.c                       */
/*                                                     */
/* This Program uses WritePixel() to create a String   */
/* that uses color cycling animation.                  */
/*                                                     */
/* Compiled with: Lattice V5                           */
/*                                                     */
/* (c) Bruno Jennrich                                  */
/*******************************************************/

#include "exec/types.h"
#include "exec/memory.h"
#include "exec/devices.h"
#include "graphics/gfx.h"
#include "graphics/gfxbase.h"
#include "graphics/gfxmacros.h"
#include "graphics/text.h"
#include "graphics/view.h"
#include "graphics/clip.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
#include "graphics/regions.h"
#include "hardware/blit.h"
#include "devices/keymap.h"

#define Width 320                              /* Width  */
#define Height 200                             /* Height */
#define Depth 4                                /* Depth  */
#define MODES 0                        /* Resolution Mode */

struct GfxBase *GfxBase;

struct View View;        /* Structures for our Display */
struct ViewPort ViewPort;
struct RasInfo RasInfo;
struct BitMap BitMap;
struct RastPort RastPort;
struct View *oldView;

int i,x,y,factor;

UWORD color = 0;                       /* Color counter */

USHORT Colors[16] = {
                      0x000,0x00f,0x0f0,0xf00,
                      0x0ff,0xff0,0xf0f,0xc34,
                      0x646,0x782,0xd23,0x5a9,
                      0x560,0xacf,0xedf,0xa09
                    };
                                /* Initial-Color Table */

char *LeftMouse = (char *)0xbfe001;

VOID Color();                  /* Forward declaration */
VOID Color_Cycle();
```

```
/*******************************************************/
/* Here we go  !                                       */
/*******************************************************/

main()
{

   if((GfxBase = (struct GfxBase *)
      OpenLibrary("graphics.library",0)) == NULL)
      {
         printf(" No Graphics !!!!!");
         Exit (0);
      }
                                 /*   Provide Display   */
   oldView = GfxBase->ActiView; /*   Save old View     */

   InitView (&View);             /*   Initialize View   */
   InitVPort (&ViewPort);        /* Initialize ViewPort */

   View.ViewPort = &ViewPort;
                              /* Link View and ViewPort */
   View.Modes = MODES;
                                       /* Set ViewMode */

   ViewPort.DWidth = Width; /* Set ViewPort-Parameters */
   ViewPort.DHeight = Height;
   ViewPort.RasInfo = &RasInfo;
   ViewPort.Modes = MODES;
   ViewPort.ColorMap = (struct ColorMap *)GetColorMap
(16);
   if (ViewPort.ColorMap == 0) goto cleanup1;
                                       /* Set ColorMap */

   RasInfo.Next = NULL;         /* Link ViewPort-BitMap */
   RasInfo.RxOffset = 0;        /*     thru RasInfo     */
   RasInfo.RyOffset = 0;
   RasInfo.BitMap = &BitMap;

   InitBitMap (&BitMap, Depth, Width, Height);
                                  /* initialize BitMap */
   for (i=0; i<Depth; i++)
      {
         BitMap.Planes[i] = (PLANEPTR)
               AllocRaster (Width,Height);
                                    /* Allocate Memory */
         if (BitMap.Planes[i] == 0)
            {
               printf ("No Memory for BitPlanes\n");
               goto cleanup1;
            }
         else
         BltClear (BitMap.Planes[i],
                  RASSIZE (Width,Height),0);
                                    /* Erase BitPlanes */
      }
```

```
InitRastPort (&RastPort);    /* Initialize RastPort */

RastPort.BitMap = &BitMap;
                             /* Link RastPort-BitMap */


LoadRGB4(&ViewPort,&Colors[0],16);
                    /* Write colors to ViewPort   */
                    /* ColorMap. Follow this with */
                    /* MakeVPort(), MrgCop() and  */
                    /*          LoadView()        */
                    /* (Intuition:RemakeDisplay())*/

MakeVPort (&View, &ViewPort);
MrgCop (&View);
LoadView (&View);

color = 1;                       /* Color Animation */

x = 0;
y = 0;
factor = 1;

while (y < Height-1)
   {
      SetAPen (&RastPort,color);
      WritePixel (&RastPort,x,y);

      x += (factor);
      if ((x >= Width-1) || (x == 0))
         {
            for (i=0; i<7; i++)
               {
                  Color();
                  SetAPen (&RastPort,color);
                  WritePixel (&RastPort,x,y+i);
               }
            factor *= -1;
            y += 6;
         }
      Color();
   }


while ((*LeftMouse & 0x40) == 0x40)
   {
      Color_Cycle();
   }

         FreeColorMap (ViewPort.ColorMap);/* Set   */
         FreeVPortCopLists (&ViewPort);   /* all   */
         FreeCprList (View.LOFCprList);   /* free  */
         FreeCprList (View.SHFCprList);

cleanup1: for (i=0; i<Depth; i++)
            {
```

```
                    if (BitMap.Planes[i] != NULL)
                    FreeRaster (BitMap.Planes[i],
                                Width,Height);
                }

            LoadView (oldView);
            CloseLibrary(GfxBase);
            return (0);
}

/*******************************************************/
/* This Routine takes care of incrementing the Color-  */
/* counter while drawing.                              */
/*-----------------------------------------------------*/
/* Entry-Parameters: None                              */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/

VOID Color()
{
   color++;
   color &= 15;
   if (color == 0) color=1;
}

/*******************************************************/
/* This Routine takes care of rotating the colors of   */
/* the ViewPort ColorMap                               */
/*-----------------------------------------------------*/
/* Entry-Parameters: None                              */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/

VOID Color_Cycle()
{
   UWORD i,help;
   static UWORD Cols[16];

   Cols[0] = Colors[0];    /* Background-Color remains */

   help = GetRGB4 (ViewPort.ColorMap,15);
   for (i=2; i<16; i++)
         Cols[i] = GetRGB4(ViewPort.ColorMap,i-1);
   Cols[1] = help;
                                        /* Shuffle      */
   LoadRGB4(&ViewPort,&Cols[0],16);
   MakeVPort (&View,&ViewPort);         /* Important !! */
   MrgCop (&View);
   LoadView (&View);
}
```

10.2 Drawing lines with Move and Draw

Naturally, the Amiga can also draw lines. What is remarkable about this capability is that the 68000 CPU is hardly used for this function. Instead, the Blitter (one of the Amiga's special coprocessors) executes this function independently.

First we have to set the starting coordinates. Drawing lines in C language differs from drawing lines in BASIC because C requires two functions instead of only one.

The `Move` function sets the actual graphic position (graphic cursor) to the specified coordinates:

```
Move (&RastPort,x,y)
```

At the same time, these coordinates are stored in the referenced RastPort (`RastPort.cp_x and RastPort.cp_y`).

The `Draw (&RastPort,x,y)` function draws a line to the specified coordinates. These coordinates automatically become the new graphic cursors position. The next `Draw` function (without any moves in between) draws from the last pixel that was drawn to the pixel coordinates specified by the new `Draw` function.

The following program demonstrates this powerful Amiga capability:

```
/*******************************************************/
/*                      Quix.c                         */
/*                                                     */
/* This Program demonstrates the hidden speed of the   */
/* AMIGA, especially when drawing lines.               */
/*                                                     */
/* Compiled with: Lattice V5                           */
/*                                                     */
/* (c) Buno Jennrich                                   */
/*******************************************************/

#include "exec/types.h"
#include "graphics/gfx.h"
#include "graphics/rastport.h"
#include "graphics/copper.h"
#include "graphics/view.h"
#include "graphics/gels.h"
#include "graphics/regions.h"
#include "graphics/clip.h"
#include "graphics/text.h"
#include "graphics/gfxbase.h"
```

```
#include "hardware/blit.h"

#define DEPTH 2                          /* 2 BitPlanes */
#define WIDTH 320                    /* 320 x 200 Pixels */
#define HEIGHT 200

#define MAX_X WIDTH - 2/* Do not exceed Borders        */
#define MAX_Y HEIGHT - 2 /* Left, Right, Bottom        */
#define MIN_X 1
#define MIN_Y 10          /* Do not write over Text    */

#define NOT_ENOUGH_MEMORY -1000
#define MAX_LINES 30              /* Maximum 30 Lines  */

struct View View;                /* Our own Structures */
struct ViewPort ViewPort;
struct RasInfo RasInfo; struct ColorMap *ColorMap;
struct BitMap BitMap;
struct RastPort RastPort;

SHORT i,j,
      Length;


struct GfxBase *GfxBase;
struct View *oldview;     /* Here we save the old View */

USHORT colortable[] = {0x000,0xf00,0x00f,0x0f0};
                              /* Our own Color Palette */

char *QuixString = "Quix - Lines *** (C) BHJ";
char *LeftMouse;                 /* For CIA - Address  */

VOID draw();                    /* Forward declaration */
VOID FreeMemory();

/*******************************************************/
/* Here we go  !                                       */
/*******************************************************/

main()
{

   if ((GfxBase = (struct GfxBase *)
      OpenLibrary("graphics.library",0)) == NULL)
      Exit (1);

   oldview = GfxBase->ActiView;        /* Save old View */

   InitView(&View);       /* Initialize View & ViewPort */
   InitVPort(&ViewPort);
   View.ViewPort=&ViewPort;       /* Link them together */

   InitBitMap(&BitMap,DEPTH,WIDTH,HEIGHT);
                                  /* initialize BitMap */
   InitRastPort(&RastPort);          /* RastPort       */
```

```
    RastPort.BitMap = &BitMap;      /* RastPort -> BitMap*/

    RasInfo.BitMap = &BitMap;      /* RasInfo for BitMap */
    RasInfo.RxOffset = 0;
    RasInfo.RyOffset = 0;
    RasInfo.Next = NULL;

    ViewPort.RasInfo = &RasInfo; /* ViewPort -> RasInfo */

    ViewPort.DWidth = WIDTH;    /* How big is ViewPort ? */
    ViewPort.DHeight = HEIGHT;

    ViewPort.ColorMap = (struct ColorMap *)GetColorMap(4);
                                  /* ViewPort's ColorMap */
    LoadRGB4 (&ViewPort,&colortable[0],4);
                           /* Load ColorMap with Colors */
    for (i=0; i<DEPTH; i++)
       {
          if ((BitMap.Planes[i] = (PLANEPTR)
             AllocRaster(WIDTH,HEIGHT)) == NULL)
                Exit(NOT_ENOUGH_MEMORY);
                      /* Reserve Memory for BitPlanes */

          BltClear ((UBYTE *)BitMap.Planes[i],
             RASSIZE(WIDTH,HEIGHT),0);
                               /* Erase BitPlane Memory */
       }

    MakeVPort(&View,&ViewPort);
                             /* CopperList for ViewPort */
    MrgCop(&View);
                         /* "Size" CopperList: for View */

    LoadView(&View);            /* Switch new Display on */

    LeftMouse = (char *)0xbfe001;
                           /* CIA Address for I/O Ports */
                           /* are used for Left         */
                           /* Mouse Button              */

    draw();                                /* Your Routine */

    LoadView(oldview);          /* WorkBench - Display on */
    FreeMemory();                    /* Return Everything */
    return(0);
}

/*******************************************************/
/* This Function takes care of drawing the lines and   */
/* reading the left Mouse Button                       */
/*-----------------------------------------------------*/
/* Entry-Parameters: None                              */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/
```

```
VOID draw()
{
   static SHORT i,
          new = 0,
          old = 0,
          full = FALSE,
          min, temp,
          delay = 1999,
          direction = 1,
          k = 0;

   struct {                /* Coordinates for one Line */
            int x1[MAX_LINES],
                x2[MAX_LINES],
                y1[MAX_LINES],
                y2[MAX_LINES];
          } line;

   static int veloc[4] = {-4,5,3,-7};
                 /* What is added to the Coordinate ? */

   static int start[4] = {110,25,160,74};
                  /* Where is the first Line drawn ? */

   static int max[4] = {MAX_X,MAX_Y,MAX_X,MAX_Y};
              /* Allowed maximum value of Coordinates */

   SetDrMd(&RastPort,JAM1);
   SetAPen(&RastPort,3);      /* Drawing color = Green */

   Length = WIDTH/2-TextLength (&RastPort,
             QuixString,strlen(QuixString))/2;
                       /* Calculate 'Central position' */

   Move(&RastPort,Length,RastPort.TxBaseline);
                            /* Position Graphic-Cursor */

   Text(&RastPort,QuixString,strlen(QuixString));
                                        /* display Text */
   SetAPen(&RastPort,1);        /* Drawing color = Red */

   Move(&RastPort,0,9);                  /* draw Frame */
   Draw(&RastPort,WIDTH-1,9);
   Draw(&RastPort,WIDTH-1,HEIGHT-1);
   Draw(&RastPort,0,HEIGHT-1);
   Draw(&RastPort,0,9);


   while ((*LeftMouse & 0x40) == 0x40)/* Mouse button? */
      {
         for (i=0;i<4;i++)/*Calculate new Line Coords  */
            {
               temp = start[i] + veloc[i];
                              /* Exit value + "Speed" */

               if (temp >= max[i])
```

```
                                        /* exceeds upper Limit ? */
                    {
                       temp = max[i]*2 - start[i] -
veloc[i];
                       veloc[i] = -veloc[i];
                    }

                 if (i==1 | i==3) min = MIN_Y;
                                  /* Check Y-Coordinates */

                 else min = MIN_X;

                 if (temp < min)
                    {
                              /* exceeds lower Border ? */
                       if (temp < 0) temp = -temp;
                       else temp = min;
                       veloc[i] = -veloc[i];
                    }
                 start[i] = temp;
              }

              if (full == TRUE)
                 {
                                     /* Erase last Line */
                    SetAPen(&RastPort,0);
                    Move(&RastPort,line.x1[old],
                                    line.y1[old]);
                    Draw(&RastPort,line.x2[old],
                    line.y2[old]);

                    old++;
                    old %= MAX_LINES;
                 }

              line.x1[new] = start[0];   /* set new Line */
              line.y1[new] = start[1];   /* coordinates  */
              line.x2[new] = start[2];
              line.y2[new] = start[3];

              if (new == MAX_LINES-1) full = TRUE;
                            /* MAX_LINES Lines drawn ? */

              SetAPen(&RastPort,2);/*Drawing color =Blue */

              Move(&RastPort,line.x1[new],line.y1[new]);
              Draw(&RastPort,line.x2[new],line.y2[new]);
                                        /* Draw new Line */
              new++;
              new %= MAX_LINES;

              if (delay != 0)
                 {
                    for (i=0; i<delay; i ++);
                                          /* Wait a bit */
                    delay += direction;
```

```
                    if (delay == 2000) direction = -1;
                }
            else
                {
                    k++;
                 if (k == 1000)/*1000 times 'full power'*/
                      {
                         direction = 1;
                         delay += direction;
                         k = 0;
                      }
                }
        }
}

/*******************************************************/
/* This Function returns allocated Memory for BitMaps  */
/* Copper-Lists, etc.                                  */
/*-----------------------------------------------------*/
/* Entry-Parameters: None                              */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/

VOID FreeMemory()
{
   for (i=0;i<DEPTH;i++)
      FreeRaster(BitMap.Planes[i],WIDTH,HEIGHT);
                              /* return BitPlane Memory */
   FreeColorMap(ViewPort.ColorMap);
                             /* Release ColorMap Memory */
                             /* that was reserved with  */
                             /* ColorMap()              */
   FreeVPortCopLists(&ViewPort);
                                /* Free Memory for      */
                                /* ViewPort CopperList  */
                                /* from MakeVPort()     */
   FreeCprList(View.LOFCprList);
   FreeCprList(View.SHFCprList);
                                      /* View CopperList */
   CloseLibrary(GfxBase);
                                        /* Close Library */
}
```

We have two additional tips for you. It is possible to exit most of our programs by using a mouse button. This is achieved by constantly checking the hardware registers. You would probably find this quite useful in your own programs.

If you don't like the line pattern we used, you can create a new pattern with `SetDrPt (&RastPort, Pattern)`. The 16 bit word pattern contains the actual pixel pattern for the line. (We set `Pattern = 0xffff` which draws a full line).

11. Color: drawing pens

After the two previous programs were up and running, you probably wondered how we calculated the colors for a pixel or line.

To explain this we must look at the bit-planes. Either we or the individual bits of the bit-planes determine a pixel's color.

For example, when the pixel bits of bit-planes one and three are set but bit-plane two is unset, we use the following formula: $(1*2^0 + 0*2^1 + 1*2^2) = 5$. This tells us that the color comes from color register five. You could also use normal binary arithmetic.

To set a specific color for a pixel, the opposite action is performed. You specify the color register to use and the graphic functions set the individual pixel bits for you.

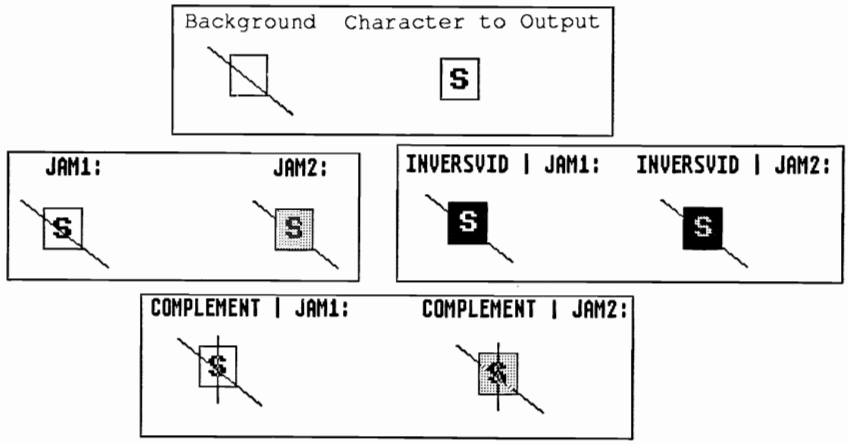
Remember that the Amiga has two color pens, the APen and the BPen. But before we continue with their functions, we will quickly discuss `Drawmodes`.

11.1 The Drawmodes

The Drawmodes determine the relationship between the existing pixel bits and the pixel bits written to their locations in the bit-planes. The available bit manipulation operations are OR, AND, NOT, and EXOR.

In the normal mode (SetDrMd (&RastPort, JAM1)) the written bits are simply OR'd with the existing bit-plane bits.

In JAM2 mode (SetDrMd (&RastPort, JAM2)) the written bits are added using AND. This has the same effect as JAM1 when using pixels and lines. However, when using text you can see the difference immediately. A drawings bit pattern is stored in memory as set and unset pixels. The Blitter (Block Image Transferrer), which is responsible for drawing output, also copies the unset bits of a drawing to the bit-plane. When JAM2 mode is on, the background is overwritten with the unset bits:



The COMPLEMENT mode performs an EXOR operation between the existing bits and the bits being written. The EXOR logic table looks like this:

Bitplane	Operation: ^	Pixel		Result
0		0	=	0
0		1	=	1
1		0	=	1
1		1	=	0

With the last mode, INVERSVID, the written bits are reversed using the NOT operator. A set bit becomes unset and an unset bit is set.

When using the NOT and EXOR operations you should remember that they take place internally. You only see the results when you use INVERSVID or COMPLEMENT together with JAM1 or JAM2 (for example: SetDrMd (&RastPort, INVERSVID | JAM2)). The bit pattern is written normally, as with JAM and JAM2, using OR or AND. The result is usually a chaotic colored line or text with colored spots.

11.2 The foreground pen

Now that we have explained the drawing modes, we can continue with the drawing pens. To set the pixel color in JAM1 mode, use SetAPen (&RastPort, Number_of_Color_Register). When working with pixels and lines, JAM1 and JAM2 mode have the same effect.

11.3 The background pen

When outputting text you can use JAM2 mode and the BPen (Background Pen) to display highlighted text. To do this, simply select a color for the unset pixels on the screen using:

```
SetBPen (&RastPort, Number_of_Color_Register).
```

You should be aware that the functions of APen and BPen affect the bit patterns in the bit-planes. They determine the bit by bit pattern that is always used when you write data to a specific location.

12. Intuition and graphics

Now that you know how to draw pixels and lines, change drawing modes and select drawing pens, we will show you a rather easy method to set up View, ViewPort, etc.

This method involves the user interface Intuition, which enables you to easily access the RastPorts.

The first and most important step is opening the Intuition.library. You do this in the same way as you opened the graphic.library. But instead of using the string name "graphics.library", you use "intuition.library". The function looks like this:

```
(IntuitionBase = (struct IntuitionBase *) OpenLibrary
("intuition.library",0)).
```

12.1 The individual screen

Intuition creates screens that are completely initialized ViewPorts with a ColorMap, a BitMap, a RastPort, etc.

Intuition also creates a View for us. However the most important feature of Intuition is that it takes control of almost everything so we don't have to do the work ourselves.

Basically, opening screens is reduced to two actions:

1. Initializing a `NewScreen` structure and
2. the `OpenScreen` call.

For details on the `NewScreen` structure please check the Appendices. We will only briefly explain the `Screen = (struct Screen *) OpenScreen (&NewScreen)`. This call creates a displayed, fully functional screen, with a screen structure that we can access (Note: `Screen` and `NewScreen` are not the same!). `&Screen->RastPort` is used with most of the graphic commands to access the RastPort.

The best example of an Intuition screen is the Workbench screen where all of the Workbench activities are performed.

Remember that `OpenScreen` uses up a lot of memory. If the pointer `Screen`, for the initialized screen structure is equal to zero, then there isn't any memory available. When this happens, it is best to have the program display a short error message and exit to the CLI or Workbench.

12.2 The window

Most of the time, windows are used for graphic output. One advantage of windows is that a drawn line cannot pass over the window border because it will be cut off before this can happen. This is not true for screens and your own ViewPorts. Under certain conditions, the excess portion of the line will be written to memory, which causes loss of data.

Opening windows requires two actions:

1. Initializing a `NewWindow` structure and
2. Calling `Window = (struct Window *) OpenWindow (&NewWindow).`

You can access the RastPort of a window by using `&Window->RastPort.`

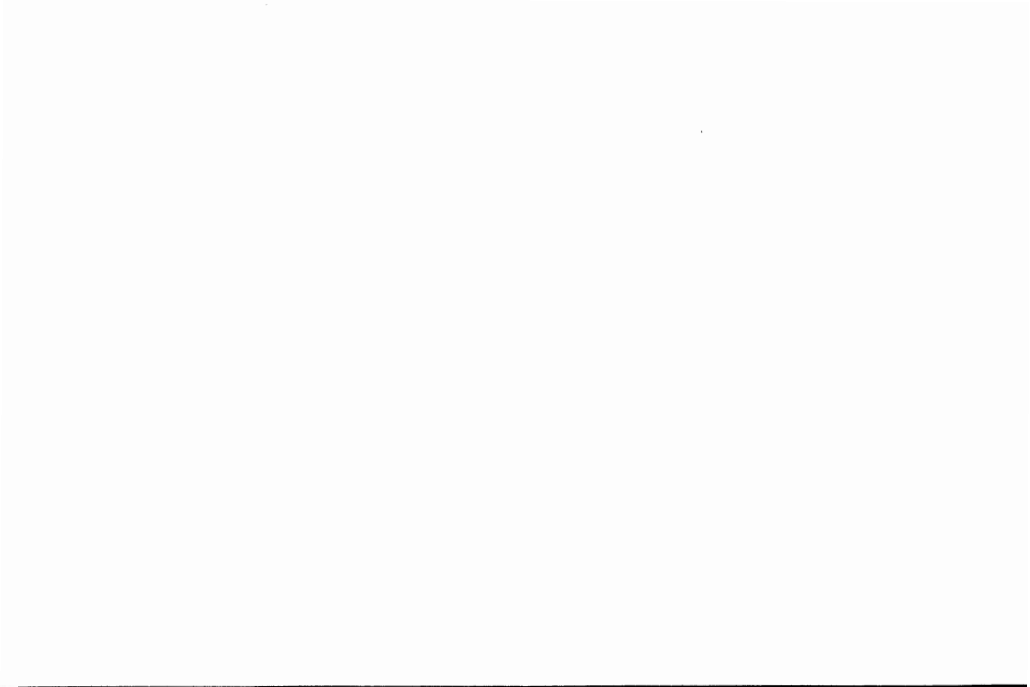
12.3 Exiting Intuition

We must return the memory used by Intuition for the same reason we had to return bit-plane memory. You must also release the memory that was used by the screens and windows.

Close a screen using `CloseScreen (Screen).` The window is closed in the same way with `CloseWindow (Window).` It is very important that you close the windows before closing the screens otherwise you will receive Guru Meditations.

Obviously, you must also close the graphic.library and Intuition.library:

```
CloseLibrary (IntuitionBase)
```



13. Filling areas in C

Now that we know how Intuition can help us, we can begin working on "areas".

13.1 A Flood function

We can fill connected areas using the `Flood` function. For a pictorial image of how `Flood` works, compare `Flood` to pouring paint into a bathtub. If you do not pour more than the tub can hold, the paint will not flow past the edge. However, the effect would be completely different if the tub had a crack in it causing the paint to spill out of the tub.

We determine the edges for a `Flood` function by drawing a frame around the area we want filled. This frame does not have to be square or have a specific shape. To perform the fill you must specify coordinates that are inside the area to be filled:

```
Flood (&RastPort,Mode,x,y)
```

With this type of fill, you must use a mode parameter equal to zero. This means that you fill all pixels, using the current drawing color in the current drawing mode, with the current fill pattern until you reach a border.

As explained above, you set the border by drawing a completely closed frame in a specific color. To set the border color, use `SetOPen (&RastPort, Color_of_Frame)`. (OPen = AOLPen = Area Out Line Pen = the color of the surface.)

When the mode parameter is equal to one, a slightly different fill method is being used. The color of the frame is no longer important. You fill only the connected areas that have the same color as the pixel at the selected X/Y coordinate.

You should be careful when using the `Flood` function because it uses up a large amount of memory.

Due to a recursive algorithm you have to prepare a temporary raster (a temporarily used bit-plane). This bit-plane must be at least as large as

the object you want to fill. It is much safer to create a complete additional bit-plane (`AllocRaster`).

Then you can make this memory available to a `TmpRas` structure and pass this information to the RastPort.

```
InitTmpRas (&TmpRas,&Memoryaddress,MemorySize)
RastPort.TmpRas = &TmpRas
```

or

```
RastPort.TmpRas = InitTmpRas (&TmpRas,&Memoryaddress,MemorySize)
```

Remember that you have to return the memory for the additional bit-plane when you are finished.

13.2 Filling rectangles?

Besides filling connected areas, the Amiga can also fill simpler areas. This is an important reason for the Blitter.

You can fill rectangles with the `RectFill (&RastPort, x1, y1, x2, y2)` command. The x1,y1 coordinates set the upper left corner and the x2/y2 coordinates set the lower right corner for the filled rectangle. Please make sure that the upper left corner coordinates are really to the left and above the coordinates for the lower right corner. You will get a Guru and a system lockup if your coordinates are not correctly specified.

This command automatically draws the frame for the filled rectangle. You set the color for the frame with `OPen (&RastPort,Color_of_Frame)`.

If you want to fill the rectangle without a frame, you use the macro `BNDRYOFF (&RastPort)` which stands for Boundary Off. This macro simply clears the AREAOUTLINE bit in the flag variable of the RastPort. To fill with the visible frame again, restore the AREAOUTLINE bit, use `RastPort.Flags |=AREAOUTLINE`. Changing the color of the `OPen` (with `SetOPen()`) automatically sets this bit.

The following program uses this method of area filling without requiring a `TmpRas` (additional bit-plane):

```
/*******************************************************/
/*                      Hanoi.c                        */
/* This Program shows you firsthand the Tower of Hanoi */
/* and the use of the RectFill() Command.              */
/*                                                     */
/* Compiled with: Lattice V5                           */
/*                                                     */
/* (c) Bruno Jennrich                                  */
/*******************************************************/

#include "exec/types.h"
#include "exec/devices.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
#include "graphics/gfx.h"
#include "graphics/gfxbase.h"
#include "graphics/gfxmacros.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
```

```
#include "graphics/regions.h"
#include "devices/keymap.h"
#include "hardware/blit.h"

#define GFX (struct GfxBase *)
#define INT (struct IntuitionBase *)
#define SCR (struct Screen *)

#define MAXHEIGHT 30                /* Maximum 30 Disks */


#define WIDTH 640                              /* Width  */
#define HEIGHT 200                             /* Height */


#define RPort &Screen->RastPort         /* Our RastPort */

struct NewScreen NewScreen =
                  {
                     0,0,WIDTH,HEIGHT,2,
                     1,0,HIRES,CUSTOMSCREEN,
                     NULL,NULL,NULL
                  };

char *String = "Tower of Hanoi with RectFill()";

struct Screen *Screen;
struct GfxBase *GfxBase;
struct IntuitionBase *IntuitionBase;

UWORD pattern[4] = {0x9248,0x2492,0x4924,0x9248};
long pos[3];                          /* For Pin Position */

long pole[3][MAXHEIGHT],             /* For which Pin, at */
                                     /* which Position,   */
                                     /* which Disk ?      */

     height[3] = {0,0,0};             /* How many Disks   */
                                     /* on the Pins ?     */
long width;
long disks;  /* How many disks on the starting Tower ? */
long high,offset;


char *LeftMouse = (char *)0xbfe001;

VOID turm();                       /* Forward declaration */
VOID init();
VOID build_up();
VOID draw();

/*******************************************************/
/* Here we go  !                                       */
/*******************************************************/

main(argc,argv)                          /* Get Argument */
```

```
int argc;
char *argv[];
{
   long i,j,
        Length;

      if (!(argc==2))
         {
            printf (" Tower Height: ");
            scanf ("%ld",&disks);
         }
      else
         sscanf(argv[1],"%ld",&disks);

      if (disks >= MAXHEIGHT)
         {
            printf ("Too many Disks !!!\n");
            exit (1);
         }


               /* Calculate Pin (X-) Position for Display
*/
     pos[0] = WIDTH/6+5;                          /* Left   */
     pos[1] = WIDTH/2;                            /* Middle*/
     pos[2] = (WIDTH/6)*5-5;                      /* Right  */

     width = (WIDTH/6)/disks - 2;
                                    /* width-difference */

     init();

     SetRGB4(&Screen->ViewPort,0,0,0,0); /* Set Colors */
     SetRGB4(&Screen->ViewPort,1,15,15,0);
     SetRGB4(&Screen->ViewPort,2,0,15,15);
     SetRGB4(&Screen->ViewPort,3,15,0,15);

                                     /* initialize Pins */
     for (i=0; i<MAXHEIGHT; i++)
         for (j=0; j<3; j++)
            pole[j][i]=0;                  /* Erase Pin */

     for (i=0; i<disks; i++)
            pole[0][i]=disks-i;
                              /* smallest disk to top */

     height[0] = disks;             /* 1 Pin has N-Disks */
     SetAfPt(RPort,pattern,2);      /* Set Fill pattern  */
     SetOPen(RPort,2);              /* Framing pen       */

     high = (HEIGHT*3/8)/disks;
                             /* How high is one Disk ? */
     offset = HEIGHT*3/8;

     build_up();                 /* Build first Display */
```

```
        SetAPen (RPort,3);

                                              /* String output */
        Length=TextLength(RPort,String,strlen(String))/2;
        Move (RPort,WIDTH/2-Length,
                    Screen->RastPort.TxBaseline);
        Text (RPort,String,strlen(String));

                                             /* Start Recursions */
        turm (disks,1,2,3);

        while ((*LeftMouse & 0x40) == 0x40);
                                            /* Wait a little bit */
        CloseScreen(Screen);
        CloseLibrary(GfxBase);
        CloseLibrary(IntuitionBase);
        return (0);
}

/*******************************************************/
/* This recursive Procedure moves the disks from Pin   */
/* One to Pin Three.                                   */
/*-----------------------------------------------------*/
/* Entry-Parameters:                                   */
/*    n ::= Number of Disks                            */
/*    l ::= Number of left Pin                         */
/*    m ::=   "    "  middle Pin                       */
/*    r ::=   "    "  right   "                        */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/

VOID turm (n,l,m,r)
long n,l,m,r;
{

   if (n == 1)
      {
               pole[r-1][height[r-1]] =
                                  pole[l-1][height[l-1]-1];


               height[r-1]++;
               draw(r-1,l-1);
               pole[l-1][height[l-1]-1] = 0;
               height[l-1]--;
      }
   else
      {
       turm (n-1,l,r,m);
               pole[r-1][height[r-1]] =
                                  pole[l-1][height[l-1]-1];
               height[r-1]++;
               draw(r-1,l-1);
               pole[l-1][height[l-1]-1] = 0;
               height[l-1]--;
```

```
        turm (n-1,m,l,r);
       }
}

/*******************************************************/
/* This Procedure opens the required Libraries and     */
/* a Screen.                                           */
/*-----------------------------------------------------*/
/* Entry-Parameters: None                              */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/

VOID init ()
{
      GfxBase = GFX OpenLibrary("graphics.library",0);
      IntuitionBase = INT OpenLibrary
("intuition.library",0);
      Screen = SCR OpenScreen (&NewScreen);
}

/*******************************************************/
/* This Procedure builds the three Towers initially.   */
/*-----------------------------------------------------*/
/* Entry-Parameters: None                              */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/

VOID build_up()
{
   long i,j;

   WaitTOF();          /* Synchronized Output with Beam */
   SetRast(RPort,0);
   SetAPen(RPort,1);

   for (j=0; j<3; j++)
      for (i=1; i<=disks; i++)
            RectFill (RPort,
                      pos[j]-(pole[j][disks-i]*width),
                      ((i-1)*high)+offset,
                      pos[j]+(pole[j][disks-i]*width),
                      (i*high)+offset);
}

/*******************************************************/
/* This Procedure erases and sets the top Disk for two */
/* Pins.                                               */
/*-----------------------------------------------------*/
/* Entry-Parameters:                                   */
/*    new : = Pin, where new disk is placed            */
/*    old : = Pin, where disk is removed from          */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/
```

```
VOID draw(new,old)
long new,old;
{
   SetAPen (RPort,0);                          /* Erase Disk */
   SetOPen (RPort,0);

   if (height[old]-1 == 0)                        /*last Pin
?*/

   RectFill (RPort,
             pos[old]-pole[old][height[old]-1]*width,
             (disks-height[old])*high+offset,
             pos[old]+pole[old][height[old]-1]*width,
             (disks-height[old]+1)*high+offset);
   else

   RectFill (RPort,
             pos[old]-pole[old][height[old]-1]*width,
             (disks-height[old])*high+offset,
             pos[old]+pole[old][height[old]-1]*width,
             (disks-height[old]+1)*high+offset-1);

   SetAPen (RPort,1);                        /* Write new Disk */
   SetOPen (RPort,2);

   RectFill (RPort,pos[old],
             (disks-height[old])*high+offset,
             pos[old],
             (disks-height[old]+1)*high+offset);

   RectFill (RPort,pos[new]-pole[new][height[new]-
1]*width,
             (disks-height[new])*high+offset,
             pos[new]+pole[new][height[new]-1]*width,
             (disks-height[new]+1)*high+offset);

   Delay(Screen->MouseX/10);
            /* Pause Depending upon the Mouse position */
}
```

If you look very closely, you can see that the rectangles drawn by the program are not completely filled. There are many small holes that allow the background to show through.

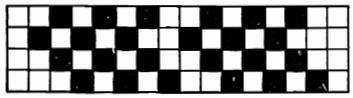
We were able to do this because we created our own fill pattern as we mentioned in the `Flood` function section.

In addition, there are two types of fill patterns, single colored and multi-colored. Both of these fill patterns require a height that is set in powers of two (1, 2, 4, 8...rows) and a width of 16 pixels (bits). We store the fill pattern in memory, for example in a word array.

Single colored patterns require only a single bit-plane. The colors of the `APen` (`BPen`) and drawing mode also have an effect on the area's appearance.

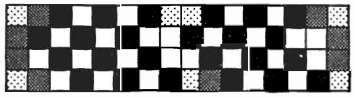
With multi-colored patterns you must specify a bit pattern for every bit-plane.

Single colored pattern

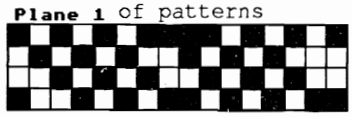


```
UWORD pattern[4] = {0x2A54,
                    0x5408,
                    0x2A54,
                    0x152A};
```

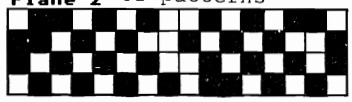
Multicolored pattern



Plane 1 of patterns



Plane 2 of patterns



```
UWORD pattern[8] = {0xABD5,
                    0x54A8,
                    0x2A54,
                    0x95AB,

                    0x6A56,
                    0xD4A9,
                    0xAA55,
                    0x556A};
```

By using the current drawing mode, you can write these patterns directly into the bit-planes. The only way we can affect them there is by changing the drawing mode.

To use these patterns with the graphic functions, we must get help from the `SetAfPt (&RastPort, &BitPattern, Height)` function (Set Area Fill Pattern).

We specify the height as an exponent of the power of two. So a pattern that is eight rows high then has a height of 3 ($2^3 = 8$).

However, this only applies to single colored patterns. When using multi-colored patterns, we set the exponent to a negative value, -3 instead of 3.

13.3 Polygon filling: Area, makes it possible

The `Area...` functions provide another method for filling in areas. These commands enable you to draw filled polygons with a frame drawn in the `OPen` color.

Before you are able to use these commands, some preparatory work is required.

The most important setup is, as in the `Flood` function, the `TmpRas` structure. You must initialize this in the RastPort where you want to draw your polygons.

You must also initialize the `AreaInfo` structure that will contain the individual base points of the polygon. To do this, use the function:

```
InitArea (&AreaInfo, &CoordsBuffer, NumCoords).
```

`InitArea` ensures that the `AreaInfo` structure for the buffer and the address of this buffer are established. These will later contain the polygon coordinates. You also have to set the maximum number of coordinates for this buffer (`NumCoords`). Your buffer must contain `(NumCoords+1)*5` bytes. The easiest way to do this is to assign the memory by using a char array `(char Buffer[(NumCoords +1)*5])`.

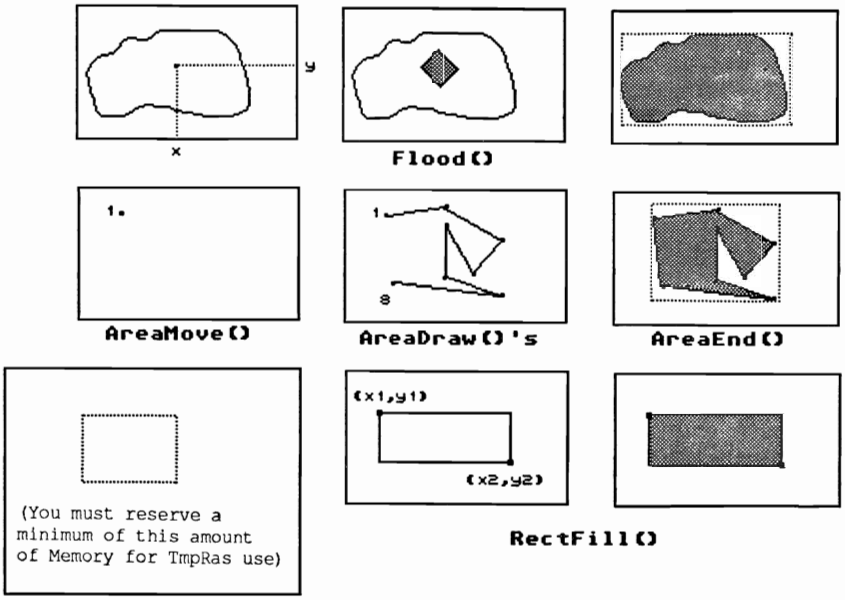
After you pass the initialized `AreaInfo` structure to the RastPort (`RastPort.AreaInfo = &AreaInfo`) you can set up your polygon. You always set the first coordinate with `AreaMove (&RastPort, x, y)`. `AreaMove` signals that you are creating a new polygon.

Now you can set the rest of the polygon base points using `AreaDraw (&RastPort, x, y)`. In this case `x` and `y` also specify the coordinates of the pixel within the RastPort.

You set the last pixel of a polygon with `AreaEnd (&RastPort, x, y)`. This also takes care of drawing the polygon (a submode of the `Flood` function) and filling it.

You can also use the `BNDRYOFF (&RastPort)` macro to turn off the boundary lines for the polygon.

The following figure illustrates all of the possible fill modes:



To complete this chapter, we have included a special program. This program allows you to outline, with the mouse, objects that we can then rotate three dimensionally.

```
/*******************************************************/
/*                      RotateIt.c                     */
/*                                                     */
/* This Program creates three dimensional Rotation   - */
/* objects that we build with Area...() and display.   */
/* Compiled with: Lattice V3.1                         */
/*  Will require major changes for Lattice V5          */
/*  and Aztec 3.6a                                     */
/* (c) Bruno Jennrich (The VS.F)                       */
/*******************************************************/

#include "exec/types.h"
#include "exec/memory.h"
#include "exec/devices.h"
#include "devices/printer.h"
#include "devices/keymap.h"
#include "graphics/gfx.h"
#include "graphics/gfxmacros.h"
#include "graphics/gfxbase.h"
#include "graphics/rastport.h"
#include "graphics/text.h"
#include "graphics/regions.h"
#include "graphics/gels.h"
#include "graphics/copper.h"
#include "hardware/blit.h"
#include "intuition/intuition.h"

#define WIDTH     640               /* Size of Screen */
#define HEIGHT    400
```

```
#define WIDTH     640                /* Size of Screen */
#define HEIGHT    400


#define INCREMENT 3                 /* Angle increment */
#define MAXITEMS  5                   /* 5 Menu-Items  */

#define MAXCOORDS 30     /* max. 30 Pixels for Outline */
#define MAXROTS   60                  /* 60 Rotations  */

#define RastPort Window->RPort    /* simple access to  */
                                  /* Window's RastPort */

#define MAXAREA (MAXCOORDS-1)*MAXROTS
                                   /* How many Areas ? */

char ASCString[6];                       /* For itoa() */

struct GfxBase *GfxBase=0;
struct IntuitionBase *IntuitionBase=0;
struct Screen *Screen=0;
struct Window *Window=0;               /* BasePointers */

struct MenuItem Items[MAXITEMS];      /* Menu Entries  */
struct IntuiText Texts[MAXITEMS];     /* Menu Text     */

struct Menu Menus;                                /* Menu */

struct NewWindow NewWindow;
struct NewScreen NewScreen;

char *IString[MAXITEMS] = {
                          "Shading","Hide It",
                          "Coordinates",
                          "Hardcopy","Quit"
                         };      /* Menu Text-Strings */

UWORD Colors[16] = {
                     0x0000,0x0111,0x0222,0x0333,
                     0x0444,0x0555,0x0666,0x0777,
                     0x0888,0x0bbb,0x0AAA,0x0BBB,
                     0x0CCC,0x0DDD,0x0EEE,0x0FFF
                   };
                  /* own Color Table with Gray shades  */

long Status = TRUE;
struct TmpRas TmpRas;          /* TmpRas for Area...() */
struct AreaInfo AreaInfo;    /* AreaInfo for Area...() */

long *Pointer=0;                       /* Help pointer */

UBYTE AreaBuffer[(4+1)*5];  /* Pixel Storage for Area. */
                           /* Always 4 (+1) Pixels per  */
                              /* Area                   */
int sintab[91] =
```

```
  {
   0,    285,   571,  857, 1142, 1427, 1712, 1996, 2280,
   2563, 2845, 3126, 3406, 3685, 3963, 4240, 4516, 4790,
   5062, 5334, 5603, 5871, 6137, 6401, 6663, 6924, 7182,
   7438, 7691, 7943, 8192, 8438, 8682, 8923, 9161, 9397,
   9630, 9860, 10086,10310,10531,10748,10963,11173,11381,
   11585,11785,11982,12175,12365,12550,12732,12910,13084,
   13254,13420,13582,13740,13894,14043,14188,14329,14466,
   14598,14725,14848,14967,15081,15190,15295,15395,15491,
   15582,15662,15749,15825,15897,15964,16025,16082,16135,
   16182,16224,16261,16294,16321,16244,16361,16374,16381,
   16384
  };
                /* Sine table, is calculated:          */
              /* sintab[x]=sin(x) * 16384 (x=1,2..,90) */
                /* See: sincos(), sin(), cos()         */

int Pixel[MAXROTS][MAXCOORDS][3];
int Mother[MAXROTS][MAXCOORDS][3];
        /* Mother-Array is calculated for the Rotation */
        /* object from the input outlines and is then  */
           /* used as a pattern to calculate the pixel */
           /* Arrays. This is necessary because throug */
           /* the many Transformations a single Array  */
        /* would suffer irreparable damage that would  */
        /* have a negative effect on the visible object*/

int Area[MAXAREA][4][3];
                /* Here we store the Areas calculated */
                /* from the pixels.                   */

int CoordArray[MAXCOORDS][2];
              /* Here we store the coordinates for the */
              /* entered outlines.                     */

int x[MAXROTS][MAXCOORDS];
int y[MAXROTS][MAXCOORDS];
              /* For Transformation  3D -> 2D (Screen) */

int Index[MAXAREA+1];                    /* For Sorting */

long ProzZ = 1500,              /* Projections Central */
     ProzY = 0,
     ProzX = 0;

long d = 0;             /* Separate Projection plane - */
                        /* Projection central          */

long bbpx = 0,                     /* Observation Point */
     bbpy = 0,
     bbpz = 0;

long AngleX=0,                       /* Display angle */
     AngleY=0,
     AngleZ=0;
```

```
VOID Show();
long Average();
char *itoa();
VOID Shade();
VOID Init();
VOID Rot();
long sincos();
long sin();
long cos();
long EnterCoords();
VOID make_menu();
struct IORequest *CreateExtIo();
VOID DeleteExtIO();
VOID hardcopy();

/*******************************************************/
/* This Function initializes the specified NewScreen   */
/* and NewWindow Structures for OpenScreen(), and      */
/* OpenWindow()                                        */
/*-----------------------------------------------------*/
/* Entry-Parameters: to initialize NewScreen-          */
/*                                Structure (NS)       */
/*                   to initialize NewWindow-          */
/*                                Structure (NW)       */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/

VOID InitScreenWindow (NS,NW)
struct NewScreen *NS;
struct NewWindow *NW;
{
   NS->LeftEdge = 0;        NS->TopEdge = 0;
   NS->Width = WIDTH;       NS->Height = HEIGHT;
   NS->Depth = 4;
   NS->DetailPen = 1;       NS->BlockPen = 0;
   NS->ViewModes = 0;

   if (WIDTH > 320) NS->ViewModes |= HIRES;
   if (HEIGHT > 200) NS->ViewModes |= LACE;

   NS->Type = CUSTOMSCREEN;
   NS->Font = NULL;
   NS->DefaultTitle = "";
   NS->Gadgets = NULL;      NS->CustomBitMap = NULL;

   NW->LeftEdge = 0;        NW->TopEdge = 0;
   NW->Width = WIDTH;       NW->Height = HEIGHT;
   NW->DetailPen = 6;       NW->BlockPen = 0;
   NW->IDCMPFlags = NULL;
   NW->Flags = BORDERLESS | ACTIVATE | NOCAREREFRESH
                                     | REPORTMOUSE;
   NW->FirstGadget = NULL;
   NW->CheckMark = NULL;    NW->Title = "";
   NW->Screen = NULL;       NW->BitMap = NULL;
   NW->MinWidth = NW->MinHeight =
   NW->MaxWidth = NW->MaxHeight = 0;
   NW->Type = CUSTOMSCREEN;
}

/*******************************************************/
/* This Function builds the Menu and the Individual,   */
```

```
   NS->Type = CUSTOMSCREEN;
   NS->Font = NULL;
   NS->DefaultTitle = "";
   NS->Gadgets = NULL;      NS->CustomBitMap = NULL;

   NW->LeftEdge = 0;        NW->TopEdge = 0;
   NW->Width = WIDTH;       NW->Height = HEIGHT;
   NW->DetailPen = 6;       NW->BlockPen = 0;
   NW->IDCMPFlags = NULL;
   NW->Flags = BORDERLESS | ACTIVATE | NOCAREREFRESH
                                     | REPORTMOUSE;
   NW->FirstGadget = NULL;
   NW->CheckMark = NULL;    NW->Title = "";
   NW->Screen = NULL;       NW->BitMap = NULL;
   NW->MinWidth = NW->MinHeight =
   NW->MaxWidth = NW->MaxHeight = 0;
   NW->Type = CUSTOMSCREEN;
}

/*******************************************************/
/* This Function builds the Menu and the Individual,   */
/* (Items).                                            */
/*-----------------------------------------------------*/
/* Entry-Parameters: None                              */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/

VOID BuildUpMenu()
{
   long i;

   for (i=0; i<MAXITEMS; i++)
      {
         Items[i].LeftEdge = 1;
         Items[i].TopEdge = i*10;
         Items[i].Width = 112;    Items[i].Height = 10;
         Items[i].Flags = ITEMTEXT | ITEMENABLED | HIGHCOMP;

         Items[i].MutualExclude = NULL;
         Items[i].ItemFill = (APTR) &Texts[i];
         Items[i].SelectFill = NULL;
         Items[i].Command = NULL;
         Items[i].SubItem =  NULL;
         Items[i].NextSelect = NULL;

         Texts[i].FrontPen = 6;     Texts[i].BackPen = 0;
         Texts[i].DrawMode = JAM1; Texts[i].LeftEdge = 1;
         Texts[i].TopEdge = 1;
         Texts[i].ITextFont = NULL;
         Texts[i].IText = IString[i];
         Texts[i].NextText = NULL;
      }

   for (i=0; i< MAXITEMS-1;i++)
      Items[i].NextItem = &Items[i+1];
```

```
   Items[MAXITEMS-1].NextItem = NULL;

   Menus.NextMenu = NULL;     Menus.LeftEdge = 10;
   Menus.TopEdge = 0;         Menus.Width = 110;
   Menus.Height = 10;         Menus.Flags = MENUENABLED;
   Menus.MenuName = " Projects";
   Menus.FirstItem = &Items[0];
}

/*******************************************************/
/* This Function ends the Program, but not before      */
/* closing all the open Libraries, Screens & Windows   */
/* and giving back all the extra allocated Memory.     */
/*-----------------------------------------------------*/
/* Entry-Parameters: Output-String (poss. ErrorMessage)*/
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/

VOID CloseIt(s)
char *s;
{
   puts(s);
   if (Pointer) FreeMem(Pointer,RASSIZE(WIDTH,HEIGHT/2));
                                        /* TmpRas-Memory */

   if (Window) {
                ClearMenuStrip(Window);
                CloseWindow(Window);
               }
   if (Screen) CloseScreen(Screen);
   if (GfxBase) CloseLibrary(GfxBase);
   if (IntuitionBase) CloseLibrary(IntuitionBase);

   exit (0);
}

/*******************************************************/
/* This opens all the required (Library, Screen,       */
/* Window) for the Program.                            */
/*-----------------------------------------------------*/
/* Entry-Parameters: None                              */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/

VOID OpenLibs()
{
   InitScreenWindow(&NewScreen,&NewWindow);

   if ((GfxBase = (struct GfxBase *)
             OpenLibrary("graphics.library",0)) == 0)
      CloseIt("No Graphics !!!");

   if ((IntuitionBase = (struct IntuitionBase *)
             OpenLibrary("intuition.library",0)) == 0)
```

```
      CloseIt("No Intuition !!!");

   if ((Screen = (struct Screen *)
            OpenScreen(&NewScreen)) == 0)
      CloseIt("No Screen !!!");

   NewWindow.Screen = Screen;

   if ((Window = (struct Window *)
            OpenWindow(&NewWindow)) == 0)
      CloseIt("No Window !!!");

   LoadRGB4 (&Screen->ViewPort, &Colors[0], 16);
                                   /* Load new Color */
   RemakeDisplay();                /* and display it */

   SetRGB4(&Screen->ViewPort,17,4,4,4);/*Mouse Pointer */
   SetRGB4(&Screen->ViewPort,18,8,8,8);  /* Color      */
   SetRGB4(&Screen->ViewPort,19,13,13,13);

   Init();
   BuildUpMenu();
}

/*******************************************************/
/* This Function switches the Intuitions Message       */
/* transfer off. This is important since normally all  */
/* Intuition messages are stored in a List (Queue) and */
/* processe one after another. Because we can not react*/
/* in time to these messages due to calculations and   */
/* drawing, it is best that you switch it off.         */
/* Otherwise messages are stored (like a keypress) and */
/* then later processed which would be confusing to the*/
/* User. For example,   after a rotation, the key press*/
/* is acted on that you pressed some time ago.         */
/*-----------------------------------------------------*/
/* Entry-Parameters: None                              */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/

VOID IDCMPoff()
{
   SetMenuStrip(Window,NULL);
   ModifyIDCMP(Window,NULL);
                    /* Separate Menu Row from Window  */
}

/*******************************************************/
/* This allows Intuition to send messages again.       */
/*-----------------------------------------------------*/
/* Entry-Parameters: None                              */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/
```

```
VOID IDCMPon()
{
   ModifyIDCMP(Window, RAWKEY | MOUSEBUTTONS | MENUPICK);
   SetMenuStrip(Window,&Menus);      /* Link Menu with  */
                                     /* Window          */
}

/*******************************************************/
/* This Routine uses the shape outline to calculate the*/
/* coordinates for the 3 dimensional rotation figure.  */
/* Rotation 3D (Y Axis).                               */
/*-----------------------------------------------------*/
/* Entry-Parameters:  rot = Number of Rotations        */
/*                    ko  = Number of Coordinates for  */
/*                          Outlines                   */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/

VOID Rotate(rot,ko)
long rot,ko;
{
   long i,
        j;
   long s,c;

   Move (RastPort,0,RastPort->TxBaseline);
   Text (RastPort, "Calculating Rotation-Matrix ...",29);

   for (i=0; i<rot; i++)
     {
      s=sin((360/rot)*i); /* Calculate Rotation angle  */
      c=cos((360/rot)*i);

      for (j=0; j<ko; j++)
         {
           Pixel[i][j][0] = Mother[i][j][0] =
                 (CoordArray[j][0]*c)/16384;      /* x */
           Pixel[i][j][1] = Mother[i][j][1] =
                 CoordArray[j][1];                /* y */
           Pixel[i][j][2] = Mother[i][j][2] =
                -(CoordArray[j][0]*s)/16384;      /* z */

         /* 'Mother' is unchanged and is used by the   */
         /* entire Program as a Mother-Mask            */
         }
     }
   Move (RastPort,0,RastPort->TxBaseline);
   Text (RastPort,"                                  ",29);
/*******************************************************/
/* This Routine calculates and draws the 2 dimensional */
/* Picture from the three dimensional Coordinates.     */
/*-----------------------------------------------------*/
/* Entry-Parameters:  rot = Number of Rotations        */
/*                    ko  = Number of Coordinates for  */
/*                          thr Outlines               */
```

```
/*-------------------------------------------------*/
/* Returned-Values: None                           */
/*************************************************/

VOID Show(rot,ko)
long rot,ko;
{
   long i,j,t;

   SetRast(RastPort,0);

   for (i=0; i<rot; i++)
      for (j=0; j<ko; j++)
         {
            t = (d-Pixel[i][j][2])/(ProzZ-Pixel[i][j][2]);

            x[i][j] = WIDTH/2-Pixel[i][j][0]+
                              (ProzX-Pixel[i][j][0])*t;
            y[i][j] = HEIGHT/2-Pixel[i][j][1]+(ProzY-
                        Pixel[i][j][1])*t;
                              /* 3d -> 2d transformation */

            if (j>0)
               {

                  Move(RastPort,x[i][j-1],y[i][j-1]);
                  Draw(RastPort,x[i][j],y[i][j]);
                           /* 'Longitudinal support' */
               }

            if (i>0)
               {
                  Move(RastPort,x[i-1][j],y[i-1][j]);
                  Draw(RastPort,x[i][j],y[i][j]);
                           /* 'Transverse support   ' */
               }
         }

   for (j=0; j<ko; j++)
      {
         Move(RastPort,x[0][j],y[0][j]);
         Draw(RastPort,x[rot-1][j],y[rot-1][j]);
                              /* last 'Transverse ' */
      }
/*************************************************/
/* This Routine calculates the middle Z-Coordinate for */
/* a specified Area                                */
/*-------------------------------------------------*/
/* Entry-Parameters:  i = Which Area               */
/*-------------------------------------------------*/
/* Returned=Value: Middle Z-Coordinate for Area i  */
/*************************************************/

long Average(i)
long i;
{
```

```
    long a,b,c,d;

    a=Area[i][0][2];
    b=Area[i][1][2];
    c=Area[i][2][2];
    d=Area[i][3][2];

    a = a+b+c+d;                    /* Picture Sum and */
    return (a/4);                   /* Average         */
}

/*******************************************************/
/* This Routine converts the received Integer Number to*/
/* ASCII.                                              */
/*-----------------------------------------------------*/
/* Entry-Parameter:  x = Integer                       */
/*                   position = How many Positions for */
/*                              entire ASCII String    */
/*-----------------------------------------------------*/
/* Returned-Value: Address of ASCII-String             */
/*******************************************************/

char *itoa(x,position)
long x,position;
{
    position--;
    if (x<0)
        {
          ASCString[0] = '-';
          x = -x;
        }
       else ASCString[0] = 32;                  /* ' ' */

    do {
          ASCString[position] = (x % 10)+'0';   /* + '0' */
          position--;
          x/=10;
       } while (position > 0);
    return (ASCString);
}

/*******************************************************/
/* This Routine calculates and draws the Areas of the  */
/* Rotation Object                                     */
/*-----------------------------------------------------*/
/* Entry-Parameters:  rot -  Number of Rotations       */
/*                    ko -   Number of Pixels per      */
/*                           Rotation Line             */
/*                    Mode - Color shading ?           */
/*                    Status - Recalculate Area ?      */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/

VOID Shade(rot,ko,Mode,Status)
long rot,ko,Mode,Status;
```

```
long i,j,k;
long count;     /* Number of Areas for Rot. Object  */
long hilf;
long x,y;
long t;
long a,b,c,e;
long w,col;

count = 0;

IDCMPoff();          /* No Messages from Intuition */

if (Status == FALSE)           /* Recalculate Area */
{                              /* and New sort     */
Move (RastPort,0,RastPort->TxBaseline);
Text (RastPort,"Calculating Areas... ",20);

for (i=0; i<(rot-1); i++)
   for (j=0; j<(ko-1); j++)
      {
         for (k=0; k<3; k++)
            {
               Area[count][0][k]=Pixel[i][j][k];
               Area[count][1][k]=Pixel[i][j+1][k];
               Area[count][2][k]=Pixel[i+1][j+1][k];
               Area[count][3][k]=Pixel[i+1][j][k];
            }
         count++;
      }


   for (j=0; j<(ko-1); j++)
      {
         for (k=0; k<3; k++)
            {
               Area[count][0][k]=Pixel[rot-1][j][k];
               Area[count][1][k]=Pixel[rot-1][j+1][k];
               Area[count][2][k]=Pixel[0][j+1][k];
               Area[count][3][k]=Pixel[0][j][k];
            }
      count++;
   }

Move (RastPort,0,RastPort->TxBaseline);
Text (RastPort,"Sorting Areas... ",20);

x = RastPort->cp_x;
y = RastPort->cp_y;

for (i=0; i<count; i++) Index[i+1] = i;
                        /* Initialize Index-Array */
for (i=2; i<count+1; i++)
   {
            /* Zero Area Components for sorting !!! */
      hilf = Average(Index[i]);
```

```
        Index[0] = Index[i];

        j = i-1;
        while (hilf < Average(Index[j]))
           {
              Index[j+1] = Index[j];
              j--;
           }
        Index[j+1] = Index[0];

        Move(RastPort,x,y);
        Text(RastPort,itoa((count-i),5),5);
     }

}

count = (ko-1)*rot+1;          /* 'count' must always */
SetRast(RastPort,0);           /* be checked          */

col = 0;
if (Mode == 0)
  {
     SetAPen (RastPort,0);/* Clear Areas and only   */
     SetOPen (RastPort,9);   /* draw Frame          */
  }
  else
  {
     w = count/9;          /* Gray scale calculations */
     SetOPen(RastPort,0);   /* no Frame             */
  }

   for (i=1; i<count; i++)
    {
      e=Index[i];
      for (j=0; j<4; j++)              /* Draw Areas */
         {
            a=Area[e][j][0];
            b=Area[e][j][1];
            c=Area[e][j][2];

            t = (d-c)/(ProzZ-c);
            x = WIDTH/2-a+(ProzX-a)*t;
            y = HEIGHT/2-b+(ProzY-b)*t;

            if (Mode == 1)           /* in Grey ? */
               {
                 if (i == w)
                   {
                     w += count/9;
                     col ++;
                   }

                 SetAPen (RastPort,4+col);
               }

            if (j==0) AreaMove (RastPort,x,y);
```

```
                                        /* Start Polygon */
            else AreaDraw (RastPort,x,y);
                                      /* New Polygon Pixel */
          }
        AreaEnd(RastPort);
                                     /* Polygon end + draw */
      }
    SetAPen (RastPort,9);
}

/*******************************************************/
/* This Routine initializes the TmpRas and AreaInfo    */
/* Structure for the RastPort.                         */
/*-----------------------------------------------------*/
/* Entry-Parameters: None                              */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/

VOID Init()
{
   InitArea (&AreaInfo,AreaBuffer,5);
                          /* always 4 (+1) Coordinates */

   if ((Pointer = (long *)AllocRaster(WIDTH,HEIGHT/2)) ==0)

      CloseIt("No free space !!!");
                                      /* Memory for TmpRas */

   RastPort->AreaInfo = &AreaInfo;
   RastPort->TmpRas = (struct TmpRas *)
    InitTmpRas (&TmpRas,Pointer,RASSIZE(WIDTH,HEIGHT/2));
}

/*******************************************************/
/* This Routine calculates the new Position for the    */
/* Rotation Object.                                    */
/*-----------------------------------------------------*/
/* Entry-Parameters:  AngleX, AngleY, AngleZ determine */
/*                     the new Object Position         */
/*                    rot - Number of Rotations        */
/*                    ko - Number of Lines per Rotation*/
/*                         Line                        */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/

VOID Rot(AngleX,AngleY,AngleZ,rot,ko)
long AngleX, AngleY, AngleZ,rot,ko;
{
   long sx,cx,sy,cy,sz,cz;
   long i,j;
   long hx,hy,hz;

      Move (RastPort,0,RastPort->TxBaseline);
      Text (RastPort,"Calculating new Position...",25);
```

```
        sx=sin(AngleX);      /* Calculate sin(x,y,z) and */
        cx=cos(AngleX);         /* cos (x,y,z) only once */
        sy=sin(AngleY);
        cy=cos(AngleY);
        sz=sin(AngleZ);
        cz=cos(AngleZ);

        for (i=0; i<rot; i++)/*Calculate Global-Rotation */
           for (j=0; j<ko; j++)
              {
                 hy = Mother[i][j][1]*cx/16384-
                      Mother[i][j][2]*sx/16384;
                 hz = Mother[i][j][1]*sx/16384+
                      Mother[i][j][2]*cx/16384;
                 Pixel[i][j][0] = Mother[i][j][0];
                 Pixel[i][j][1] = hy;
                 Pixel[i][j][2] = hz;

                 hx = Pixel[i][j][0]*cy/16384+
                          Pixel[i][j][2]*sy/16384;
                 hz =-Pixel[i][j][0]*sy/16384+
                          Pixel[i][j][2]*cy/16384;
                 Pixel[i][j][0] = hx;
                 Pixel[i][j][2] = hz+bbpz;

                 hx = Pixel[i][j][0]*cz/16384-
                          Pixel[i][j][1]*sz/16384;
                 hy = Pixel[i][j][0]*sz/16384+
                          Pixel[i][j][1]*cz/16384;
                 Pixel[i][j][0] = hx+bbpx;
                 Pixel[i][j][1] = hy+bbpy;
              }
        Move (RastPort,0,RastPort->TxBaseline);
        Text (RastPort,"                          ",25);
}

/*******************************************************/
/* This Function gets the Sine Value from the Table.   */
/*-----------------------------------------------------*/
/* Entry-Parameter:  x for sin(x)                      */
/*-----------------------------------------------------*/
/* Returned-Value: sin(x) * 16384                      */
/*******************************************************/

long sincos(x)
long x;
{
   long factor = 1;

   x %= 360;
   if (x>180)                          /* x > 180 Deg  */
      {
         x -= 180;
         factor = -1;
      }
```

```
   if (x>90) x = 180 - x;
   return (sintab[x]*factor);
}

/*******************************************************/
/* This Function calculates the Sine for x             */
/*-----------------------------------------------------*/
/* Entry-Parameter:  x                                 */
/*-----------------------------------------------------*/
/* Returned-Value: sin(x) * 16384                      */
/*******************************************************/

long sin(x)
long x;
{
   return (sincos(x));
}

/*******************************************************/
/* This Function calculates the Cosine for x           */
/*-----------------------------------------------------*/
/* Entry-Parameter:  x                                 */
/*-----------------------------------------------------*/
/* Returned-Value: cos(x) * 16384                      */
/*******************************************************/

long cos(x)
long x;
{
   return (sincos(x+90));     /* cos(x) = sin (90 + x) */
}

/*******************************************************/
/* This Function allows the input for the Outlines     */
/*-----------------------------------------------------*/
/* Entry-Parameter:   rot - How many Rotations are     */
/*                          allowed                    */
/*-----------------------------------------------------*/
/* Returned-Value: Number of entered Coordinates       */
/*******************************************************/

long EnterCoords(rot)
long rot;
{
   long Class,                  /* for Specifying the */
        Code;                   /* Intuition Messages */

   long Position;   /* Position where the Coordinates  */
   long x,y;        /* are stored in memory.           */

   long oldkoorx,oldkoory,oldx,oldy;

   long Ende = FALSE;              /* Entry Complete ? */

   ModifyIDCMP(Window, MOUSEMOVE | MOUSEBUTTONS | RAWKEY);
```

```
                                    /* Entry only requires */
                                    /* Mouse and Keyboard  */

SetRast(RastPort,0);                    /* Erase Bitmap   */

SetAPen (RastPort,9);     /* APen, BPen and DrawMode */
SetBPen (RastPort,0);        /* setup                */

SetDrMd (RastPort,JAM2);

Position = 0;      /* Where are Coordinates stored ? */

Move(RastPort,WIDTH/2,0);      /* Axis Intersections */
Draw(RastPort,WIDTH/2,HEIGHT);

Move(RastPort,0,HEIGHT/2);
Draw(RastPort,WIDTH,HEIGHT/2);

x=Screen->MouseX; /*actual Mouse position for Output*/
y=Screen->MouseY;

while (!Ende && (Position < MAXCOORDS))
   {
      if (1 << Window->UserPort->mp_SigBit)
                  /* Any Messages from Intuition ? */

      while (Message = (struct IntuiMessage *)
             GetMsg(Window->UserPort))      /* Yes */
         {
         Class = Message->Class;     /* Get Required */
         Code = Message->Code;       /* Data         */
         x = Message->MouseX;
         y = Message->MouseY;
         ReplyMsg(Message);            /* Have Found */
                                       /* Message !  */

         if (Position == 0)
              {
                oldx = x;
                oldy = y;
              }

         switch (Class)
            {
               case MOUSEBUTTONS:
                  if (Code == SELECTDOWN)
                     {
                          /* Coordinates relative   */
                          /* to Axis Intersection ! */

                        if (Position == 0)
                           {
                              oldkoorx = x;
                              oldkoory = y;
                           }
```

```
            CoordArray[Position][0] =
                      x-WIDTH/2;
            CoordArray[Position][1] =
                       HEIGHT/2-y;

            SetDrMd (RastPort,JAM2);
            Move (RastPort,oldkoorx,oldkoory);

            Draw (RastPort,x,y);

             /* draw small Intersection */
            Move (RastPort,x-3,y-3);
            Draw (RastPort,x+3,y+3);

            Move (RastPort,x-3,y+3);
            Draw (RastPort,x+3,y-3);

            Move (RastPort,x,y);
                 /* old Cursor position */

            oldkoorx = x;
            oldkoory = y;

            Position ++;
         }
break;

case MOUSEMOVE:

 if (Position > 0)
   {
    if ((x != oldx) || (y != oldy))
      {
         SetDrMd(RastPort,COMPLEMENT|JAM1);


         Move (RastPort,oldkoorx,oldkoory);

         Draw (RastPort,oldx,oldy);

         Move (RastPort,oldkoorx,oldkoory);

         Draw (RastPort,x,y);
            /* draw Lines to Actual  */
            /* Mouse position        */

         oldx = x;
         oldy = y;
      }
  }
break;

case RAWKEY:
   if ((Code == 0x50) && (Position > 0))

          Ende = TRUE;
```

```
                                                    /* F1 pressed  */
                        break;

                    }

                }
                SetDrMd(RastPort,JAM2);
                                /* output Mouse position */

                Move (RastPort,3*WIDTH/4,RastPort->TxBaseline);

                Text (RastPort,"X: ",3);
                Text (RastPort,itoa((x-WIDTH/2),4),4);
                Text (RastPort," Y: ",4);
                Text (RastPort,itoa((HEIGHT/2-y),4),4);
        }


    SetDrMd(RastPort,COMPLEMENT|JAM1);

    Move (RastPort,oldkoorx,oldkoory);
    Draw (RastPort,oldx,oldy);

    SetDrMd(RastPort,JAM2);
    Rotate(rot,Position);  /* calculate Rotation object */
    return (Position);
}

/*******************************************************/
/* This Routine allows the User to control the Rotation*/
/* Object                                              */
/*-----------------------------------------------------*/
/* Entry-Parameters:  rot - Number of Rotation lines    */
/*                    ko - Number of Pixels per        */
/*                         Rotation line               */
/*-----------------------------------------------------*/
/* Returned-Value: None                                */
/*******************************************************/

VOID make_menu(rot,ko)
long rot,ko;
{
    long Ende = FALSE;                  /* Program end  ? */
    long First = TRUE;              /* first time Shading ? */
    long Class,                      /* Intuition's Message */
         Code;
    long x,y;

    IDCMPon();                      /* Messages Please   ! */
    while (!Ende)
       {
        Wait (1 << Window->UserPort->mp_SigBit);
                                        /* Answered  ? */
         while (Message = (struct IntuiMessage *)
                GetMsg(Window->UserPort))       /* Yes */
            {
```

```
Class = Message->Class;
Code = Message->Code;
ReplyMsg(Message);
          /* We have received it, Thanks */

switch (Class)
   {
      case RAWKEY:
         x=y=0;
         switch (Code)
            {
               case 0x4c:   /* Arrow keys  */
                  y=-1;
               break;
               case 0x4d:
                  y=1;
               break;
               case 0x4f:
                  x=-1;
               break;
               case 0x4e:
                  x=1;
               break;
            }

         if ((x==0) && (y==0)) Status = TRUE;
         else
         {
           IDCMPoff();      /* no Intuition */
           if ((y == -1) && (AngleX == 0))
               AngleX=360;
           if ((x == -1) && (AngleZ == 0))
               AngleZ=360;
           AngleX += y*INCREMENT;
           AngleZ += x*INCREMENT;
           Rot(AngleX,AngleY,AngleZ,rot,ko);
           Show(rot,ko);
           IDCMPon();
           First = TRUE;
           Status = FALSE;
         }
        break;

        case MENUPICK:
            if (Code != MENUNULL)
              switch(MENUNUM (Code))
                 {
                    case 0:
                      switch (ITEMNUM (Code))
                        {
/* Shading */               case 0:
                              IDCMPoff();
                              if ((First) ||
                                  (!Status))
                              Shade(rot,ko,1,FALSE);
```

```
                                        else
                                        Shade(rot,ko,1,TRUE);

                                        First = FALSE;
                                        Status = TRUE;
                                        IDCMPon();
                                     break;

            /* Hide It */            case 1:
                                        IDCMPoff();
                                        if ((First) ||
                                            (!Status))
                                        Shade(rot,ko,0,FALSE);

                                        else
                                        Shade(rot,ko,0,TRUE);

                                        First = FALSE;
                                        Status = TRUE;
                                        IDCMPon();
                                     break;

         /* New Coordinates */       case 2:
                                        IDCMPoff();
                                        ko = EnterCoords
                                              (MAXROTS);
                                        AngleX=0;
                                        AngleZ=0;
                                             /* AngleY=0; */

                                        Show(rot,ko);
                                        First = TRUE;
                                        Status = FALSE;
                                        IDCMPon();
                                     break;

           /* Hardcopy */            case 3:
                                        IDCMPoff();
                                        hardcopy();
                                        IDCMPon();
                                     break;

           /* Quit */                case 4:
                                        Ende = TRUE;
                                     break;
                                  }
                               }
                            }
                         }

      }
}

/*******************************************************/
/* Here is where the Hardcopy Routines start ! Both of */
```

```
/* the following Functions are used for 'DumpRastPort' */
/* (External Input/OutPut)                              */
/******************************************************/

struct IORequest *CreateExtIO(ioReplyPort,size)
struct MsgPort *ioReplyPort;
long size;

   struct IORequest *ioReq;

   if (ioReplyPort == NULL) return (NULL);
   ioReq = (struct IORequest *)AllocMem (size,MEMF_CLEAR);

   if (ioReq == NULL) return (NULL);
   ioReq->io_Message.mn_Node.ln_Type = NT_MESSAGE;
   ioReq->io_Message.mn_Length = size;
   ioReq->io_Message.mn_ReplyPort = ioReplyPort;
   return (ioReq);
}

VOID DeleteExtIO (ioExt)
struct IORequest *ioExt;
{
   if (ioExt)
      {
         ioExt->io_Message.mn_Node.ln_Type = 0xff;
         ioExt->io_Device = (struct Device *)-1;
         ioExt->io_Unit = (struct Unit *)-1;
         FreeMem (ioExt,ioExt->io_Message.mn_Length);
      }
}

/******************************************************/
/* Here are the actual Hardcopy-Routines              */
/******************************************************/

VOID hardcopy()

   union printerIO
      {
         struct IOStdReq ios;
         struct IODRPReq iodrp;
         struct IOPrtCmdReq iope;
      };

   union printerIO *request;
   struct MsgPort *printerPort;

      printerPort = CreatePort ("printer.port",0);
      request = (union printerIO *)CreateExtIO (printerPort,

                     sizeof(union printerIO));

      if (OpenDevice("printer.device",0,request,0) != 0)
        {
            DeleteExtIO(request);       /* Open Printer */
```

```
            DeletePort(printerPort);
            CloseIt("Sorry, no printer !!!");
        }

    request->iodrp.io_Command = PRD_DUMPRPORT;
                            /* DumpRastPort Command  */

    request->iodrp.io_RastPort = RastPort;
                                /* Which RastPort ? */

    SetRGB4(&Screen->ViewPort,0,15,15,15);
                        /* Background color = White */

    request->iodrp.io_ColorMap = Screen->ViewPort.ColorMap;

                        /* for Shading on the Printer */

    request->iodrp.io_Modes = NewScreen.ViewModes;
                                /* Which ViewMode  ? */
    request->iodrp.io_SrcX = 0; /* upper left Corner */
    request->iodrp.io_SrcY = 0;
    request->iodrp.io_SrcWidth = WIDTH;
    request->iodrp.io_SrcHeight = HEIGHT;
                                    /* lower right    */
                                    /* Corner         */
    request->iodrp.io_DestCols = 0;
    request->iodrp.io_DestRows = 0;
    request->iodrp.io_Special = 0x004;       /* Flag */

    DoIO(request);                          /* Printing */

    CloseDevice(request);
    DeleteExtIO(request);         /* close everything */
    DeletePort(printerPort);
    SetRGB4(&Screen->ViewPort,0,0,0,0);
                        /* Background back to normal */
}

/*******************************************************/
/* Here is the MAIN PROGRAM:                           */
/*******************************************************/

main ()
{
   long Coords;

   OpenLibs();                              /* open Libs */

   Coords = EnterCoords(MAXROTS);      /* Get Outline */
   Show(MAXROTS,Coords);               /* Draw Object */

   make_menu(MAXROTS,Coords);                /* Menu */

   CloseIt("Bye Bye");
   return (0);
}
```

The program: After the program starts you can use the mouse to enter the outline of your object. When you finish setting all the pixels that design your object, press the <F1> key to see the three dimensional object on the screen. The object will automatically be generated without using the <F1> key if you have reached the value of MAXCOORDS that is allowed.

You can now use the cursor keys to rotate your object.

You can do much more with the object you just created. To display the object with a shaded gray scale, select the menu item `Shading`.

We create the shading effect with a simple algorithm. First we calculate an area list that contains the coordinates of all the filled areas' corner points. We then sort these points by their middle Z coordinate. After this, we first draw the areas that are furthest away from the viewer (naturally using `Area...` functions).

We use an incrementing counter that changes the shading color after drawing a specified number of areas.

The way the menu item `Hide it` functions is similar to the shading functions. Instead of shading the areas, it makes the area colors equal to the background color and therefore hides them. The framing color in `OPen` displays the object and `Hide it` simulates a hidden line algorithm.

When you get tired of your current rotation object you can select `Coordinates` from the menu and make a new object.

The menu items `Hardcopy` and `Quit` are self-explanatory.

Please make sure that the routines for calculating the rotation object are kept together. Also you can create your own objects (not just rotation objects) by providing the three dimensional coordinates yourself. With a slightly modified show routine and help from `Rot`, you can create and display your own rotation objects.

14. The Colormap functions

Now that you know how to work with the graphic primitives, the simple graphic functions, we will take a more detailed look into the Amiga's color capabilities.

At the moment, you know how to change your drawing colors by changing the foreground pen. However, we have not shown you how to change the color itself (the color stored in the color registers).

To do this you must change the corresponding entry in the colormap of the ViewPort. Then after modifying the colormap entry, you have to update the Copper list.

Basically each color entry is composed of 16 bits, or one word. At the moment, the Amiga uses only the lower 12 bits. The bit structure is composed of bits 0-3, which contain the blue component, bits 4-7, which contain the green component, and bits 8-11, which contain the red component. By changing the red, green and blue values with additive mixing, you can achieve any desired color. The term RGB monitor is derived from the red, green and blue color technique. With the Amiga, it is possible to have $2^{12} = 16^3 = 4096$ different colors.

When all four bits of the three color components are set, your color is white (R=15, G=15, B=15). When all of these bits are unset, your color is black (R=0, G=0, B=0).

The following section shows you how to change the colors.

14.1 Setting a new color palette

`LoadRGB4 (&ViewPort, &Palette, Color_Value)` transfers `Color_Value` words from the indicated palette (memory area) to the colormap of the selected ViewPort.

Most of the time the memory area or palette is a word array that you created. You copy the values from this array into the colormap of the selected ViewPort. However, you will not see any color changes until you update the part of the Copper list that uses the color registers.

To do this, use `MakeVPort`, `MrgCop` and `LoadView` and calculate a completely new Copper list (`RemakeDisplay` when using Intuition screens).

14.2 Changing one color

In the following function you do not have to change or update the entire Copper list. `SetRGB4 (&ViewPort, ColorReg, Red, Green, Blue)` not only changes the color for the selected color-register in the colormap, but also updates this color in the Copper list. The new color is modified through additive mixing of the red, green and blue and is immediately visible on the screen (when pixels in the color being changed exist). You can use `SetRGB4` to make an immediately visible color change to an individual color register.

Again, the red, green and blue color components are limited to values between 0 and 15 (%0000 to %1111).

14.3 Available colors

The opposite of SetRGB4 and LoadRGB4 is Color = GetRGB4 (ViewPort.ColorMap, Color_Reg). This command returns a color value that is the current color in the selected Color_Reg. The following small procedure allows you to separate this value back into the red, green and blue components:

```
Red   = (Color>>8) & 15
Green = (Color>>4) & 15
Blue  =  Color     & 15
```

The following program makes use of this calculation. We change the background color in the color register zero to the color currently under the mouse pointer.

```
/*******************************************************/
/*                     SimpleColor.c                   */
/*                                                     */
/* This Program uses ReadPixel(), Load-, Get-, and     */
/* SetRGB4().                                          */
/*                                                     */
/* Compiled with: Lattice V5                           */
/*                                                     */
/* (c) Bruno Jennrich                                  */
/*******************************************************/

#include "exec/types.h"
#include "exec/memory.h"
#include "exec/devices.h"
#include "graphics/gfx.h"
#include "graphics/text.h"
#include "graphics/regions.h"
#include "graphics/gfxbase.h"
#include "graphics/gfxmacros.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
#include "intuition/intuition.h"
#include "devices/keymap.h"
#include "hardware/blit.h"


#define Width 320
#define Height 200
#define Depth 5
#define MODES 0

struct GfxBase *GfxBase;
struct IntuitionBase *IntuitionBase;
```

```
struct Screen *Screen;
struct Window *Window;
struct IntuiMessage *Message;
struct RastPort *RPort;

struct NewScreen NewScreen =
            {0,0,
             Width,Height,Depth,
             0,1,
             MODES,
             CUSTOMSCREEN,
             NULL,
             NULL,
             NULL,NULL
            };

struct NewWindow NewWindow =
            {0,0,
             Width,Height,
             0,1,
             NULL,
             ACTIVATE|BORDERLESS,
             NULL,NULL,
             "Simple-Color-Selection",
             NULL,
             NULL,
             NULL,NULL,NULL,NULL,
             CUSTOMSCREEN
            };

char string[16][4] = {
                        {"0  "},{"1  "},{"2  "},{"3  "},
                        {"4  "},{"5  "},{"6  "},{"7  "},
                        {"8  "},{"9  "},{"A  "},{"B  "},
                        {"C  "},{"D  "},{"E  "},{"F  "}
                      };
                                              /* Hex-Values */
UWORD RED,
      GREEN,  /* Red-, Green, Blue components of Color */
      BLUE;

UWORD dummy;
int Length, x, y, i;
char text[] = "R  G  B";

UWORD Colors[] =
              {
                 0x0200,0x0405,0x060A,0x080F,
                 0x0214,0x0419,0x061E,0x0823,
                 0x0228,0x042d,0x0632,0x0837,
                 0x023C,0x0441,0x0646,0x084B,
                 0x0250,0x0455,0x065A,0x085F,
                 0x0264,0x0469,0x066E,0x0873,
                 0x0278,0x047D,0x0682,0x0887,
                 0x028C,0x0491,0x0696,0x089B,
                 0x02A0,0x04A5,0x06AA,0x08AF
```

```
                };                          /* own ColorMap */

char *LeftMouse = (char *) 0xbfe001;

extern struct IntuiMessage *GetMsg();
/*******************************************************/
/* Here we go  !                                       */
/*******************************************************/
main()
{
   if((GfxBase = (struct GfxBase *)
      OpenLibrary("graphics.library",0)) == NULL)
      {
         printf(" No Graphics !!!!!\n");
         Exit (0);
      }

   if((IntuitionBase = (struct IntuitionBase *)
      OpenLibrary("intuition.library",0))== NULL)
      {
         printf(" No Intuition !!!!!\n");
         goto cleanup1;
      }

   if ((Screen = (struct Screen *)OpenScreen(&NewScreen))
            == NULL)
      {
         printf("No Screen !!!!!!\n");
         goto cleanup2;
      }

   NewWindow.Screen = Screen;

   if ((Window = (struct Window *)
       OpenWindow(&NewWindow)) == NULL)
      {
         printf("No Window !!!!!!\n");
         goto cleanup3;
      }

   RPort = Window->RPort;

   LoadRGB4(&Screen->ViewPort, &Colors[0], 32);
   RemakeDisplay();              /* Load own ColorMap */
                                 /* and display it    */
   for (i=0;i<32;i++)
      {
         SetAPen(RPort,i);
         RectFill(RPort, i*(Width/32),(Height/100*90),
                         (i+1)*(Width/32)-1,Height-1);
      }                                /* Draw Rectangle */

   Length = TextLength(RPort,text,7);
   x = (Width/2)-(Length/2);
   y = (Height/2)+RPort->TxBaseline;
                                       /* center String */
```

```
    while ((*LeftMouse & 0x40) == 0x40)
        {
            dummy = ReadPixel(RPort,Screen->MouseX,
                                    Screen->MouseY);
            /* Read Color of Pixel under Mouse Pointer */

            SetAPen(RPort,dummy + 15);
            dummy =
                GetRGB4(Screen->ViewPort.ColorMap,dummy);

            RED = (dummy >> 8) & 15;
            GREEN = (dummy >> 4) & 15;
            BLUE = dummy & 15;
                                /* Extract Components */

            Move (RPort,x,y);
            Text (RPort,&text[0],7);
            Move (RPort,x,y+20);
                                 /* set Text Position */

            Text (RPort,&string[RED][0],3);
            Text (RPort,&string[GREEN][0],3);
            Text (RPort,&string[BLUE][0],3);
                                      /* output Text   */

            SetRGB4(&Screen->ViewPort,0,RED,GREEN,BLUE);
        }

             CloseWindow(Window);
    cleanup3: CloseScreen(Screen);
    cleanup2: CloseLibrary(GfxBase);
    cleanup1: CloseLibrary(IntuitionBase);
    return (0);
    /* Bye !!!*/
}
```

The four (4) in the name of the functions `Load`, `Get` and `SetRGB4` refers to the four bits which comprise a color component.

14.4 The pixel's color

Of course, in order for the above program to work, we must have a way to determine the color under the mouse pointer. `ColorReg = ReadPixel (&RastPort, x, y)` helps us do this by providing the number of the color register for the color at the specified x/y coordinate. Then we simply read the color register contents with `GetRBG4`, extract the red, green and blue components and use `SetRGB4` to write them into the background color register.

15. Text output

Most of the time, we will probably want a graphic display that contains more than just lines and pixels. Often, we need text combined with our graphics. One possibility would be to draw the text with lines, but this method would be very tiring.

The best solution is to use the built-in text output routines.

To output text at the current graphic cursor position, use `Text (&RastPort, "String", Number_of_Characters).` The `Number_of_Characters` variable determines how many characters of the specified string should be outputted.

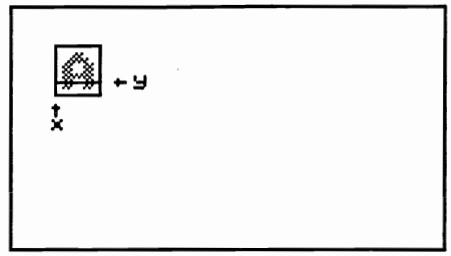
When using the above function, instead of counting the number of characters in the string, let the compiler calculate it for you. Using the `Count = strlen ("String")` function, which is part of almost every standard library in C compilers (`c.lib` for Lattice), is much easier.

Now you can use `Text(&RastPort, "String", strlen("String"))` to easily output your text.

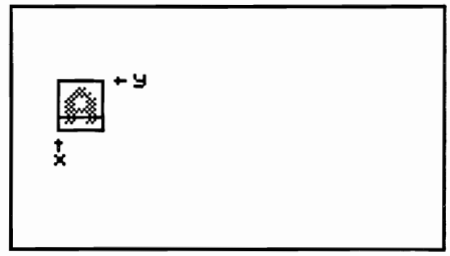
However, there is another problem with positioning your string. We use the actual graphic cursor position but the top line of the characters is not located at the Y coordinate of this position. Instead, the baseline is used for the actual position. So the text is displayed a little higher than expected.

To position the top row of the text string, add the desired Y position for the text to the baseline of the actual font:

```
Move (&RastPort,x,y);
```




```
Move (&RastPort,x,y+RastPort.TxBaseline);
```



15.1 The text length

Sometimes it is necessary to know the exact width of the text in the RastPort. You can determine this using `Width = TextLength (&RastPort, "String", strlen("String"))`. You can see that this function uses the same parameters as text. In this case, instead of outputting the string it calculates the width in pixels.

You could use this with CAD programs to determine if the output string will fit at the desired location.

You can also use this technique to center text on the screen. To do this you calculate the X coordinate as follows:

```
x = Width_of_Screen/2 - Width_of_String/2
```

To set the position use `Move (&RastPort, x, y)` to place the text in location it should appear (remember the baseline).

15.2 Fonts with the Amiga

The various character shapes are stored in the fonts which are actually a type of packed data array. We access the fonts using the `TextAttr (Text Attribute)` and `TextFont` structures.

15.3 Opening fonts

With the `TextAttr` structure you can set the name of the font that you want opened. However, you must also specify the font size since most fonts have more than one version that determines their different sizes. The best example of this is the CLI font. In Preferences you have the choice between 60 and 80 column text. This selection is possible because of two differently sized fonts (topaz.font with heights of 8 or 9 pixels).

The name of the font you specify with `TextAttr.ta_Name = "NAME"` must be a file in the `Sys:Fonts` directory. The selected file, with the extension `font`, is the header file for all of the different sized fonts with this name.

By using this header you can determine whether or not the selected font size and style are available in a particular font. Lets take a look at the `Ruby.Font`:

In the `Sys:Fonts` directory we find:

```
"ruby.font" (Header File)
...
ruby (dir)
```

`ruby.font` is the header file that contains the information for the individual font sizes. In the ruby directory we find the following files:

```
ruby (dir):

   12   15   8
```

These files represent three fonts that look the same, but are different sizes: 8, 12 and 15 pixels high.

To select the desired font height, use `TextAttr.ta_YSize` (for example: `TextAttr.ta_YSize = 8`).

It is also possible to set a specific style for your font by using `TextAttr.ta_Style`. The folowing `Font Style Flags` are used:

FS_NORMAL	The font is displayed without special styles.
FSF_UNDERLINED	All the characters of the font are underlined.
FSF_BOLD	The font characters are bold.
FSF_ITALICS	The font characters are italic.

You must specify, in the variable `TextAttr.ta_Flags`, whether the font is in the system font list (FPF_ROMFONT) or is on the disk (FPF_DISKFONT). If you removed the font from the system font list using `RemFont`, the FPF_REMOVED bit is set in `ta_flags`.

At the moment, you only need to know that all the fonts are on disk except Topaz.

Now we can finally open the fonts. To open fonts in the system font list, such as Topaz, use `TextFont = OpenFont (&TextAttr).`

We access the font through the textfont structure. `SetFont (&RastPort, TextFont)` switches fonts and any additional output that uses text is in the new font.

To use a font that is only on disk, you have to open the `DiskFont` library. To do this use `DiskFontBase = (ULONG *) OpenLibrary ("diskfont.library",0).` You can probably tell by the cast (`ULONG *`), that the `DiskFontBase` is a pointer to a `ULONG` variable. So a `DiskFontBase` structure does not exist.

Now you can use `TextFont = OpenDiskFont (&TextAttr)` to open a disk based font. Then use `SetFont` to make this font available to the `Text` function. However, `AddFont` (see Appendix B) can also be used to add this font to the system font list. Then in order to use the font, use the simpler `OpenFont` function (when you have not used `RemFont`).

15.4 Closing the font

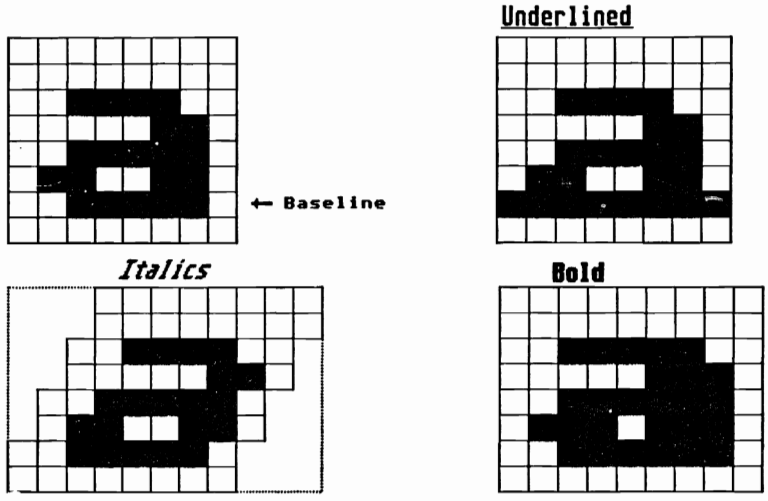
Just as with the libraries, you also have to close the fonts when you are finished with them. To do this we use the `CloseFont (TextFont)` command, which also closes disk fonts. If you are using a disk font, you must also close the diskfont library (`CloseLibrary (DiskFontBAse)`).

15.5 Software controlled text styles

Previously, we demonstrated how to select your font style using `TextAttr.ta_Style`. However, this style is firmly anchored in the bit pattern for the individual characters.

To change the font style without making a permanent change to the font bit pattern, use `SetSoftStyle (&RastPort, StyleBits, StyleEnable)`. This command changes the font shape output for text strings before they are written into the RastPort bit-map. For example, use underline by making `StyleBits = FSF_UNDERLINED`.

With italic characters every two rows of the character, starting from bottom to top, are shifted one pixel to the right:



Bold characters are actually output twice - once in the specified position and again one pixel to the right. Underline is achieved by setting all pixels in the baseline.

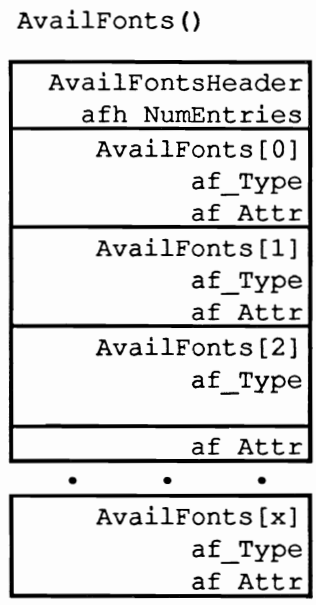
We have not discussed the variable `StyleEnable` yet. This variable contains the style flags that can still be set with `SetSoftStyle`. If a font is already italic it wouldn't be practical to modify it again and have double italics because the characters would be barely legible.

By using `StyleEnable = AskSoftStyle (&RastPort)` you can set all of the font flags in `StyleEnable` that still can be set.

15.6 Fonts a la carte

Now we will explain the actual meaning of `TextAttr.ta_Flags`. You can use this variable to determine whether a font is a disk or system font. However the variable will only contain this information once you perform a specific function with the `TextAttr` structure.

This function, called `Error = AvailFonts (&Buffer, Num_Bytes, Flags)`, fills the specified buffer with an `AvailFontsHeader` and the following `AvailFonts` structure.



The `AvailFonts` structure contains the `TextAttr` structure for all available fonts (`AvailFonts.af_Attr`). You can use the flags parameters to determine which font `TextAttr` structure you want - system fonts (`AFF_MEMORY`), disk fonts (`AFF_DISK`), or all fonts (`AFF_MEMORY | AFF_DISK`).

The variable `AvailFonts[i].af_Type` indicates where the font for this `TextAttr` structure is located on disk (`AFF_DISK`) or in the system font list (`AFF_MEMORY`).

The unique variable of `AvailFontsHeader` contains, after `AvailFonts`, the number of available fonts.
(`AvailFontsHeader.afh_NumEntries`).

The following program duplicates a short text string in all the possible styles and fonts:

```
/*******************************************************/
/*                      SetFont.c                      */
/*                                                     */
/* This Program demonstrates the use of:               */
/* AvailFonts(), OpenFont(), OpenDiskFont(), SetFont() */
/* AskSoftStyle() and SetSoftStyle().                  */
/*                                                     */
/* Compiled with: Lattice V5                           */
/*                                                     */
/* (c) Bruno Jennrich                                  */
/*******************************************************/

#include "exec/types.h"
#include "exec/memory.h"
#include "exec/devices.h"
#include "devices/keymap.h"
#include "graphics/gfx.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
#include "graphics/gfxbase.h"
#include "graphics/regions.h"
#include "hardware/blit.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
#include "libraries/diskfont.h"

#define WIDTH 320
#define BUFSIZE 1000          /* How big is the Buffer? */

#define RP Screen->RastPort      /* Access the RastPort */

struct GfxBase *GfxBase;             /* Our BasePointer */
struct IntuitionBase *IntuitionBase;
ULONG *DiskfontBase;
                      /* No special Base for DiskFonts */

struct NewScreen NewScreen =
                      {
                        0,0,WIDTH,200,2,
                        1,0,
                        0,
                        CUSTOMSCREEN,
                        NULL,
                        "",
                        NULL,NULL
                      };

struct AvailFontsHeader *Buffer;
struct AvailFonts *Availfonts;
                     /* One time Header and Pointer to */
                     /* AvailFonts-Structures          */
struct Screen *Screen;
```

```
int StyleEnable, Style;

/*******************************************************/
/* Here we go  !                                       */
/*******************************************************/

main()
{
   struct TextFont *TextFont;
   BOOL   Error;
   long   i,j,Length;
   char   *LeftMouse = (char *) 0xBFE001;

   if ((GfxBase = (struct GfxBase *)
            OpenLibrary("graphics.library",0)) == NULL)
              {
                  printf ("Sorry, No Graphics !!!\n");
                  goto cleanup3;
              }

   if ((IntuitionBase = (struct IntuitionBase *)
            OpenLibrary("intuition.library",0)) == NULL)
              {
                  printf ("Sorry, No Intuition !!!\n");
                  goto cleanup2;
              }

   if ((DiskfontBase = (ULONG *)
            OpenLibrary("diskfont.library",0)) == NULL)
              {
                  printf ("Sorry, No DiskFonts !!!\n");
                  goto cleanup1;
              }

   Screen = (struct Screen *) OpenScreen (&NewScreen);
   if (Screen == 0)
         {
            printf ("Sorry, No Screen !!!\n");
            goto cleanup0;
         }

   SetRGB4 (&Screen->ViewPort,0,0,0,0);/*Change colors */
   SetRGB4 (&Screen->ViewPort,1,15,0,0);
   SetRGB4 (&Screen->ViewPort,2,0,15,0);
   SetRGB4 (&Screen->ViewPort,3,0,0,15);

   SetRast (&RP,0);
               /* Clear Screen to get rid of 'Gadgets' */

   Buffer = (struct AvailFontsHeader *)
          AllocMem (BUFSIZE, MEMF_CLEAR | MEMF_CHIP);

   if (Buffer == 0)
             {
                printf ("Sorry, No Buffer !!!\n");
```

```
            goto cleanup0;
         }

SetAPen (&RP,3);
Length = TextLength (&RP,"AvailFonts()...",15);
Move (&RP,WIDTH/2-Length/2,30);
Text (&RP,"AvailFonts()...",15);
                                    /* Message to User */

Error = AvailFonts (Buffer, BUFSIZE, AFF_MEMORY |
                                       AFF_DISK);

Availfonts = (struct AvailFonts *) &Buffer[1];
                          /* Skip AvailFontsHeader */
                 /* (Buffer Type AvailFontsHeader) */

for (i=0; i<Buffer[0].afh_NumEntries; i++)
   {
      if (Availfonts[i].af_Type == AFF_DISK)
         TextFont = (struct TextFont *) OpenDiskFont
                     (&Availfonts[i].af_Attr);

                    /* Font on Disk or in Memory ? */

      else
            TextFont = (struct TextFont *) OpenFont
                     (&Availfonts[i].af_Attr);

      if (TextFont != 0)   /* Font opened correctly */
      {
         SetFont (&RP, TextFont);
                          /* Link Font to RastPort */

         StyleEnable = AskSoftStyle (&RP);
                      /* Which Styles are allowed ? */

         SetAPen (&RP,0);           /* Clear Screen */

         WaitTOF();              /* To prevent Blink */

         RectFill (&RP, 0,10,WIDTH,50);
                                  /* clear top Row */

         SetAPen (&RP,2);
         Length = TextLength (&RP,"The AMIGA Fonts:",
                              20);

         Move (&RP,WIDTH/2-Length/2,30);

         Text (&RP,"The AMIGA Fonts:",20);

         for (Style = FSF_ITALIC*2; Style>=0; Style--)
            {
               SetSoftStyle (&RP,Style,StyleEnable);
             /* Display all Styles once with a Loop */
```

```
                    Length = TextLength (&RP,
                      Availfonts[i].af_Attr.ta_Name,
                      strlen
(Availfonts[i].af_Attr.ta_Name));

                    Move (&RP,WIDTH/2-Length/2,100);

                    SetAPen (&RP,0);

                    WaitTOF();
                    RectFill (&RP, 0,80,WIDTH,110);
                    SetAPen (&RP,1);

                    Text (&RP,
                      Availfonts[i].af_Attr.ta_Name,
                      strlen
(Availfonts[i].af_Attr.ta_Name));
                                        /* output Font name */
                    j=0;
                    while (((*LeftMouse & 0x40) == 0x40) &&
                                (j<500000)) j++;
                    Delay(10);
                  }
              CloseFont (TextFont);
            }
        }


   FreeMem (Buffer, BUFSIZE);
                                      /* free extra Memory */
   cleanup0:   CloseScreen (Screen);
   cleanup1:   CloseLibrary(DiskfontBase);
   cleanup2:   CloseLibrary(IntuitionBase);
   cleanup3:   CloseLibrary(GfxBase);
   return (0);
}
```

16. The Blitter functions

Now we will discuss a set of functions that are only used with the Amiga. These functions are designed to take advantage of the graphic powers of the Blitter coprocessor. This coprocessor can transfer memory from one area to another at the rate of 125 KBytes per second. It also can simultaneously copy a memory area and perform logic functions on the data being moved.

We control the type of logic function performed with the so called "minterms". However, it is only possible to change the minterms by using two functions. The remaining functions we will present in this chapter are limited to copying and erasing data.

16.1 Clearing a memory area

To clear a memory area, use the `BltClear` function. This function was introduced in our first program to clear our self-defined bit-planes.

We must also provide the address of the memory area to be cleared. The `Num_Bytes` and `Flags` parameters indicate the size of the area we want cleared (`BltClear (&Memory, Num_Bytes, Flags)`).

When you set bit one of the `Flags` parameter (Flags = 2) then `Num_Bytes` is interpreted as a long word (32Bits). The lower 16 bits (0-15) determine the number of bytes (each equal to 8 pixels) per line to clear. The upper 16 bits (16-31) provide the number of lines to clear. The maximum value possible for the lower 16 bits is 128. This routine is designed to clear a maximum bit-plane size of 1024*1024 pixels (128 bytes * 8 pixels = 1024).

As you can see, this method for clearing a bit-plane is awkward. It is much easier when you set the `Flags` parameter to one and `Num_Bytes` contains the actual value for the number of bytes to clear.

The `RASSIZE (Width,Height)` macro calculates the number of bytes for you. The width and height parameters define, in pixels, the bit-plane area that you want cleared.

16.2 Copying data with the Blitter

We can also use this capability to copy data from one bit-map to another. The function `BltBitMap (&SourceBitMap, X1, Y1, &TargetBitMap, X2, Y2, Width, Height, Minterm, Mask, &TmpA)` helps us do this. It is possible to take a rectangular piece of data from a source bit-map and copy it to a target bit-map.

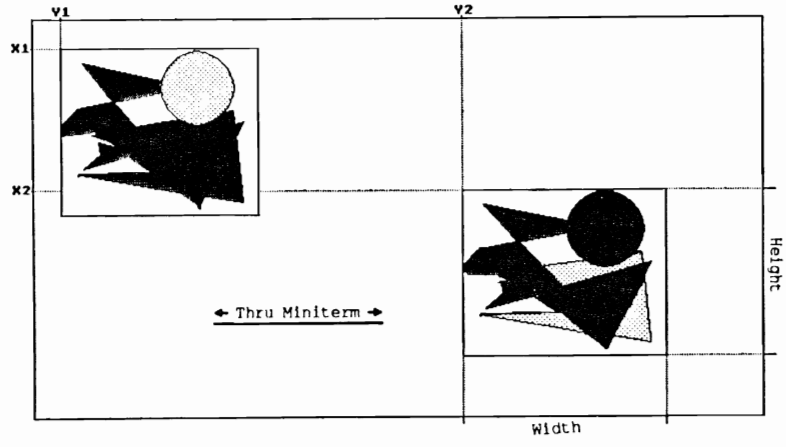
The source, from which you read the data, and the target, where you write the data, can also be in the same bit-map.

To select the rectangle you want to copy, simply specify the upper left corner of the rectangle in the (source-)bit-map (X1, Y1). Also set the target position, for the rectangle to be copied, as the upper left corner (X2, Y2) in the target bit-map.

Then set the width of the rectangle in pixels and the height in lines. Since the size is not going to change, we have to set these sizes only once. The Blitter only copies data; it does not change the size as we previously did with printer graphics.

In order to copy a rectangle with the Blitter, the width of the rectangle must be between 1 and 976 pixels and the height must be between 1 and 1023 lines.

Also remember that the entire rectangle must be inside the bit-map. If any portion of your rectangle overlaps a bit-map boundary, a Software Failure or Guru Mediation will occur.



If you carefully follow these instructions, everything should work properly. Now we will discuss the methods used for logic operations.

As we mentioned earlier, the Blitter can perform logic operations on the data while it is copying. For example, the data from the source and target can be logically modified with AND before being written to the target location.

We use the minterms to determine which logic operation will take place between the source and target areas (see the Appendix on hardware registers).

Each bit of this parameter (of type `char`) determines how you merge the source and target. However, when using this, you must remember to use only two sources and one target, which is used as source two. The Blitter is capable of merging three different sources to a fourth location, the target. In our example we use only source B and C. Target D and source C are the same (see hardware register `(bltcon)`).

Of the 256 possible minterms of the `BltBitMap` functions, our example leaves only 16 unused, which can be set in the upper four bits (7-4) of the minterm parameters. The various minterms are represented by the following bits:

Minterm	Bit	Value
BC (B and C)	7	0x80
$B\overline{C}$ (B and !C)	6	0x40
$\overline{B}C$ (!B and C)	5	0x20
$\overline{BC}$ (!B and !C)	4	0x10

When a single minterm is insufficient for your operation, you must OR the required bits, which represent the desired minterm, with each other.

Here are a few examples of how to use minterms:

With the combination `(!B AND !C) OR (!B AND C) = !B`, which equals a minterm value of 0x30, your target area is overwritten with an inverted version of the source.

With `(!B AND !C) OR (B AND !C) = !C` we invert only the target rectangle, each bit-plane separately.

AND can be replaced with * and OR or +. The hierarchy of operation rules (logic and arithmetic) also apply here.

Here are some often-used minterms:

0x60: Where pixels in C are unset, the pixels from source B are set. When a pixel in source B is set, then C remains as is.

0x80: In this case an AND is performed on the two rectangles to be merged. This means that a pixel from B can only be set if the same pixel in C is also set.

0xC0: Copies the data from source B to target C without any data changes.

With logic operations, the bit-maps are processed bit-plane for bit-plane. Unlike the positive/negative effect with single plane operations, these multi-plane operations can cause color changes.

In order to achieve the desired positive/negative effects, allow the `BltBitMap` function access to only specific bit-planes. You can use the `Mask` parameter to select which bit-planes you want affected. This parameter is also of the type `char` where you set a bit to determine which bit-planes can be accessed (bit 0 for the first bit-plane, etc.). Normally this value is 0xFF which allows all bit-planes to be affected (The function `SetWrMsk (&RastPort)` can be used to mask out specific bit-planes. Protecting a bit-plane from being changed with these methods will also affect other functions like `Draw`, `Text`, etc.).

The last parameter, the `TmpA-Pointer`, points to a buffer area of memory that is reserved for use when the source and target areas overlap. This only occurs when you use the `BltBitMap` command on a single bit-map. You can determine the size of this buffer according to the size of the overlapping area of source and target. If you are certain that the two areas do not overlap and that you don't need a buffer, you can set this parameter to zero.

A subset of the `BltBitMap` function is the `ClipBlit` function.

16.2.1 The ClipBlit function

The functions `BltBitMap` and `ClipBlit (&SourceRastPort, X1, Y1, &TargetRastPort, X2, Y2, Width Height, Minterm)` are almost the same. The difference between the two is that `BltBitMap` merges bit-maps and `ClipBlit` merges RastPort structures used by the Blitter. Again, both source and target can be in the same RastPort.

Another difference is that `ClipBlit` does not use the `Mask` parameter. It uses the RastPorts, where the `Mask` variable resides, which we can change with `SetWrMsk (&RastPort,Mask)`.

The remaining `BltBitMap` and `ClipBlits` parameters have the same rules and uses. However, for your own safety, you should always use `ClipBlit` with Intuition because `ClipBlit` is capable of clipping graphic information past a window border. So using `ClipBlit` with Intuition allows Intuition to control this problem without causing a system crash.

Here is an example Blitter program:

```
/*********************************************************/
/*                    BltBitMap.c                        */
/*                                                       */
/* This Program demonstrates the basic Functions of the*/
/* BltBitMap(), and ClipBlit() Commands.                 */
/*                                                       */
/* Compiled with: Lattice V5                             */
/*                                                       */
/* (c) Bruno Jennrich                                    */
/*********************************************************/

#include "exec/types.h"
#include "exec/memory.h"
#include "exec/devices.h"
#include "devices/keymap.h"
#include "graphics/gfx.h"
#include "graphics/text.h"
#include "graphics/regions.h"
#include "graphics/gfxmacros.h"
#include "graphics/gfxbase.h"
#include "graphics/gels.h"
#include "graphics/copper.h"
#include "intuition/intuition.h"
#include "hardware/blit.h"

#define Width 320               /* Screen Definitions */
#define Height 200
#define Depth 2
#define MODES 0

char *LeftMouse = (char *) 0xbfe001;

struct GfxBase *GfxBase;
struct IntuitionBase *IntuitionBase;

struct Screen *Screen;
struct Window *Window;
struct IntuiMessage *Message;
struct RastPort *RPort;

struct NewScreen NewScreen =
```

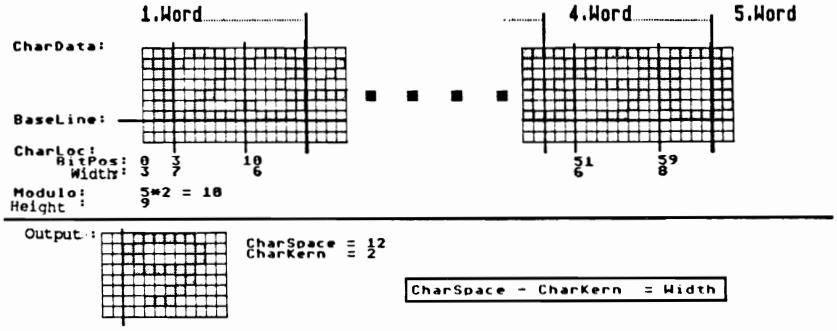
```
                {0,0,
                 Width,Height,Depth,
                 1,0,
                 MODES,
                 CUSTOMSCREEN,
                 NULL,
                 NULL,
                 NULL,NULL
                };

struct NewWindow NewWindow =
                {0,0,Width,Height,
                 1,0,NULL,
                 ACTIVATE|BORDERLESS,
                 NULL,NULL,
                 "Blit-BitMap",
                 NULL,
                 NULL,
                 NULL,NULL,NULL,NULL,
                 CUSTOMSCREEN
                };

int Length,
    x,y,
    oldx, oldy,
    i,j,Mint;

char Mins[16][5] = {
                      "$00 ","$10 ","$20 ","$30 ",
                      "$40 ","$50 ","$60 ","$70 ",
                      "$80 ","$90 ","$A0 ","$B0 ",
                      "$C0 ","$D0 ","$E0 ","$F0 "
                    };
APTR TmpA,
     Save;

BOOL Ende = FALSE;

/*******************************************************/
/* Here we go  !                                       */
/*******************************************************/

main()
{
   if((GfxBase = (struct GfxBase *)
      OpenLibrary("graphics.library",0)) == NULL)
      {
         printf(" No Graphics !!!!!");
         Exit (0);
      }

   if((IntuitionBase = (struct IntuitionBase *)
      OpenLibrary("intuition.library",0))== NULL)
      {
         printf(" No Intuition !!!!!");
         goto cleanup1;
```

```
    }

  if ((Screen = (struct Screen *)
      OpenScreen(&NewScreen)) == NULL)
    {
      printf("No Screen !!!!!!");
      goto cleanup2;
    }

  NewWindow.Screen = Screen;

  if ((Window = (struct Window *)
      OpenWindow(&NewWindow)) == NULL)
    {
      printf("No Window !!!!!!");
      goto cleanup3;
    }

  RPort = Window->RPort;
  Length = TextLength (RPort, NewWindow.Title,
                       strlen (NewWindow.Title))+5;

/*
  if ((TmpA = (APTR) AllocMem (RASSIZE(Width,Height),
                      MEMF_CHIP)) == NULL)
    {
      printf("No Rasterbuffer  !!!!!!");
      goto cleanup4;
    }                                                  */

  SetDrMd (RPort, JAM2);

  oldx = oldy = -1;

  while (Ende == FALSE)
    {
      x = Screen->MouseX;
      y = Screen->MouseY;

      if (((x == 0) && (y == 0)) &&
          ((*LeftMouse & 0x40) != 0x40)) Ende = TRUE;
                    /* Upper Left Clicked  == End  */

      if (((x != oldx) || (y != oldy)) &&
           (y > RPort->TxHeight+1)     &&
          ((*LeftMouse & 0x40) != 0x40))
        {
                  /* Only on Mouse Click 'blitter' */

                   /* Save = TmpA;
                      BltBitMap(&Screen->BitMap,0,0,
                      &Screen->BitMap,
                      x, y, Length, RPort->TxHeight,
                                Mint, 0xff, Save); */
          ClipBlit(RPort,0,0,
                   RPort,
```

```
                                x,y,Length,RPort->TxHeight,
                                Mint);              /* Blitter */

                        SetAPen (RPort,2);
                        Move (RPort,Length,RPort->TxBaseline);
                        Text (RPort,"  Minterms: ",12);
                        Text (RPort,&Mins[(Mint >> 4)][0],4);

                        Mint += 0x10;              /* Minterm ++ */
                        Mint &= 0xF0;

                        oldx = x;
                        oldy = y;
                    }
            }


    /* if (TmpA != 0) FreeMem(TmpA,RASSIZE(Width,Height)); */

        cleanup4: CloseWindow(Window);
        cleanup3: CloseScreen(Screen);
        cleanup2: CloseLibrary(IntuitionBase);
        cleanup1: CloseLibrary(GfxBase);

        return(0);    /* Bye */
    }
```

16.3 Reading data with the Blitter

It is possible to read data from a packed data array with the function `BltTemplate (&Buffer, BitPosition, Modulo, &Rastport, X,Y, Width, Height)`. For example, the Amiga fonts are stored in this type of packed data array and the individual characters are stored bit by bit in memory.



To read the individual character from this format, use the `BltTemplate` function. Before we discuss the parameter details of this function, we have a short program which simulates a mini font:

```
/*******************************************************/
/*                Figures.c                            */
/*                                                     */
/* This Program demonstrates the Functions of the      */
/* BltTemplate() Command on a Sample of a Pseudo-Font.*/
/*                                                     */
/* Compiled with: Lattice V5                           */
/*                                                     */
/* (c): Bruno Jennrich                                 */
/*******************************************************/

#include "exec/types.h"
#include "exec/memory.h"
#include "exec/devices.h"
#include "devices/keymap.h"
#include "graphics/gfx.h"
#include "graphics/text.h"
#include "graphics/regions.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
#include "graphics/clip.h"
#include "graphics/gfxmacros.h"
#include "graphics/view.h"
#include "graphics/gfxbase.h"
```

```
#include "intuition/intuition.h"
#include "hardware/blit.h"

#define Width 640                    /* Screen Definition */
#define Height 200
#define Depth 2
#define MODES HIRES

char *LeftMouse = (char *) 0xbfe001;

struct GfxBase *GfxBase;
struct IntuitionBase *IntuitionBase;

struct TextAttr TextAttr =
            {"topaz.font",
             8,
             FS_NORMAL,
             FPF_ROMFONT};

struct Screen *Screen;
struct Window *Window;
struct IntuiMessage *Message;
struct RastPort *RPort;

struct NewScreen NewScreen =      /* Screen-Definition */
            {0,0,
             Width,Height,Depth,
             1,0,
             MODES,
             CUSTOMSCREEN,
             &TextAttr,
             NULL,
             NULL,NULL
            };

struct NewWindow NewWindow =          /* Window-Defines */
            {0,0,
             Width,Height,
             1,0,
             NULL,
             ACTIVATE|BORDERLESS,
             NULL,NULL,
             NULL,
             NULL,
             NULL,
             NULL,NULL,NULL,NULL,
             CUSTOMSCREEN
            };

int i;

BOOL ende = FALSE,
     MOVE;

UWORD Class,
      Code;
```

```
UWORD CharData [] = {
   /* 1. Row */    0x001E,0x0CFE,0x11F9,0xC3C0,0x0000,
   /* 2. Row */    0x6F23,0x1CC0,0x221A,0x3467,0x8000,
   /*     "  */    0xF183,0x2CC0,0x4036,0x3C68,0xC000,
                  0x618E,0x4CF8,0xF863,0xCC78,0xC000,
                  0x6303,0xFF0D,0x8CC1,0xE7D8,0xC000,
                  0x6C43,0x0C07,0x8CC6,0x3198,0x8000,
                  0x7F82,0x0C05,0x8986,0x230f,0x0000,
                  0x003C,0x0CF8,0xF183,0xC600,0x0000,
   /* 9. Row */    0x0000,0x0800,0x0000,0x0000,0x0000
                   };
                      /* Numbers: 1 2 3 4 5 6 7 8 9 0 */

UWORD *ChipCharData,*Help;     /* Blitter can only use */
                               /* Memory 'below' 512K. */
                               /* For this reason we   */
                               /* write CharData[] in  */
                               /*  'Chip-Accessible'   */
                               /* Memory, or copied !! */
UWORD CharHeight = 9;
                     /* Each Character is 9 Rows high */

UWORD CharLoc[] = {0,3 ,3,7, 10,6, 16,8, 24,7,
                  31,7, 38,7, 45,7, 52,7, 59,7};
                                  /* Bitpos, Width  */

UWORD Modulo = 5*2;
                           /* 5 Words * 2 = 10 Bytes */

UWORD TxSpacing = 11;
                                /* largest Width + 3 */

char *String1 = "Well, how about these Numbers :";
char *String2 =
"Yes, and output using BltTemplate() instead of Text.";

VOID Figures();                /* Forward declaration */

/*******************************************************/
/* Here we go  !                                       */
/*******************************************************/

main()
{
   if((GfxBase = (struct GfxBase *)
      OpenLibrary("graphics.library",0)) == NULL)
      {
         printf(" No Graphics !!!!!");
         Exit (1000);
      }

   if((IntuitionBase = (struct IntuitionBase *)
      OpenLibrary("intuition.library",0))== NULL)
      {
         printf(" No Intuition !!!!!");
```

```
      goto cleanup1;
   }

if ((Screen = (struct Screen *)
    OpenScreen(&NewScreen)) == NULL)
   {
      printf("No Screen !!!!!!");
      goto cleanup2;
   }

NewWindow.Screen = Screen;  /* Screen-Structure for */
                             /* NewScreen           */

if ((Window = (struct Window *)
    OpenWindow(&NewWindow)) == NULL)
   {
      printf("No Window !!!!!!");
      goto cleanup3;
   }

ChipCharData = (UWORD*) AllocMem
          (sizeof(CharData),MEMF_CLEAR|MEMF_CHIP);

if (ChipCharData == 0)
  {
     printf (" No more Chip-Memory !!!\n");
     goto cleanup4;
  }

Help = ChipCharData;
for (i=0; i<(sizeof(CharData)/sizeof(UWORD));i++)
    {
       *Help = CharData[i];
       Help++;
    }

RPort = Window->RPort;

SetAPen (RPort,1);
SetDrMd (RPort,JAM1);

Move (RPort,20,20);
Text (RPort,String1,strlen(String1));
                                    /* output Text */

Move (RPort,20,35);
Figures();                        /* output Numbers */

Move(RPort,20,50);

SetDrMd (RPort,JAM2);
Figures();

SetDrMd (RPort,JAM1);
SetAPen (RPort,1);
                                          /* OUTLINE */
```

```
    Move (RPort,20,76);
    Figures();

    Move (RPort,20,74);
    Figures();

    Move (RPort,21,75);
    Figures();

    Move (RPort,19,75);
    Figures();

    SetAPen (RPort,2);
    Move (RPort,20,75);
    Figures();

    Move (RPort, 20,100);
    Text(RPort,String2,strlen(String2));

    while ((*LeftMouse & 0x40) == 0x40);

              FreeMem (ChipCharData,sizeof(CharData));
    cleanup4: CloseWindow(Window);
    cleanup3: CloseScreen(Screen);
    cleanup2: CloseLibrary(IntuitionBase);
    cleanup1: CloseLibrary(GfxBase);

    return (0); /* Bye */
}

/*****************************************************/
/* This Function is for reading the Data from the    */
/* Array.                                            */
/*---------------------------------------------------*/
/* Entry-Parameters: None                            */
/*---------------------------------------------------*/
/* Returned-Values: None                             */
/*****************************************************/

VOID Figures()
{
    int i;

    for (i=0; i < 10; i++)
        {
           BltTemplate(ChipCharData,CharLoc[i*2],Modulo,
                       RPort,RPort->cp_x,RPort->cp_y,
                       CharLoc[i*2+1], CharHeight);
                      /* Get' Data from CharData Array */

           Move (RPort,RPort->cp_x+TxSpacing,RPort->cp_y);
                           /* New Position for Output */
        }
}
```

In our program we filled a UWORD array (CharData) with a mini font consisting of the numbers zero through nine. Then we included the starting address of this array in the BltTemplate command.

We also had to provide the BltTemplate function information for the bit position at which a specific character begins. To do this we created an additional array that contains the beginning of a character (Charloc, and Element 0, 2, 4...) and the width in pixels (Charloc, Element 1, 3, 5...). We also made the character height constant (Amiga fonts do not allow a variable height per character).

Now let's discuss how to use arrays with BltTemplate.

Since CharData is the buffer we want to read our data from, the first parameter must be our buffer address. CharLoc is the array containing the bit positions for our characters and contains even spaced elements (0, 2, 4...). Specifying the bit position means that instead of a character beginning with a new byte, it begins at a specific position within a byte. This method saves a lot of memory.

Next we provide the modulo to the BltTemplate function. This value determines how many bytes and therefore pixels (1 byte = 8 pixels) we have to add to the actual bit position in order to read the data for the next "now". In other words, modulo provides the width of the array in bytes.

After specifying the RastPort where you want the data written, you set the position for the bit-map with X,Y. At this location, you are setting the upper left corner where the copied rectangle should be positioned.

Width and height set the width in pixels and height in rows for the rectangle you are copying. You can read the value for width from the uneven Charloc elements.

All of these parameters determine the size and screen position of the rectangle being read. However this function can't help you perform any logic operations on the data unless it is written to the bit-map first.

17. Amiga resolution modes

We are now going to discuss the resolution modes we have mentioned in the previous chapters. With the use of many program examples, we have demonstrated how to open windows and how to create your own graphics in these windows using the graphics functions.

You are probably wondering how many resolution modes exist and how to set them up in a ViewPort. However, some modes require more than just changing the mode variable of the View or ViewPort. We will provide additional information about this in the following section.

First we will present a few basics about resolution modes. You can separate them into four types:

The first type of mode sets the resolution, or number of pixels, for both vertical and horizontal directions.

The second type are the color modes that determine the number of colors you can display.

The third type consists of the special modes. These modes are involved with the software changes you can make rather than with the actual hardware control of resolutions.

Sprites and other special effects appear in the fourth type.

17.1 The resolution modes

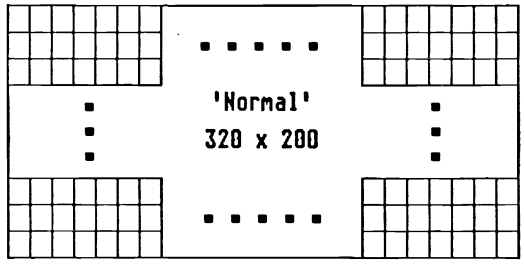
The quality of a graphic computer, especially the Amiga, is determined by the quality of its display resolution. The more pixels a computer can display on the monitor, the better the computer. Naturally, the speed of a computer is also an important factor, but at the moment we aren't considering this.

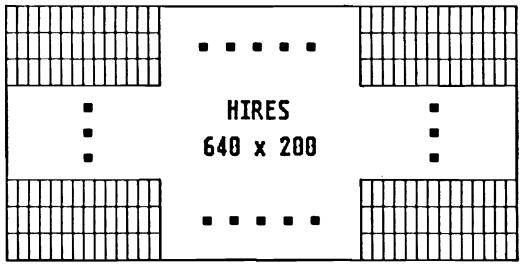
With a maximum of 640x400 pixels in the Interlaced mode, the Amiga isn't exactly one of the best graphic computers but it is definitely in the middle range. The Amiga's strength is determined not only by its resolution, but also by its ability to use 4096 colors (at one time).

The following table presents a small overview of the possible resolution modes:

1. 320 * 200 pixels with 32/64 colors (lo-res)
2. 320 * 400 pixels with 32/64 colors
3. 640 * 200 pixels with 16 colors (hi-res)
4. 640 * 400 pixels with 16 colors

Please remember that the above table applies to the visible portion of the monitor seen with the Workbench screen. If you use the Overscan techniques, which enable you to fill the entire screen without a border, you can attain resolutions up to 720x480 pixels.





As you can see, the various resolution modes are different in the X and Y directions by a factor of two. To display these different modes, you must use the flags, HIRES and LACE. These flags, which are located in the modes variables of the View and ViewPorts, can be set for various pixel resolutions.

Set the hi-res flag when you want to switch from 320 to 640 pixel resolution horizontally (ViewPort.Modes = View.Mode = HIRES). Of course you must also double the bit-map memory area where the doubled resolution will be displayed.

One side effect of the hi-res mode is that it also affects your color. More colors also means more data must be read and manipulated, per pixel, from the bit-map. Actually, the real problem involves the best way to manipulate large amounts of data in the least amount of time.

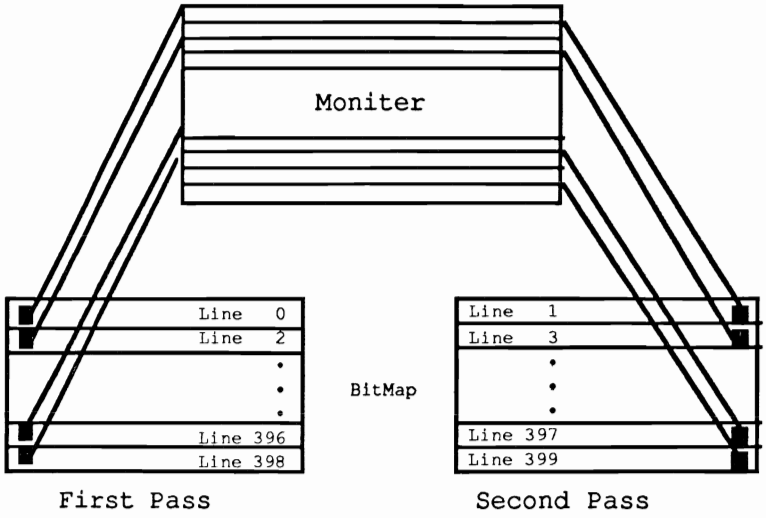
In normal resolution mode the electronic beam is working with an area about one millimeter square (mm^2) per pixel. The electronic beam scans (moves across) the screen with a constant speed regardless of the resolution mode. This means that, in normal mode, the Amiga has more time to read the required data from memory and to display it. In hi-res, the Amiga has to read twice as much data in the same amount of time because each mm^2 area contains two pixels.

So, the number of available colors is also limited. You can use up to six bit-planes in normal mode, but only four bit-planesie

with the hi-res mode. This means that you can only use 16 colors in hi-res.

To increase the vertical resolution, set the LACE flag (ViewPort.Modes = View.Modes = LACE). This increases your vertical resolution from 200 lines to 400 lines.

An old television technique provides us with doubled vertical resolution. The electronic beam, which moves across the screen makes two passes across the screen. The first pass displays all the even numbered lines of the bit-map, 0, 2, 4,..398. The second pass displays all the odd lines from the bit-map, 1, 3, 5..399.



The beam has to make two passes to display the entire bit-map. This causes the screen display frequency to be halved. So, instead of 60 frames a second, it only displays 30 frames a second and a visible flicker is produced.

This slight flicker can become very intense and annoying. If you use very bright colors in your ViewPorts, they will lose their intensity very quickly. This loss of intensity is caused by the limited light holding capabilities of the monitor, from the phosphor particles. These particles cannot be allowed to glow too long, especially in normal resolution modes. A phosphor particle with a high intensity that remains on the screen for an extended time can actually burn an image permanently into the screen.

This effect is very noticeable with monochrome monitors (you will see it quite often on IBM PCs and compatibles). Color monitors have an added disturbing effect that can be kept to a minimum when using interlace mode. This doesn't solve the problem but offers a slight compromise instead.

If you use interlace, you should use colors that aren't too bright and that have the least amount of contrast between them. This will produce the best display and least amount of flicker.

Interlace's biggest advantage is that it doesn't affect the number of colors you can use. It actually displays two pictures with a slight time difference and a one line horizontal offset. You can still use up to six bit-planes for your graphics (a single scan takes only a sixtieth of a second).

If you still are unsure about the organization of interlaced bit-maps, remember that you are only responsible for reserving enough memory. The Amiga will control the display characteristics (the even and odd scan lines). Just remember that when you reserve memory, you must reserve twice as much for interlace and four times as much for hi-res and interlace together.

Also remember that the resolution mode of the View must match the resolution mode of the ViewPort. When your View doesn't allow hi-res, you cannot use this mode with any of your ViewPorts. However, you can use lo-res with your ViewPorts even though you have set the hi-res flag in your View.

17.2 The color modes

Now we will discuss the color modes of the Amiga. You will discover that the power of Amiga color is astonishing.

First, the Amiga can easily display 64 colors at one time. Secondly, the Amiga can display all of its 4096 colors on the screen simultaneously.

17.2.1 The EXTRA_HALFBRITE mode

This mode allows you to display up to 64 colors at one time in a low resolution ViewPort.

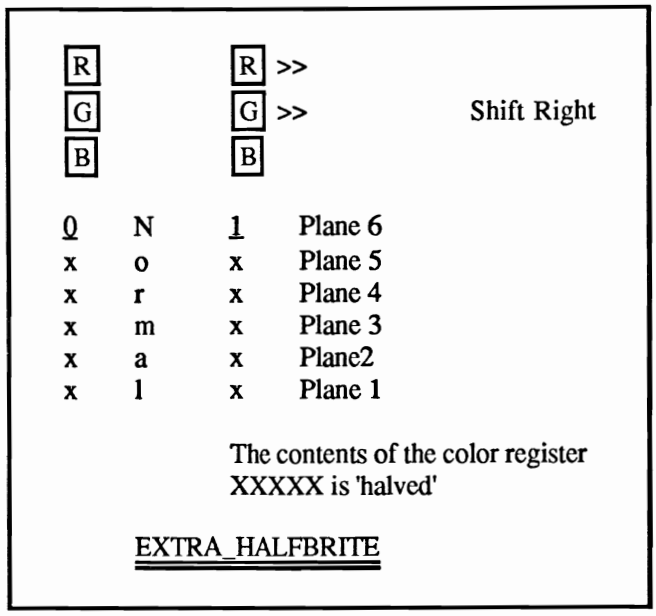
Just like in the lo-res mode, it is possible to use six bit-planes. In addition to the six bit-planes, you must also set the `EXTRA_HALFBRITE` flag in the mode variables of the ViewPort.

In order to discover what the halfbrite does, we must take a closer look at the Amiga.

Appendix C which covers the hardware registers indicates that the Amiga has only 32 color registers. To take advantage of all of these registers you must use five bit-planes.

The sixth bit-plane is where we encounter the halfbrite mode. When you set a bit in the sixth bit-plane, the pixel receives its color information from the color register being used by the other five bit-planes. However, this color is only half as bright. This is the secret of the halfbrite mode - making the intensity half as bright provides you with 32 additional colors. This half intensity is achieved by shifting the color register value to the right by one bit.

When you use `SetAPen` to add 32 to the value of the color register, each newly set pixel is displayed with half intensity.



17.2.2 Hold-and-modify (HAM) for 4096 colors

The ultimate in color graphics is the hold-and-modify or HAM mode.

With this mode you can display all 4096 colors on the screen. However, there are a few problems that can occur, but we will show you how to avoid them.

In order to use this mode, you first need five bit-planes, although six bit-planes are actually better to use with the HAM bit-map (Naturally, this prevents the use of the hi-res mode.).

In addition you must set the HAM flag in the mode variables of your ViewPort (`ViewPort.Modes = HAM`, View is not included yet).

In order to clearly explain the functions of this mode, we must review the construction of a color register in the ViewPort colormap. The 16 bits used for the color register contain red in bits 11-8, green in bits 7-4 and blue in bits 3-0.

Hold-and-modify pixels take their color from the pixel directly to the left. To determine the new color, two of the color components are used (hold) and a third color component is modified (modify).

We use the pixel bit pattern in bit-planes five and six to determine which two color components are used "as is" and which component is modified.

The value %01.... (=0x10) specifies that you want the red and green components used as is and bit-planes 1-4 to determine your new blue component for the pixel.

When you write a value of %10....(0x20) into bit-planes five and six, you are using the green and blue components and modifying the red component.

Setting both bits, or the value %11....(0x30), in planes five and six means you modify the green component.

If bit-planes five and six both contain a value of zero, use bit-planes 1 to 4 and your colors will be determined from the normal colormap. You can use these 16 normal colors as the starting point for your modifications and quickly and easily make color changes.

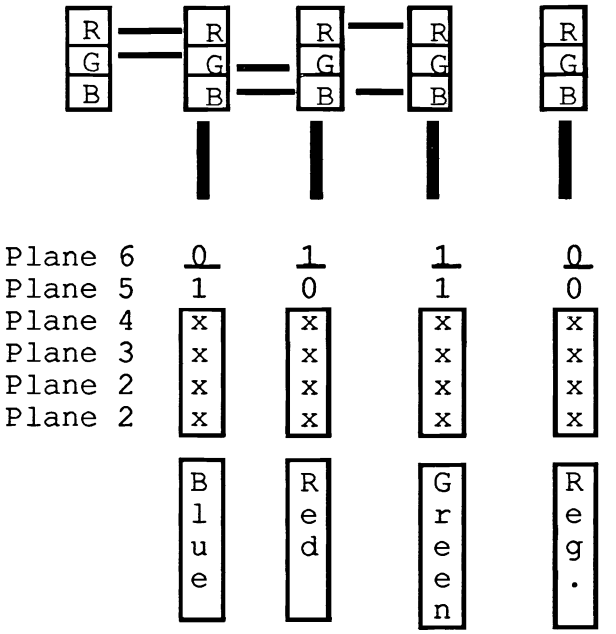
To use the red component of a pixel as the modifier, use `SetAPen` with a value of 0x20 and then add the new red intensity (0-15) to it. `SetAPen` determines which pixel bits in the bit-plane are set to zeros and ones. Now your set pixel (for example, using `WritePixel`) uses the green and blue components of the neighboring pixel and sets the new red component to your specified value.

If you want to use a color from the colormap, simply use `SetAPen` with the number of the desired color register without adding any additional value to it.

You should remember that hold-and-modify does not wrap from row to row. The last pixel in a row has no effect on the first pixel in the next row. To set a pixel with an X coordinate of zero you must clear the neighboring pixel's color components even though it really doesn't exist.

As we mentioned earlier, you must reserve five or six bit-planes for HAM mode. However, if you decide to use only five bit-planes (most likely for memory reasons), you must remember that plane six will be considered as clear. This means that you can only set a pixel with color from the colormap or only modify the blue component of a pixel.

The following figure illustrates the functions of the HAM mode:



Hold and Modify

17.3 The special modes

The previously explained modes pertain to how a pixel is displayed in a bit-map. The modes we will discuss now pertain to the organization of a bit-map. Following are explanations of how you can display overlapping and double-buffered bit-maps.

17.3.1 Dual playfield

You have probably never thought of displaying more than one bit-map in a ViewPort. However, this is possible with the Amiga.

By using the Dual Playfield mode, it is possible to display two bit-maps, with up to three bit-planes each, in a single ViewPort. These bit-maps are not merged or next to each other, but are on top of each other. Any set pixel in the top bit-map prevents you from seeing a set pixel in the bottom bit-map.

Besides setting the `DUALPF` flag in the `ViewPort.modes` variables, there is another way you can set up this mode.

We know that the `RasInfo` structure determines the connection between the ViewPort and the bit-map. While discussing this structure we told you that the pointer of the `RasInfo` structure for another `RasInfo` structure (`RasInfo.Next`) is empty. However, with Dual Playfield mode we need a second `RasInfo` structure connected to the first (`RasInfoA.Next = &RasInfoB; RasInfoB.Next = NULL;`). We also have to point to the additional bit-map (`RasInfoA.BitMap = &BitMapA; RasInfoB.BitMap = BitMapB`).

We represent playfield A by the bit-map pointed to by the first `RasInfo` structure that is also linked to the ViewPort (`ViewPort.RasInfo = &RasInfoA;`). Bit-map number two represents playfield B and normally is covered by playfield A. When you set a pixel in playfield A the pixel directly underneath it in playfield B is no longer visible. When a pixel in playfield A is clear, the pixel from playfield B is visible. The term playfield, which actually means bit-map, is used because you have created two areas that you can change any way you want.

You can open the screen once you have completed the following preparations: set the `ViewPort.Modes` variables for `DUALPF`, initialized both bit-maps, `RasInfos` and RastPorts. This means using `MakeVPort`, `MrgCop` and `LoadView` to open the screen.

To place playfield B on top instead of playfield A, simply set the `PFBA` flag in addition to the `DUALPF` flag (`ViewPort.Modes = DUALPF | PFBA`).

The colors for the separate bit-maps are determined as follows: playfield A uses color registers 0-7 and playfield B uses color registers 8-15. Register zero is normally used as the background color and register eight is taken from playfield B as transparent.

It is possible to use this mode to represent the cockpit of an airplane. Playfield A could be the interior of the cockpit and playfield B could be the view outside the cockpit.

You access the two bit-maps by using their corresponding initialized RastPorts.

17.3.2 Double buffering

Now we will show you how to create graphics in the background while completely different graphics are being displayed.

To create double buffered displays, first you need two bit-maps. The graphics data must be stored somewhere and both bit-maps must be the same size.

The rest is rather simple. You open a screen as usual, with an initialized View and ViewPort.

Then you set up a pointer in the `RasInfo` structure to the first bit-map and continue as usual (`MakeVPort`, `MrgCop`). However, `LoadView` is not used yet. First you must store the addresses of both hardware Copper lists into an array:

```
struct CprList *Copper[2][2];
...

Copper[0][0] = View.LOFCprList;
Copper[0][1] = View.SHFCprList;
```

Now you can point the `RasInfo` pointer for the bit-maps to the second bit-map. Then generate the second hardware Copper list by

using MakeVPort and MrgCop. However, before making the second list, you have to set both hardware Copper list pointers to zero. If you don't do this, MrgCop will assume a Copper list exists and the additional list will not be generated.

These additional Copper lists must also be stored using (Copper[1][0] = View.LOFCprList; Copper[1][1] = View.SHFCprList;). Now you can safely use LoadView which will display your second bit-map on the screen.

This is how you display one bit-map and modify a second bit-map through the RastPort structures that were initialized for both bit-maps. When the second bit-map is complete, you simply load the View structure with the saved address of the first bit-map's Copper list (View.LOFCprList = Copper[x][0]; and View.SHFCprList = Copper[x][1];). Then you use LoadView and continue. Of course you must make sure that you access the bit-maps correctly. You should organize the RastPorts of both bit-maps as arrays so you can switch between them similar to the Copper arrays.

The following program uses all of the described modes divided among several ViewPorts and displayed in a single View.

```
/*********************************************************/
/*                      The_Modes.c                      */
/*                                                       */
/* This Program demonstrates all of the possible         */
/* Display Modes on one Screen in 4 ViewPorts.           */
/*                                                       */
/* Compiled with: Lattice V5                             */
/*                                                       */
/* (c) Bruno Jennrich                                    */
/*********************************************************/

#include "exec/types.h"
#include "graphics/gfx.h"
#include "graphics/rastport.h"
#include "graphics/copper.h"
#include "graphics/view.h"
#include "graphics/gels.h"
#include "graphics/regions.h"
#include "graphics/clip.h"
#include "exec/exec.h"
#include "graphics/text.h"
#include "graphics/gfxbase.h"
#include "hardware/dmabits.h"
#include "hardware/custom.h"
#include "hardware/blit.h"
#include "devices/keymap.h"

#define DUAL1 3       /* Indexes for DualPlayfield     */
#define DUAL2 4       /* (Access to Rast and ViewPort) */
```

```
#define DBUFF1 DUAL1              /* Double-Buffer      */
#define DBUFF2 DUAL2+1            /* Indexes            */

UWORD BUFF[2] = {DBUFF1,DBUFF2};/* For Double-Buffer   */
                                /* (will be triggered) */

int Trigger;                        /* Triggervariable */

struct View View;
struct ViewPort ViewPorts[4];        /* Four ViewPorts */


struct RasInfo RasInfos[6];      /* Six BitMaps. For    */
struct BitMap BitMaps[6];        /* the 'above' 3 Modes */
struct RastPort RastPorts[6];    /* and also one used   */
                                 /* for DualPF 2 and    */
                                 /* for Double Buffer   */
                                 /* an additional (= 6) */

SHORT i,j,k,l,n;                     /* Help variables */

struct GfxBase *GfxBase;

struct View *oldview;           /* To save the old View */

UWORD ColorTable[4][16] = {{
                            0x000,0x00F,0x0F0,0xF00,
                            0xFF0,0x0FF,0xF0F,0xABC,
                            0xFF2,0xD30,0x7CF,0x4C3,
                            0x7A8,0x9C6,0x1D6,0xFD9
                           },
                           {
                            0x000,0x00F,0x0F0,0xF00,
                            0xFF0,0x0FF,0xF0F,0xABC,
                            0xFF2,0xD30,0x7CF,0x4C3,
                            0x7A8,0x9C6,0x1D6,0xFD9
                           },
                           {
                            0x000,0x100,0x200,0x300,
                            0x400,0x500,0x600,0x700,
                            0x800,0x900,0xA00,0xB00,
                            0xC00,0xD00,0xE00,0xF00,
                           },
                           {
                            0x000,0x00F,0x0F0,0x00f,
                            0xFF0,0x0FF,0xF0F,0x000,
                            0xFF2,0xD30,0x7CF,0x4C3,
                            0x7A8,0x9C6,0x1D6,0xFD9
                           }};
                    /* Color Palette for own ViewPorts */
int x,y;
int Red,Green,Blue;                  /* For HAM-Usage  */
int Length;

char *LeftMouse = (char *) 0xBFE001;   /* HARDWARE !!! */
```

```
char *LaceString = "Uff !! Very tight fit !";
char *HAMString = "The HAMmer";
char *HighString = "This is the HIRES Power !!!";
char *Pf2String = "Nothing in Foreground !!!";


struct cprlist *LOF[2];   /* Double Buffer CopperLists */
struct cprlist *SHF[2];

int Edges[4][2] = {{50,23},        /* For Movement of  */
                   {270,23},       /* Rectangle        */
                   {50,43},
                   {270,43}};

int Veloc[4][2] = {{2,-3},          /* 'Speed control' */
                   {3,2},
                   {-3,-2},
                   {-2,3}};


VOID Make();
VOID FreeMemory();

/*******************************************************/
/* Here we go  !!!                                     */
/*******************************************************/

main()
{
   if ((GfxBase = (struct GfxBase *)
      OpenLibrary("graphics.library",0)) == NULL)
      Exit (1000);

   oldview = GfxBase->ActiView;
                                      /* Save old View */

   InitView(&View);              /* Initialize new      */
   for (i=0; i<4; i++)           /* View, and ViewPorts */
      InitVPort(&ViewPorts[i]);

   View.ViewPort = &ViewPorts[0];  /* Link View with    */
                                   /* first ViewPort    */

   View.Modes = HIRES | LACE;/* Highest Resolution mode*/
                             /* for View so that LACE  */
                             /* and HIRES is possible   /
                             /* in ViewPorts.          */

   InitBitMap(&BitMaps[0],6,320,65);
                        /* 1.  LACE | HALFBRITE BitMap */

   InitBitMap(&BitMaps[1],4,640,33);
                                   /* 2.  HIRES BitMap */

   InitBitMap(&BitMaps[2],6,320,67);
```

```
                                        /* 3.  HAM BitMap */

  InitBitMap(&BitMaps[DUAL1],3,320,67);
  InitBitMap(&BitMaps[DUAL2],3,320,67);
                        /* 4.  DualPlayfield BitMaps */

  InitBitMap(&BitMaps[DBUFF2],3,320,67);
                          /* Double Buffering BitMap */
                          /* DUALPF1 BitMap is       */
                          /* buffered                */

  for (i=0; i<6; i++)
     {
        RasInfos[i].BitMap = &BitMaps[i];
        RasInfos[i].RxOffset = 0;
        RasInfos[i].RyOffset = 0;
        RasInfos[i].Next = NULL;
                          /* initialize all RasInfos */
     }


  ViewPorts[0].DxOffset = 0;  ViewPorts[0].DyOffset = 0;
  ViewPorts[0].DWidth = 320;  ViewPorts[0].DHeight = 64;
  ViewPorts[0].RasInfo = &RasInfos[0];
  ViewPorts[0].Modes = LACE | EXTRA_HALFBRITE;
  ViewPorts[0].Next = &ViewPorts[1];
          /* LACE | EXTRA_HALFBRITE ViewPort (first) */


  ViewPorts[1].DxOffset = 0; ViewPorts[1].DyOffset = 33;
  ViewPorts[1].DWidth = 640; ViewPorts[1].DHeight = 32;
  ViewPorts[1].RasInfo = &RasInfos[1];
  ViewPorts[1].Modes = HIRES;
  ViewPorts[1].Next = &ViewPorts[2];
                          /* HIRES ViewPort (second) */

  ViewPorts[2].DxOffset = 0; ViewPorts[2].DyOffset = 66;
  ViewPorts[2].DWidth = 320; ViewPorts[2].DHeight = 66;
  ViewPorts[2].RasInfo = &RasInfos[2];
  ViewPorts[2].Modes = HAM;
  ViewPorts[2].Next = &ViewPorts[3];
                             /* HAM ViewPort (third) */

  ViewPorts[3].DxOffset = 0;ViewPorts[3].DyOffset = 133;
  ViewPorts[3].DWidth = 320;  ViewPorts[3].DHeight = 66;
  ViewPorts[3].RasInfo = &RasInfos[DUAL1];
  ViewPorts[3].Modes = DUALPF | PFBA;
              /* Playfield 2 in front of Playfield 1 */

  ViewPorts[3].Next = NULL;
/* Dual-Playfield and Double Buffer ViewPort (fourth) */

  RasInfos[DUAL1].Next = &RasInfos[DUAL2];
                          /* DUALPF RasInfos linking */
```

```
   ViewPorts[0].ColorMap=(struct
ColorMap*)GetColorMap(16);
   ViewPorts[1].ColorMap=(struct
ColorMap*)GetColorMap(16);
   ViewPorts[2].ColorMap=(struct
ColorMap*)GetColorMap(16);
   ViewPorts[3].ColorMap=(struct
ColorMap*)GetColorMap(16);
            /* Get ColorMap-Memory for each Viewport */

   for (i=0;i<4;i++)

LoadRGB4(&ViewPorts[i],&ColorTable[i][0],16);
                    /* Each ViewPort has own Colors */

                    /* Get Memory now for the BitMaps */
   for (i=0; i<6; i++)
      {
         if ((BitMaps[0].Planes[i] =
            (PLANEPTR)AllocRaster(320,65)) == NULL)
            Exit (1000);          /* LACE Requirements */
         BltClear ((UBYTE *)BitMaps[0].Planes[i],
                        RASSIZE(320,65),0);
      }

   for (i=0; i<4; i++)
      {
        if ((BitMaps[1].Planes[i] =
           (PLANEPTR)AllocRaster(640,33)) == NULL)
           Exit (1000);          /* HIRES Requirements */
        BltClear ((UBYTE *)BitMaps[1].Planes[i],
                     RASSIZE(640,33),0);
      }

   for (i=0; i<6; i++)
      {
         if ((BitMaps[2].Planes[i] =
            (PLANEPTR)AllocRaster(320,67)) == NULL)
            Exit (1000);           /* HAM Requirements */
         BltClear ((UBYTE *)BitMaps[2].Planes[i],
                     RASSIZE(320,67),0);
      }

   for (i=0; i<3; i++)
      {
         if ((BitMaps[DUAL1].Planes[i] =
            (PLANEPTR)AllocRaster(320,67)) == NULL)
            Exit (1000);       /* DUALPF1 Requirements */
         BltClear ((UBYTE *)BitMaps[DUAL1].Planes[i],
                     RASSIZE(320,67),0);
      }

   for (i=0; i<3; i++)
      {
         if ((BitMaps[DUAL2].Planes[i] =
            (PLANEPTR)AllocRaster(320,67)) == NULL)
```

```
            Exit (1000);        /* DUALPF2 Requirements */
         BltClear ((UBYTE *)BitMaps[DUAL2].Planes[i],
                     RASSIZE(320,67),0);
      }

   for (i=0; i<3; i++)
      {
         if ((BitMaps[DBUFF2].Planes[i] =
            (PLANEPTR)AllocRaster(320,67)) == NULL)
            Exit (1000); /* Double Buffer Requirements */
         BltClear ((UBYTE *)BitMaps[DBUFF2].Planes[i],
                     RASSIZE(320,67),0);
      }

   for (i=0; i<6; i++)
      {
         InitRastPort (&RastPorts[i]);
         RastPorts[i].BitMap = &BitMaps[i];
                              /* Prepare RastPorts */
      }

   MakeVPort(&View,&ViewPorts[0]); /* Calculate Copper */
   MakeVPort(&View,&ViewPorts[1]); /* List for every   */
   MakeVPort(&View,&ViewPorts[2]); /* ViewPort         */
   MakeVPort(&View,&ViewPorts[3]);

   MrgCop (&View);          /* Merge all ViewPort Lists */

   LOF[0] = View.LOFCprList;         /* Store lists for */
   SHF[0] = View.SHFCprList;         /* Double Buffer   */

   ViewPorts[3].RasInfo = &RasInfos[DBUFF2];
   RasInfos[DBUFF2].Next = &RasInfos[DUAL2];
         /* Prepare RasInfo for DUALPF & Double Buffer */

   View.LOFCprList = NULL;
   View.SHFCprList = NULL;          /* Important !!!!!! */
                     /* Otherwise no second CopperList */

   MakeVPort(&View,&ViewPorts[0]);      /* Same as above*/
   MakeVPort(&View,&ViewPorts[1]);      /* once more    */
   MakeVPort(&View,&ViewPorts[2]);      /* for  Double- */
   MakeVPort(&View,&ViewPorts[3]);      /* Buffering    */

   MrgCop (&View);

   LOF[1] = View.LOFCprList;/* Store second CopperList */
   SHF[1] = View.SHFCprList;

             /* Here we write all of the BitMaps and  */
             /* build the Screens                     */

            /* The first is the HALFBRITE | LACE Mode */
   for (x=0; x<16 ; x++)
         {
            SetAPen(&RastPorts[0],x);
```

```
            RectFill(&RastPorts[0],x*(320/16)+1,1,
                                    (x+1)*(320/16)-1,31);
            SetAPen(&RastPorts[0],x+32);
                                      /* For Halfbrite */
            RectFill(&RastPorts[0],x*(320/16)+1,32,
                                    (x+1)*(320/16)-1,62);
         }

SetAPen (&RastPorts[0],2);

Move (&RastPorts[0],0,0);              /* Draw frame */
Draw (&RastPorts[0],319,0);
Draw (&RastPorts[0],319,63);
Draw (&RastPorts[0],0,63);
Draw (&RastPorts[0],0,0);

SetAPen(&RastPorts[0],5);             /* output Text */
SetDrMd (&RastPorts[0],JAM1);
Length=TextLength(&RastPorts[0],LaceString,
                  strlen(LaceString));
Move (&RastPorts[0],320/2-Length/2,
                    64/2+RastPorts[0].TxBaseline);
Text (&RastPorts[0],LaceString,strlen(LaceString));

                                    /* HIRES Viewport */
for (i=1;i<31;i++)
   {                               /* Draw a few lines */
      SetAPen(&RastPorts[1],i%16);
      Move(&RastPorts[1],1,i);
      Draw(&RastPorts[1],638,30-i);
   }

for (i=1;i<638;i++)
   {
      SetAPen(&RastPorts[1],i%16);
      Move(&RastPorts[1],i,1);
      Draw(&RastPorts[1],638-i,31);
   }

SetAPen(&RastPorts[1],2);

Move (&RastPorts[1],0,0);              /* New Frame */
Draw (&RastPorts[1],639,0);
Draw (&RastPorts[1],639,31);
Draw (&RastPorts[1],0,31);
Draw (&RastPorts[1],0,0);

SetDrMd(&RastPorts[1],JAM1);                /* Text */
SetAPen(&RastPorts[1],0);
Length = TextLength(&RastPorts[1],HighString,
                    strlen(HighString));
Move (&RastPorts[1],640/2-Length/2,
                    32/2+RastPorts[1].TxBaseline);
Text (&RastPorts[1],HighString,strlen(HighString));

                    /* And the Hold and Modify Mode */
```

```
SetAPen(&RastPorts[2],15+0x20);

Move (&RastPorts[2],0,0);                          /* Frame */
Draw (&RastPorts[2],319,0);
Draw (&RastPorts[2],319,65);
Draw (&RastPorts[2],0,65);
Draw (&RastPorts[2],0,0);

SetDrMd(&RastPorts[2],JAM1);
SetAPen(&RastPorts[2],14+0x30);

x = 0;
y = 1;
Red = 0;
Green = 0;

for (i=0; i<64; i++)                              /* 64 Rows */
   {
   x = 320/2-(64+8)/2;
   for (l=0; l<4; l++)
      {
         SetAPen(&RastPorts[2],(Red + 0x20));/* Red */
         WritePixel(&RastPorts[2],x,y+i);
         x++;
         SetAPen(&RastPorts[2],(Green + 0x30));
         WritePixel(&RastPorts[2],x,y+i);  /* Green */
         x++;
         for (Blue=0; Blue<16; Blue++)
            {
               SetAPen(&RastPorts[2],(Blue + 0x10));
               WritePixel(&RastPorts[2],x,y+i);
                                                  /*Blue*/
               x++;
            }
         Green++;
      }
   if (Green == 16){Green = 0; Red ++;}
   }

l = 2;
Length = TextLength(&RastPorts[2],HAMString,
                    strlen(HAMString));
for (i=1; i<(66-RastPorts[2].TxHeight);
            i += RastPorts[2].TxHeight)
{
   SetAPen(&RastPorts[2],l);
   l++;
   Move (&RastPorts[2],1,
                     i+RastPorts[2].TxBaseline);
   Text (&RastPorts[2],HAMString,strlen(HAMString));
   Move (&RastPorts[2],320-Length-1,       /* Text */
                     i+RastPorts[2].TxBaseline);
   Text (&RastPorts[2],HAMString,strlen(HAMString));
}

                                    /* DUALPF2 'color' */
```

```
SetAPen(&RastPorts[DUAL2],1);

RectFill (&RastPorts[DUAL2],0,0,319,6);
RectFill (&RastPorts[DUAL2],0,59,319,65);

RectFill (&RastPorts[DUAL2],0,6,16,59);
RectFill (&RastPorts[DUAL2],303,6,319,59);

SetAPen(&RastPorts[DUAL2],2);

Length = TextLength(&RastPorts[DUAL2],Pf2String,
                    strlen(Pf2String));
Move (&RastPorts[DUAL2],320/2-Length/2,
                  66/2+RastPorts[DUAL2].TxBaseline);
Text (&RastPorts[DUAL2],Pf2String,strlen(Pf2String));

Trigger = 0;

SetAPen(&RastPorts[BUFF[0]],10);  /* Set colors for */
SetAPen(&RastPorts[BUFF[1]],10);  /* both BitMaps   */

while ((*LeftMouse & 0x40) == 0x40)
   {                  /* Program End on Mouse Click */

      SetRast (&RastPorts[BUFF[Trigger]],0);
      Move(&RastPorts[BUFF[Trigger]],Edges[0][0],
                                     Edges[0][1]);
                                /* Draw a few lines */
      for (i=1; i<4; i++)
         {
           Draw(&RastPorts[BUFF[Trigger]],Edges[i][0],
                                          Edges[i][1]);
         }

      Draw(&RastPorts[BUFF[Trigger]],Edges[0][0],
                                     Edges[0][1]);
      for (i=0; i<4; i++)
         {
            Edges[i][0] += Veloc[i][0];
            Edges[i][1] += Veloc[i][1];

         if ((Edges[i][0] <= 0) || Edges[i][0] >= 319)
               {
                  Edges[i][0] -= Veloc[i][0];
                  Veloc[i][0] = -Veloc[i][0];
               }

         if ((Edges[i][1] < 0) || (Edges[i][1] >= 65))
               {
                  Edges[i][1] -= Veloc[i][1];
                  Veloc[i][1] = -Veloc[i][1];
               }
           }

      Make(&View,Trigger);        /* Switch View to  */
      Trigger ^= 1;               /* other BitMap    */
```

```
    }

  LoadView(oldview);                    /* Load Old View */
  FreeMemory();
  return(0);
}

/*******************************************************/
/* This Function manages the Switching between the two */
/* View CopperLists.                                   */
/*-----------------------------------------------------*/
/* Entry-Parameters:  View: that contains the new List,*/
/*                    Trigger: Which CopperList ?      */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/

VOID Make(View,Trigger)
struct View *View;
int Trigger;
  {
    View->LOFCprList = LOF[Trigger];
    View->SHFCprList = SHF[Trigger];
    LoadView(View);
    WaitTOF();
   }

/*******************************************************/
/* This Functions returns all memory to the system that*/
/* was used for the Bitmaps & any other reserved areas.*/
/*-----------------------------------------------------*/
/* Entry-Parameters: None                              */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/

VOID FreeMemory()
{
  for (i=0; i<6; i++)
        FreeRaster(BitMaps[0].Planes[i],320,64);
                                /* LACE | HALFBRITE */

  for (i=0; i<4; i++)
        FreeRaster(BitMaps[1].Planes[i],640,32);
                                        /* HIRES */

  for (i=0; i<6; i++)
        FreeRaster(BitMaps[2].Planes[i],320,66);
                                          /* HAM */

  for (i=0; i<3; i++)
        FreeRaster(BitMaps[DUAL1].Planes[i],320,66);
                                /* DualPlayfield 1 */

  for (i=0; i<3; i++)
```

```
            FreeRaster(BitMaps[DUAL2].Planes[i],320,66);
                                        /* DualPlayfield 2 */

    for (i=0; i<3; i++)
            FreeRaster(BitMaps[DBUFF2].Planes[i],320,66);
                                         /* Double Buffer */

    for (i=0;i<4;i++)
       FreeColorMap(ViewPorts[i].ColorMap);
                                        /* ColorMaps free */

    for (i=0; i<4; i++)
       FreeVPortCopLists(&ViewPorts[i]);
                             /* ViewPort CopperLists free */

    FreeCprList(LOF[0]);
    FreeCprList(SHF[0]);
    FreeCprList(LOF[1]);
    FreeCprList(SHF[1]);
                                      /* Free the Copper ! */

    CloseLibrary(GfxBase);
}
```

17.4 Other modes

The next group of modes include sprites and are called "other" modes. However, before we complete our discussion of modes, we will discuss another one that isn't actually a mode.

17.4.1 VP_HIDE

If you set this mode flag in a ViewPort, then the ViewPort will be ignored when you generate the Copper list. The result is that the ViewPort will not appear on the screen. (For example, you set this flag to place a screen in the background.)

17.4.2 Sprites

Now we will discuss sprites, one of the Amiga's technical accomplishments. In the next section we will show you the vsprites function. Right now we explain the hardware sprites in detail.

17.4.2.1 The hardware sprites

As the name indicates, the hardware sprites are an achievement of the Amiga's hardware (the vsprites include some skillful Copper programming). There are a total of eight hardware sprites. Each sprite has its own shape and motion that is independent of the others.

The sprites do have specific limitations. They can only be a maximum of 16 pixels wide, but any desired height. Also, they can have only three colors. However, a small trick enables you to use 15 colors.

To initialize the sprites, first set up a `SimpleSprite` structure (for example with `struct SimpleSprite SimpleSprite;`). The sprite functions `GetSprite`, `ChangeSprite` and `MoveSprite` give you access to your hardware sprite. However, you must set the

SPRITES flag in the ViewPorts mode variables before you can use either the hardware or VSprites sprites.

Once you complete the SimpleSprite structure and set the SPRITES flag, you can begin programming.

The first step to creating your own sprites is to call the function GetSprite. This function checks whether or not the sprite you want to use is available.

Use status = GetSprite (&SimpleSprite, SpriteNumber) to discover whether or not the sprite number SpriteNumber is available (sprites are numbered from zero to seven). It is possible that another running program is already using the requested sprite or this sprite is being used to display a VSprite. When two tasks attempt to use the same sprite, a conflict occurs which causes problems, for either task, in displaying the sprite.

When you have selected a specific sprite with SpriteNumber (0-7), the variable Status will contain this sprite number if the sprite is available. If it doesn't matter which sprite you use, set SpriteNumber to -1. Status will then contain the number of an available sprite that you can use. If Status contains a value of -1, this indicates that all sprites are being used at the moment.

Use the variable SimpleSprite.num with both sprite functions in order to point the SimpleSprite structure to the correct hardware sprite. This variable is stored in the SimpleSprite structure after you use GetSprite.

Additional sprite functions must specify only the SimpleSprite structure that contains the sprites you want to affect.

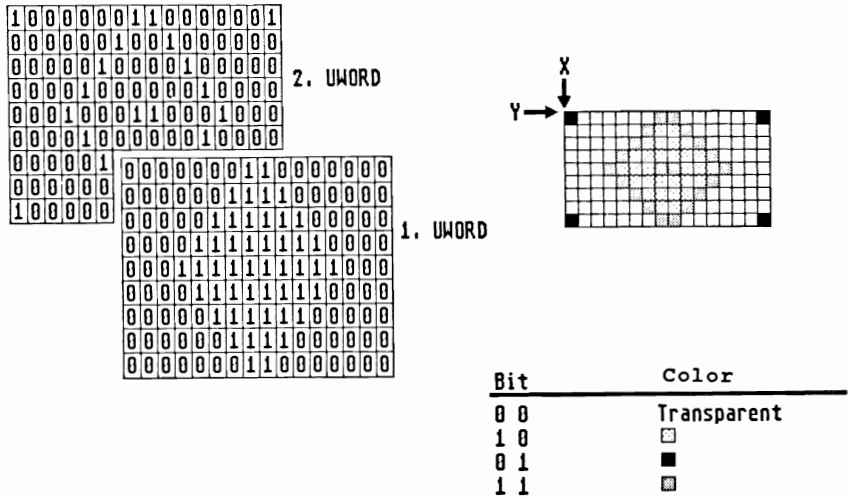
At the end of your program you should use FreeSprite (Status) to return the sprite that you used to the system. Remember to use the Status variable only for your assigned sprite from the system and not for other uses in the program.

Now we will continue to initialize your sprites. When you have received a sprite you must specify a height, in rows, for it. Use the variable SimpleSprite.height for your desired value.

Use the variables SimpleSprite.x and SimpleSprite.y to set the starting or initial screen position. This is the place where the sprite will appear on the screen when you use the ChangeSprite function.

The ChangeSprite function changes the appearance of your sprite. We previously mentioned that, without using any tricks, the hardware

sprites are always three colors. To display three colors a minimum of two color bits per pixel is required. You define each sprite row as two sequential UWORDS that are 16 bits wide (`#define UWORD unsigned int (s. 'exec/types.h')`).



Theoretically, you could use four colors instead of only three. However, the color "transparent", which is used when both bits are equal to zero isn't really a color.

To easily define the appearance of your sprite, define a two dimensional array. Then define all of the sprite rows with two UWORDS:

```
UWORD Appearance [Height][2] =
                        {
                            0x...., 0x...., /* first Row */

                            ...

                            0x...., 0x.... /* last Row   */
                        }
```

Simply passing the array starting address with `ChangeSprite (&ViewPort, &SimpleSprite, &SpriteData)` would be too easy. To create your sprites correctly, four additional UWORDs, that are used for system storage, are required. You must place two UWORDs before the array that determines the sprites appearance. Then you add the other two UWORDs at the end of the array. The following structure results for the `SpriteData` passed with `ChangeSprite`:

```
struct SpriteData
     {
          UWORD posctl[2];
          UWORD Appearance[Height][2];
          UWORD Reserved[2];
     }
```

You don't have to worry about the `pocst` and `reserved` arrays right now. All you need to know is that they must be included in order for everything to work properly and both of them must be set to zero.

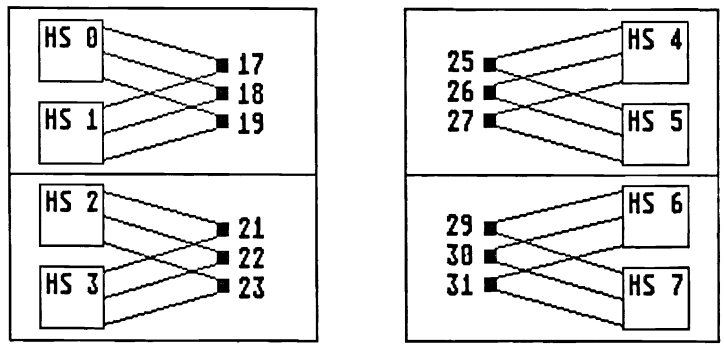
After setting the appearance of your sprite with `ChangeSprite`, you can make it appear at a selected position (`SimpleSprite.x/y`). Use the parameter ViewPort to determine whether your sprite appears relative to the ViewPort or relative to the View. You can select View by using a value of zero instead of a ViewPort address.

So far we have performed all the tasks needed to display a hardware sprite on the screen. To move your sprite to a new coordinate, use `MoveSprite (&ViewPort, &SimpleSprite, X,Y)`. Again the variable ViewPort determines whether the destination coordinates are relative to ViewPort or View.

Sprites use only lo-res positioning. If you want to move sprites in a `HIRES` or `LACE` ViewPort remember that position 320/200 places the sprite in the center of the screen. So, in order to use a sprite in these screens, you must change your coordinates by a factor of two.

Another problem that occurs with sprites is color. All hardware sprites cannot have individual color sets. They are paired to color registers so that every two sprites share four color registers (These color registers are the same as those in your RastPort that determine pixel colors).

The shared color registers look like this:



Color register and Hardware Sprites

The following bit pattern determines which color register is used for a specific sprite pixel:

1. UWORD	2. UWORD	Color Register for Sprite 0/1	2/3	4/5	6/7
1	0	17	21	25	29
0	1	18	22	26	30
1	1	19	23	27	31

A bit pattern of 00 is transparent and allows the background to show through.

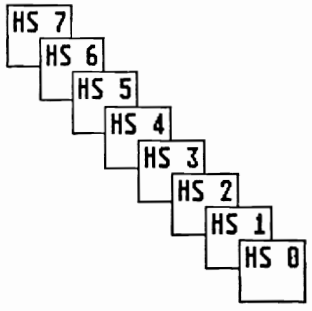
17.4.2.2 Hardware sprites in 15 colors

Now we will show you how to display sprites in 15 colors. As you can see, the sprites are divided into four even and odd numbered pairs (sprites 0/1, 2/3, 4/5, and 6/7).

After initializing the `SpriteData` structure for a pair of sprites, set the `SPRITE_ATTACHED` bit for this pair (`posctl[1] |=SPRITE_ATTACHED`). Now you can use the bit patterns of these sprites together, which gives you 15 colors plus one transparent (all bits of a pixel equal to zero).

Both sprites in a pair must occupy the same position and overlap completely. This overlapped area uses color registers 17-31 for colors that are determined by the sprites bit patterns. Also, you must set both `UWORD`s of the odd numbered sprite to their highest value and the `UWORD`s of the even numbered sprite to their lowest value for color selection.

Another quality of the sprites is that each one has an individual video priority. When all the sprites overlap, sprite zero is in front of sprite one, one is in front of two, etc. Sprite seven has the lowest priority.



17.4.2.3 Sprite collisions

Now that you know how to initialize and control the hardware sprites, you should have a few ideas about how to use them. One way these small graphics can be very useful is in action games.

For example, there must be a way to determine when a fired shot (such as a cannonball) hits a target. There are three possible ways to determine this.

The first, and easiest way is to constantly check your cannonball sprites position. When your sprite reaches the desired target position you can create the explosion or desired action effect by using a `ChangeSprite` loop.

The second collision control possibility involves sprites and background objects. You can check the area around a sprite with `ReadPixel`. If a pixel is set, you change the sprites movement direction.

The third method involves checking a hardware register that is responsible for collision control. For sprite to sprite collisions this method is much better than the first one we described. However, for sprite to background collisions, you can only test for true or not true. You can select specific bit-planes to test (see hardware registers `clxdat` and `clxcon`), but you do not have as much control as with the `ReadPixel` method.

Appendix C which covers the hardware registers provides complete bit information for using the `clxdat` hardware collision test.

The following program uses all three methods of testing for collisions and is quite a nice game.

```
/*******************************************************/
/*                    Breakout.c                       */
/*                                                     */
/* This is the familiar Game Breakout. We demonstrate  */
/* all the various methods used for                    */
/* 'Collision Detection'.                              */
/* Compiled with: Aztec V3.6a   cc +L -S Breakout,c    */
/*                              ln Breakout.o -lc32    */
/* (c) Bruno Jennrich                                  */
/*******************************************************/

#include "exec/types.h"
#include "exec/memory.h"
#include "exec/tasks.h"
#include "exec/devices.h"
```

```
#include "graphics/gfx.h"
#include "graphics/gfxbase.h"
#include "graphics/gfxmacros.h"
#include "graphics/regions.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
#include "graphics/sprite.h"
#include "intuition/intuition.h"
#include "hardware/blit.h"
#include "hardware/custom.h"
#include "devices/printer.h"
#include "devices/keymap.h"

#define WIDTH 320
#define HEIGHT 200

#define RP Window->RPort

struct GfxBase *GfxBase=0;
struct IntuitionBase *IntuitionBase=0;

struct Screen *Screen=0;
struct Window *Window=0;

struct NewWindow NewWindow;
struct NewScreen NewScreen;

struct SimpleSprite BumperL, BumperR, Ball;

UWORD Ball_Data[] = {           /* Sprite Definitions */
                    0,0,
                    0x0ff0,0x0000,
                    0x0ff0,0x0000,
                    0x0ff0,0x0000,
                    0x0ff0,0x0000,
                    0x0ff0,0x0000,
                    0x0ff0,0x0000,
                    0,0
                /* 1.Word,2.Word */
                  };

UWORD BumperR_Data[] = {
                    0,0,
                    0xffff,0x0000,
                    0xffff,0x0000,
                    0xffff,0x0000,
                    0xffff,0x0000,
                    0xffff,0x0000,
                    0xffff,0x0000,
                    0xffff,0x0000,
                    0,0
                  };

UWORD BumperL_Data[] = {
                    0,0,
                    0xffff,0x0000,
                    0xffff,0x0000,
                    0xffff,0x0000,
                    0xffff,0x0000,
                    0xffff,0x0000,
                    0xffff,0x0000,
```

```
                        0xffff,0x0000,
                        0xffff,0x0000,
                        0,0
                    };
                       /* SetPointer requires an Address  */
                       /* not equal to 0x00000000         */
UWORD *NewPointer;

UWORD *ChipBall,*ChipBumperR,*ChipBumperL,*Help;
                       /* Sprite-Data must be in Chip_Mem */
                       /* (lowest 512K) !!                */

UWORD Colors[32] = {
                        0x000,0x00f,0x0f0,0xf00,
                        0x0ff,0xff0,0xf0f,0xaaa,
                        0x789,0xa2e,0x5ad,0x8ed,
                        0xdb9,0xa56,0x789,0xabc,
                        0xfd0,0xccc,0x8e0,0xfb0,
                        0xfac,0x93f,0x06d,0x6fe,
                        0xfac,0x0db,0x4fb,0xf80,
                        0xf90,0xc5f,0xc80,0x999
                      };
                         /* Color definitions of Screen  */
char ASCString [11];                    /* For 'itoa()' */

struct TextAttr TextAttr = { "topaz.font",
                             8,0,FPF_DISKFONT|FPF_ROMFONT
                           };
                        /* Always 80 Characters per Row */

WORD Spr1, Spr2, Spr3;              /* Sprite Identifier */

long score = 0,                                 /* Score */
     balls = 3;                      /* How many Balls ? */
long New,                                 /* New Level ? */
     count;                  /* How many Bricks removed ? */

/*extern struct Custom custom;    in custom.h */
             /* Direct Access of the Hardware Register */

VOID InitScreenWindow();          /* Forward declaration */
VOID CloseIt();
VOID OpenLibs();
VOID DelBox();
VOID Score();
VOID itoa();
VOID AwaitClick();

/*******************************************************/
/* Here is where everything starts !!!                 */
/*******************************************************/

main ()
{
   long x,y,
        vx=-1,vy=-1,              /* Direction of Balls */
        px,py,                    /* Position of Balls  */
        MouseX,                  /* Position of Bumpers */
        Color;                      /* Color for Bricks */

   BOOL Ende = FALSE;                 /* Program end  ? */
   long i=0;
```

```
long level=0;                    /* Difficulty level */
char *LeftMouse = (char *)0xBFE001;
                                 /* Access from CIA  */
UWORD Collision;                 /* Which Collision  */


OpenLibs();

LoadRGB4 (&Screen->ViewPort,Colors,32);
                                 /* Load own Colors */
RemakeDisplay();         /* Link Colors to Screen !! */

NewPointer = (UWORD *)AllocMem(sizeof(UWORD)*2,
    MEMF_CLEAR|MEMF_CHIP);

if (NewPointer == 0) CloseIt("No NewPointer !!!\n");


SetPointer(Window,NewPointer,0,0,0,0);
                              /* Clear Mouse Pointer */
                     /* NewPointer-Pointer must != 0 */

MoveSprite (&Screen->ViewPort,&BumperL,0,200);
MoveSprite (&Screen->ViewPort,&BumperR,0,200);
                      /* Remove Bumper from ViewPort */
custom.clxcon = 0x0;
               /* Set Collision Control Register    */
               /* Provides us every Collision !     */
               /* We have to filter out the         */
               /* ones we are looking for !         */

while (!Ende)                     /* Program end  ? */
{
   balls = 3;
   level = 0;
   score = 0;

   while (balls > 0)                /* More Balls ? */
      {
         New = FALSE;
         count = 0;          /* All Bricks available */

         MoveSprite (&Screen->ViewPort,&Ball,0,200);
                       /* Remove Ball from ViewPort */
         SetRast (RP,0);             /* Clear BitMap */
         SetAPen (RP,1);         /* Foreground Color */
         SetDrMd (RP,JAM1);          /* Drawing Mode */

         Move (RP,0,HEIGHT-1);       /* Frame around */
         Draw (RP,0,10);
         Draw (RP,WIDTH-1,10);
         Draw (RP,WIDTH-1,HEIGHT-1);

         for (y=4; y<16; y++)        /* Build Bricks */
            for (x=2; x<15; x++)
               {
                  Color = 2 + level%64 - (x+y)%2;
                  if ((Color == 0) ||
                     (Color == 32)) Color ++;
                  SetAPen (RP,Color);
                  RectFill (RP,x*19+1,y*8+1,
                     (x+1)*19-1,(y+1)*8-1);
```

```
      }

                    /* Draw Beams for Level */
if (level%4 == 1)
   {
      SetAPen (RP,1);
      RectFill (RP,30,18*8,100,19*8-1);
      RectFill (RP,WIDTH-100,18*8,
                     WIDTH-30,19*8-1);
   }

if (level%4 == 2)
      RectFill (RP,WIDTH/2-75,17*8,
                WIDTH/2+75,18*8-1);

if (level%4 == 3)
   {
      SetAPen (RP,1);
      RectFill (RP,30,18*8,100,19*8-1);
      RectFill (RP,WIDTH-100,18*8,
                     WIDTH-30,19*8-1);
      RectFill (RP,WIDTH/2-75,17*8,
                    WIDTH/2+75,18*8-1);
   }

level++;
Score (score,level,balls);
AwaitClick();      /* Wait for Mouse click */

px = Screen->MouseX;       /* Calculate    */
if (px >= WIDTH-1) px -= 14;/* Ballposition*/
if (px > WIDTH/2) vx *= -1;
py = 192-7;


while ((balls > 0) && (New == FALSE))
                        /* Ball in Play ? */
   {
      MouseX = Screen->MouseX;

      MoveSprite (&Screen->ViewPort,
                  &BumperL,MouseX-16,192);
      MoveSprite (&Screen->ViewPort,
                  &BumperR,MouseX,192);
           /* Bumper controlled with Mouse */

      i++;     /* Always wait a little bit */
      if (i > 3)
         {
            px += vx;  /* new Ballposition */
            py += vy;

            Collision = custom.clxdat;
                /* read Collision register */
            i=0;
            if (py > 200)
                     /* Ball passed thru ? */
               {
                  balls--;
                  Score(score,level,balls);
                  vy = vx = -1;
                  while (((*LeftMouse & 0x40)
```

```
                                        == 0x40) &&
                                  (balls != 0))
                                 {
                                 MouseX = Screen->MouseX;
                                    MoveSprite
                                    (&Screen->ViewPort,
                                &BumperL, MouseX-16,192);
                                    MoveSprite
                                    (&Screen->ViewPort,
                                &BumperR, MouseX,192);
                                 }

                               px = MouseX;
                              if (px >= WIDTH-2) px -= 14;
                               if (px > WIDTH/2) vx *= -1;
                               py = 192-7;
                            }

                         if ((Collision & 0x3000) > 0)
                            {        /* Ball crashed to */
                               vy = -1; /* Bumper       */
                                    /* S2 with S4 or S6 */
                               if (px <= 0) px = 1;
                               if (px >= WIDTH-14)
                                          px=WIDTH-15;
                            }

/* Alternate Collision-Control: Actual Sprite Pos. is  */
/* compared with the Position of another Sprite.       */
/* Using a predetermined area limit value a Collision  */
/* can be registered:                                  */

/*                   if (py >= 192-6)
                        {
                           if ((px >= (MouseX-20)) &&
                           (px <= (MouseX+16)))
                              {
                                 vy = -1;
                                 py = 192-7;
                              }
                        }                              */

                      else
                        {
                         /* Use ReadPixel() to check   */
                         /* the Balls edges and react  */
                         /* accordingly                */

                           if (ReadPixel(RP,px+4,py+3)>0)
                                    /* calculate edge */
                              {
                                 vx *= -1;
                                 DelBox(px+4,py+3,level);
                              }

                          if (ReadPixel(RP,px+11,py+3)>0)
                                          /* left edge */
                              {
                                 vx *= -1;
                                DelBox(px+11,py+3,level);
                              }
```

```
                                if (ReadPixel(RP,px+8,py)>0)
                                                /* top edge */
                                   {
                                      vy *= -1;
                                      DelBox(px+8,py,level);
                                   }


                                if (ReadPixel(RP,px+8,py+6)>0)
                                            /* bottom edge */
                                   {
                                      vy *= -1;
                                      DelBox(px+8,py+6,level);
                                   }
                             }
                          MoveSprite (&Screen->ViewPort,
                                      &Ball,px,py);
                                 /* Ball to new Position */
                       }
                  }
               }

         MoveSprite (&Screen->ViewPort,&Ball,0,200);
         SetDrMd (RP,JAM2);
         SetAPen (RP,2);
         Move (RP,WIDTH/2-TextLength (RP,"Game Over",9)/2,100);
         Text (RP,"Game Over",9);
                                        /* Game is Over */
         AwaitClick();

         while ((*LeftMouse & 0x40) != 0x40)  i++;
                          /* How long is it pressed ? */

         if (i > 10000) Ende = TRUE;    /* Program end  ? */
      }
      ClearPointer(Window);                        /* Yes */
      CloseIt("Bye Bye");
      return(0);
}

/*******************************************************/
/* This Function Initializes the required NewScreen    */
/* and NewWindow Structures, and opens the Screen and  */
/* the Window. The Hardware-Sprites are also allocated */
/* and changed                                         */
/*-----------------------------------------------------*/
/* Entry-Parameters: Pointer to the initialized        */
/*                   NewScreen  and NewWindow Structure*/
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/

VOID InitScreenWindow (NS,NW)
struct NewScreen *NS;
struct NewWindow *NW;
{
   int i;

   NS->LeftEdge = 0;        NS->TopEdge = 0;
   NS->Width = WIDTH;       NS->Height = HEIGHT;
   NS->Depth = 6;
   NS->DetailPen = 1;       NS->BlockPen = 0;
```

```
NS->ViewModes = 0;

if (WIDTH > 320) NS->ViewModes |= HIRES;
if (HEIGHT > 200) NS->ViewModes |= LACE;
NS->ViewModes |= SPRITES | EXTRA_HALFBRITE;
             /* Mode SPRITES for the use of Sprites */

NS->Type = CUSTOMSCREEN;
NS->Font = &TextAttr;
NS->DefaultTitle = "";
NS->Gadgets = NULL;      NS->CustomBitMap = NULL;

NW->LeftEdge = 0;        NW->TopEdge = 0;
NW->Width = WIDTH;       NW->Height = HEIGHT;
NW->DetailPen = 6;       NW->BlockPen = 0;
NW->IDCMPFlags = NULL;
NW->Flags = BORDERLESS | ACTIVATE | RMBTRAP |
         NOCAREREFRESH | REPORTMOUSE;
NW->FirstGadget = NULL;
NW->CheckMark = NULL;    NW->Title = "";
NW->Screen = NULL;       NW->BitMap = NULL;
NW->MinWidth = NW->MinHeight =
NW->MaxWidth = NW->MaxHeight = 0;
NW->Type = CUSTOMSCREEN;

Spr1 = GetSprite (&Ball,2);      /* Allocate Sprites */
Spr2 = GetSprite (&BumperR,4);   /* for own use      */
Spr3 = GetSprite (&BumperL,6);

if ((Spr1 == -1) | (Spr2 == -1) | (Spr3 == -1))
    CloseIt("No Sprites !!!\n");

Ball.x = BumperL.x = BumperR.x = 0;  /* Set Position*/
Ball.y = BumperL.y = BumperR.y = 0;  /* and Height  */

Ball.height = 6;
BumperL.height = 8;
BumperR.height = 8;

if ((Screen = (struct Screen *)
          OpenScreen(&NewScreen)) == 0)
   CloseIt("No Screen !!!");

NewWindow.Screen = Screen;

if ((Window = (struct Window *)
          OpenWindow(&NewWindow)) == 0)
   CloseIt("No Window !!!");

ChipBall = (UWORD*)
    AllocMem(sizeof(Ball_Data),MEMF_CLEAR|MEMF_CHIP);
ChipBumperR = (UWORD*)
    AllocMem(sizeof(BumperR_Data),MEMF_CLEAR|MEMF_CHIP);
ChipBumperL = (UWORD*)
    AllocMem(sizeof(BumperL_Data),MEMF_CLEAR|MEMF_CHIP);

if ((ChipBall == 0) |
   (ChipBumperR == 0) |
   (ChipBumperL == 0))  CloseIt("No ChipMem !!!\n");

Help = ChipBall;
for (i=0; i<(sizeof(Ball_Data)/sizeof(UWORD));i++)
```

```
        {
           *Help = Ball_Data[i];
           Help++;
        }

     Help = ChipBumperR;
     for (i=0; i<(sizeof(BumperR_Data)/sizeof(UWORD));i++)
        {
           *Help = BumperR_Data[i];
           Help++;
        }

     Help = ChipBumperL;
     for (i=0; i<(sizeof(BumperL_Data)/sizeof(UWORD));i++)
        {
           *Help = BumperL_Data[i];
           Help++;
        }

     ChangeSprite (&Screen->ViewPort,&Ball,ChipBall);
     ChangeSprite (&Screen->ViewPort,&BumperL,ChipBumperL);
     ChangeSprite (&Screen->ViewPort,&BumperR,ChipBumperR);
                          /* Link Sprite to Data-Block */
}

/*******************************************************/
/* This Routine returns all memory used by the Program */
/* (Window,Screen, Sprites...).                        */
/*-----------------------------------------------------*/
/* Entry-Parameters: Errormessage (String)             */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/

VOID CloseIt(s)
char *s;
{
   puts(s);
   if (Spr1 != -1) FreeSprite(Spr1);/* Sprites released*/
   if (Spr2 != -1) FreeSprite(Spr2);  /* to System     */
   if (Spr3 != -1) FreeSprite(Spr3);

   if (NewPointer != 0) FreeMem(NewPointer,
                              sizeof(UWORD)*2);

   if (ChipBall != 0) FreeMem(ChipBall,sizeof(Ball_Data));
   if (ChipBumperR != 0)
       FreeMem(ChipBumperR,sizeof(BumperR_Data));
   if (ChipBumperL != 0)
       FreeMem(ChipBumperL,sizeof(BumperL_Data));

   if (Window) CloseWindow(Window);   /* Close Window  */

   if (Screen) CloseScreen(Screen);   /* Close Screen  */
   if (GfxBase) CloseLibrary(GfxBase);/* Close Librarys*/
   if (IntuitionBase) CloseLibrary(IntuitionBase);
   exit (0);                                /* Bye !!! */
}

/*******************************************************/
/* This Function opens the required Librarys, and calls*/
/* InitScreen()                                        */
```

```
/*-------------------------------------------------------*/
/* Entry-Parameters: None                                */
/*-------------------------------------------------------*/
/* Returned-Values: None                                 */
/*******************************************************/
VOID OpenLibs()
{
   if ((GfxBase = (struct GfxBase *)
            OpenLibrary("graphics.library",0)) == 0)
      CloseIt("No Graphics !!!");

   if ((IntuitionBase = (struct IntuitionBase *)
            OpenLibrary("intuition.library",0)) == 0)
      CloseIt("No Intuition !!!");

   InitScreenWindow(&NewScreen,&NewWindow);
}

/*******************************************************/
/* This Function deletes a hit Brick from the Screen   */
/* and raises the point standing.                      */
/*-----------------------------------------------------*/
/* Entry-Parameters:  x,y Position of Ball, Play Level.*/
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/
VOID DelBox(x,y,level)
long x,y,level;
{
   if ((x != 0) && (y < 17*8) && (x <= WIDTH -16)
                        && (y > 16))
                              /* Ball in valid area ? */
      {
         x /= 19;          /* Calculate Brick-Position */
         y /= 8;

         SetAPen (RP,0);                    /* Erase ? */
         RectFill (RP,x*19+1,y*8+1,
                (x+1)*19-1,(y+1)*8-1);

         score += (16-y)*10;          /* Raise Score ? */
         count ++;                /* Another Brick Hit */
         Score (score,level,balls);
                              /* Display New Score (also */
                              /*  go to new level )      */
         if (count == 13*12) New = TRUE;
                                       /* New Level ? */
      }
}

/*******************************************************/
/* This Function shows the Point standing, the Play    */
/* level and the Number of Balls left                  */
/*-----------------------------------------------------*/
/* Entry-Parameters:  score (Point stand),             */
/*                    level (Play level) ,             */
/*                    balls (Balls left)               */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/
VOID Score(score,level,balls)
long score,level,balls;
```

```
{
   SetDrMd(RP,JAM2);
   SetAPen (RP,2);
   Move (RP,0,RP->TxBaseline);
   Text (RP,"Level:  ",8);
   itoa (level,2);
   Text (RP,"  Score: ",9);
   itoa (score,10);
   Text (RP,"  Balls: ",9);
   itoa (balls,2);
   SetDrMd(RP,JAM1);
}

/*******************************************************/
/* This Function converts an Integer-Value to ASCII,   */
/* and displays the String                             */
/*-----------------------------------------------------*/
/* Entry-Parameters: Number to Convert, how            */
/*                   many positions for this Number?   */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/
VOID itoa(num,places)
long num,places;
{
   long saver;

   saver = places;
   places --;      /* Position value reduced by one    */
                   /* because of +- 1 Error            */
   ASCString[0] = 32;    /* Zero Character equals  ' ' */

                                 /* Convert Integer */
   do {
         ASCString[places] = (num % 10) + '0';
         places --;
         num /= 10;
      } while (places > 0);
   Text (RP,ASCString,saver);          /* Display Text */
}
/*******************************************************/
/* This Function waits for a Mouse Click               */
/*-----------------------------------------------------*/
/* Entry-Parameters: None                              */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/
VOID AwaitClick()
{
   long MouseX;
   char *LeftMouse = (char *)0xBFE001;

   while ((*LeftMouse & 0x40) == 0x40)
      {
        MouseX = Screen->MouseX;

        MoveSprite (&Screen->ViewPort,&BumperL,
                          MouseX-16,192);
        MoveSprite (&Screen->ViewPort,&BumperR,
                          MouseX,192);
      }
}
```

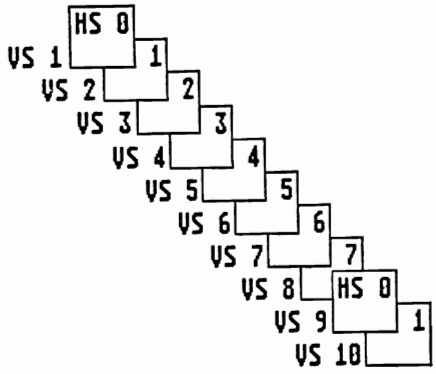
18. The Amiga animation system

Now that you have been introduced to the hardware sprites, we will present the vsprites and bobs. These graphic elements (GELs) are small graphic images, which are similar to the hardware sprites but are controlled differently.

18.1 The vsprites

With the help of the hardware sprites and some creative Copper programming, we can use the vsprites (the V stands for virtual = visible).

To display a vsprite we use a hardware sprite once and then again at a lower location:



Before the electronic beam can reach the bottom of the screen, the same hardware sprite is displayed again. By repeating this procedure, we can display many vsprites.

However, there are some limitations when using vsprites. You have to allow a short pause for the electronic beam between each use of a hardware sprite. This pause must equal at least one raster row. The DMA, which is responsible for displaying the hardware sprites, is very fast but not as fast as the electronic beam. Also, it is not possible to use a single hardware sprite to display several vsprites starting in the same row.

A maximum of eight vsprites in one row, one for each of the eight hardware sprites, can be used. An additional limitation occurs when you use one of the eight sprites yourself. When you use `GetSprite` to gain control of a sprite for your use, you must remove two sprites from vsprite use.

Each vsprite can use different colors. Just before the Copper displays the vsprite, you can change the color values in the color registers. Remember that hardware sprites use the color registers in pairs. So changing the colors of one hardware sprite will also affect the other sprite in the pair.

To prevent these color problems, test whether the vsprites are going to be displayed in different colors by using the pointer `GelsInfo.lastcolor`. If this test is positive (true), the system will no longer allow the use of both hardware sprites in a pair at the same time.

To clearly explain this, we have included a short example:

The Intuition mouse pointer is displayed with hardware sprite zero. Hardware sprites zero and one use the same color registers and colors. In our example, we will use hardware sprite one for vsprites and we will make it use different colors than the mouse pointer. However, when the vsprite is displayed, the mouse pointer would immediately change colors to match the vsprite colors set by hardware sprite one. To prevent this from happening, the system doesn't allow the use of hardware sprite one for vsprites.

You can determine which sprites are available for vsprite use in the variable `GelsInfo.sprRsvrd`. Each set bit in this variable determines if the corresponding hardware sprite is available for the vsprite software. This variable has a different meaning than `Gfxbase->SpriteReserved`. Normally this variable contains information about your assigned sprites. When Intuition bit zero is set (= 0x01), it means that sprite zero is reserved for Intuition and cannot be used for vsprites. To duplicate this, you must write a value of `0xfe` in `GelsInfo.sprRsrvd`. This makes all hardware sprites, except zero, available for displaying vsprites.

When using vsprites of different colors, you should not use hardware sprite pairs.

Since the Copper is not capable of making the needed changes fast enough, it isn't able to display a hardware sprite pair as different colored vsprites on the same raster row.

This limits you to a maximum of four vsprites, of four different colors, on the same raster row.

Just as the hardware sprites have the `SimpleSprite` structure, the vsprites also have a structure named `VSprite` that helps you initialize the vsprites. This structure contains all the required vsprite data.

The first step in any program that uses vsprites is to create this structure for each vsprite (`struct VSprite VSprite;`).

You must also initialize two additional vsprite structures that mark the beginning and end of the GEL list. Without a GEL list nothing happens and vsprites or bobs won't work.

To build a GEL list we use another structure called the `GelsInfo` structure. You must provide the pointers `nextline`, `lastcolor` and the variable `sprRsvd` with the necessary values (for example, memory area pointers). Without this information, nothing will happen on your screen. Appendix A has a complete description of the `GelsInfo` structure.

After all the variables and pointers are initialized, you can initialize the `GelsInfo` structure with `InitGels (&StartVSprite, &EndVSprite, &GelsInfo)`. You provide `GelsInfo.gelHead` and `GelsInfo.gelTail` with the addresses of both specified vsprite structure).

These two vsprites mark the beginning and end of the GEL list. To add any additional GELs (vsprites and bobs) to this list, use `AddVSprite` and `AddBob`. Then you use the commands `SortGList` and `DrawGList` to organize the GEL list for display purposes.

You must also specify the appearance of your vsprites. The pointer to the data that contains the vsprites' image (`VSprite.ImageData`) is used for this. This data is organized exactly like the hardware sprites `Appearance[Height][2]`. The vsprite row data is provided in two `UWORD`s. Unlike the hardware sprite, no additional memory areas are required (this is handled by the vsprite structure and the GEL list).

The vsprite height is set in the variable `VSprite.Height`. Use the variables `VSprite.X` and `VSprite.Y` to position your vsprite. Once again, you must consider the resolution difference between normal, hi-res and lace. Change your X and Y coordinates by a factor of two, depending on which mode you are using. If your coordinates are off, the movement may be very jerky-looking.

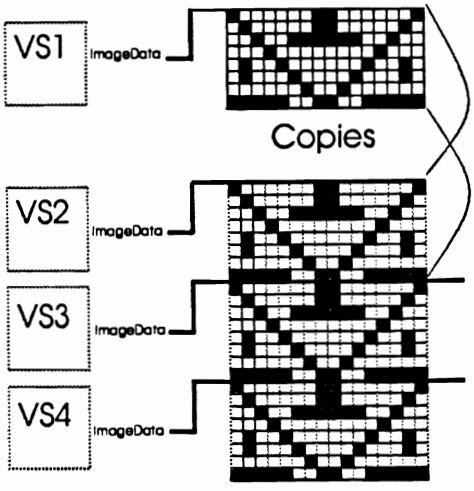
As we previously mentioned, each vsprite can use different colors. These colors are set with the help of the pointer

`VSprite.SprColors`, which references a three dimensional `UWORD` array that contains the colors. You format this array like the color registers (bits 11-8 contain red, bits 7-4 the green and bits 3-0 the blue color components). Remember, placing many vsprites, that contain many colors, close to each other, limits your possibilities.

Due to the hardware sprite pairing that helps us display the vsprites in different colors, some problems can occur. It is possible to display too many different colored vsprites on one raster row (Y position), which can cause some vsprites to disappear. This happens because the Amiga isn't able to quickly switch between the many different color definitions.

To avoid this problem, display all your vsprites with the same colors. The Amiga will then ensure that all the `SprColor` pointers for the vsprites point to the same area. In order to test this, you must first make sure that `GelsInfo.lastcolor` has enough memory assigned. This is where the pointer to the last color definition for an individual vsprite is stored. The color definition for a new vsprite is then compared to this value.

If all of your vsprites have the same shape, don't point all the `ImageData` pointers to the same address. The memory access timing of the Amiga doesn't allow several vsprites to use the same `ImageData` pointer. Instead, copy the image to a memory area and point the `ImageData` pointer to the different starting addresses of the data.



Many VSprites with
the same appearance

You still have to set the flags variable for vsprites to `VSPRITE` (`VSprite.Flags = VSPRITE`). This informs the system that

you are using vsprites, not bobs. Later we will explain how you can also use vsprite structures for bobs.

Even though it isn't possible to create vsprites with a width greater than 16 pixels, or one UWORD, you still have to set their width. Use a value of one with `VSprite.Width`. If you forget to do this, your vsprite may not appear on the screen.

Although a different value isn't possible, you still must set the depth of your vsprite (`VSprite Depth`) to a value of two. This ensures that the system knows exactly what you want. Vsprites cannot be attached to each other like hardware sprites.

When all the parameters are set, you can join the vsprite to your RastPort with `AddVSprite (&VSprite, &RastPort)`. This links your vsprite to the RastPort that was specified earlier with `RastPort GelsInfo = &GelsInfo`. Now we can finally display the vsprite.

First you have to sort the GEL list with `SortGList (&RastPort)`. All the vsprites in the GEL list are sorted by their Y coordinates (ascending) and X coordinates (ascending by Y coordinate). This makes it easier for the Copper to display as many vsprites as possible in the least amount of time.

Next you can generate a new Copper list with `DrawGList (&RastPort, &ViewPort)`; for the specified ViewPort. This Copper list will control everything that is needed for displaying vsprites. To execute the new Copper list use `MakeVport`, `MrgCop` and `LoadView`, `RemakeDisplay` for Intuition screens, and your vsprites appear on the screen.

These are all the necessary steps for displaying one or more vsprites.

To change the position, appearance or colors of a vsprite, change the corresponding pointers and variables. Then sort the GEL list again (`SortGList`) and draw again (`DrawGList`, `MakeVPort`, `MrgCop`, and `LoadView`).

18.1.1 Vsprites and collisions

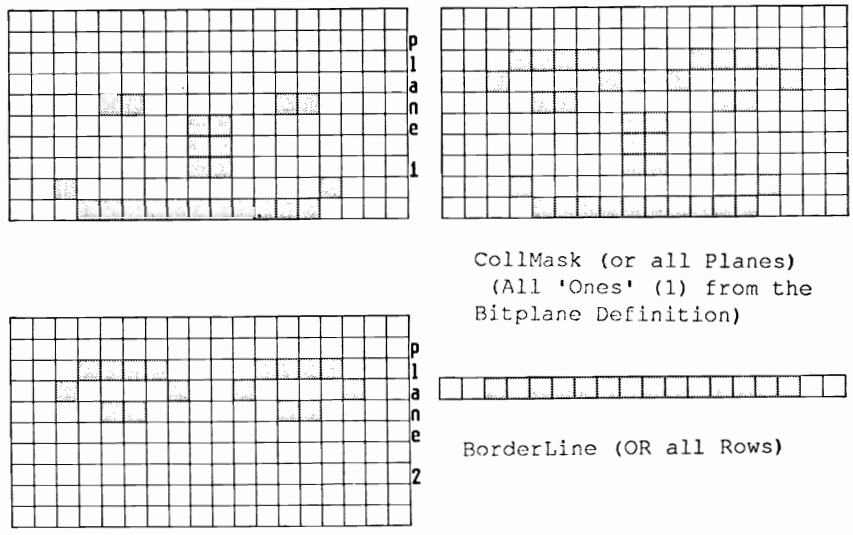
The above procedures provide information on how to display and move vsprites. However, you cannot use these procedures to check for collisions, which is very important, for example, in games.

Two pointers in the vsprite structure, `BorderLine` and `CollMask`, help us check for collisions. The variables `VSprite.MeMask` and `VSprite.HitMask` and the pointer `CollTable` from the `GelsInfo` structure (These must be initialized whether you use collisions or not) are also needed.

We will discuss `BorderLine` and `CollMask` first. These pointers point at two memory areas where you have stored data.

After calling `Initmasks`, `Borderline` contains the logical `OR` of all rows for your vsprite and requires a single `UWORD` pointer. `BorderLine` is used for very fast collision control. When you use `BorderLine` to determine whether a collision is possible, the `CollMask` is used to discover whether a collision occurred.

The `CollMask` contains the logical `OR` for both `UWORD`s of each vsprite row. You can use a one dimensional `UWORD` array that must contain the `VSprite.Height` element.



After providing the required memory for both pointers, you can use `InitMasks(&VSprite)` to initialize both collision masks. `BorderLine` and `CollMask` are now initialized for the specified vsprite (bob). You should have already pointed `ImageData` to the correct memory area because this is where `CollMask` and `Borderline` are calculated from.

Once both masks are correctly initialized, you can set the variables `MeMask` and `Hitmask` for each vsprite to determine which collisions will be registered. The routine `DoCollision (&RastPort)` not only tests whether a collision has occurred, but also decides if another

action should be performed. This routine tests all vsprites in the GEL list and then determines if the collision requires any action. For example, whether or not to execute another routine.

Suppose you wanted to program the game "PacMan" by using vsprites. For ghost collisions there shouldn't be a reaction. But when a ghost collides with PacMan, you should lose the PacMan.

You can determine the actions of the `DoCollision` routine by using the variables `MeMask` and `HitMask`. We set bit one in the ghost `MeMask`, which means: "I am a ghost". For the PacMan vsprite we only set bit two which means, "I am PacMan".

Now we set bit two in the ghost `HitMask`. This means that the ghost can collide with another ghost without causing anything to happen. However, a collision with a PacMan will cause a reaction.

Finally we set bit one in the PacMan's `HitMask`. This means that any collision between a PacMan and a ghost will cause the PacMan to disappear.

The `MeMask` and `HitMask` for both look like this:

```
Ghost:      MeMask      %0000000000000010
            HitMask     %0000000000000100

PacMan:     MeMask      %0000000000000100
            HitMask     %0000000000000010
```

When `DoCollision` registers a collision, the `MeMask` of the vsprite, located above and left, is `AND`ed with the `HitMask` of the other vsprite. The resulting bit of this logical `AND` identifies the routine that `DoCollision` executes for this collision.

For example, when two ghosts collide, `DoCollision` performs the following routine:

```
Ghost1.MeMask AND Ghost2.HitMask = %10 AND %100 = %000
```
(or no collision)

When a ghost and a PacMan collide:

```
Ghost.MeMask AND PacMan.HitMask = %10 AND %10 = %10
```
(here routine one is called)

When a PacMan and a ghost collide it means that the PacMan is above and left of the ghost (above was the opposite).

```
PacMan.MeMask AND Ghost.HitMask = %100 AND %100 = %100
```
(here routine two is executed)

To determine the collision routines, you first have to initialize the pointer `CollTable` in the `GelsInfo` structure. (For this you use the structure `CollTable` which contains the pointers to the collision routines).

Of course, the collision routines themselves must exist. Use `SetCollision (CollisionNumber, Routine, &GelsInfo)` to specify which routines you want for each collision. When the result bit of the logical `AND` of `MeMask` and `HitMask` determine that a collision has occurred, your program routines are executed. You write the addresses for these routines into the `GelsInfo.CollTable`.

Make sure that your collision routine provides two parameters for `DoCollision`. You must provide the addresses of both vsprites in the collision. The first address parameter is for the vsprite that is to the left and above the other vsprite:

```
Routine (&VSprite1, &VSprite2)
struct VSprite *VSprite1, *VSprite2;
{...}
```

This is how you test for collisions between vsprites (and bobs). To test for collisions with the border, first set the variables `topmost`, `bottommost`, `leftmost`, and `rightmost` in the `GelsInfo` structure. These variables determine the limits of a bit-map rectangle within which the vsprites can move without registering a collision.

Collision routine number zero is executed when you set bit zero (reserved for border collisions) in a vsprites `HitMask` and it touches a border. You set up this routine with `SetCollision (0, Routine, &GelsInfo);`. It has the following parameters:

```
BorderControl (VSprite, Flags)
struct VSprite *VSprite;
BYTE Flags;
{...}
```

`VSprite` is the vsprite that has a collision with the border. Flags contains the border where the collision occurred (TOPHIT, BOTTOMHIT, LEFTHIT, RIGHTHIT).

The vsprite structure does not support collisions with the background. You have to use the `ReadPixel` method for vsprite to background collisions.

The following program demonstrates how vsprites are created and tested for collisions:

```
/*******************************************************/
/*                      Tribars.c                      */
/*                                                     */
/* VSprites pur.                                       */
/*                                                     */
/* Compiled with: Lattice V5                           */
/*                                                     */
/* (c) Bruno Jennrich                                  */
/*******************************************************/

struct VSpriteExt {              /* Our Data is linked */
                    int vx,vy;   /* with the VSprite   */
                  };             /* Structure          */

#define VUserStuff struct VSpriteExt
                       /* Must happen before #INCLUDES!*/

#include "exec/types.h"
#include "exec/memory.h"
#include "exec/devices.h"
#include "devices/keymap.h"
#include "graphics/gfx.h"
#include "graphics/gfxmacros.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
#include "graphics/collide.h"
#include "graphics/gfxbase.h"
#include "graphics/regions.h"
#include "hardware/blit.h"
#include "hardware/custom.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
#include "libraries/diskfont.h"
#include "hardware/dmabits.h"

#define RP Screen->RastPort          /* Access RastPort */

#define MAXVSPRITES 16           /* How many VSprites ? */
struct GfxBase *GfxBase;                 /* BasePointer */
struct IntuitionBase *IntuitionBase;

struct NewScreen NewScreen =              /* Our Screen */
                      {
                        0,0,320,199,1,
                        1,0,
                        0,
                        CUSTOMSCREEN,
                        NULL,
                        "",
                        NULL,NULL
                      };
struct Screen *Screen;
```

```
struct VSprite Start, Ende,       /* VSprites for GEL */

              VSprite[MAXVSPRITES];  /* List          */

UWORD VSBorderLine[MAXVSPRITES];/* VSprites Borderline */

UWORD VSCols[3] =
        {0xfff,0x057,0x889};       /* Colors for all  */

struct GelsInfo GelsInfo;     /* GelsInfo must be       */
                              /* completely initialized */
                              /* prior to any use of    */
                              /* VSprites !             */

WORD Nextlines[8] = {0,0,0,0,0,0,0,0};
                           /* Nextline-Array for GInfo */

WORD *lastColors[8] = {0,0,0,0,0,0,0,0};
                          /* lastColor-Array for GInfo */

struct collTable collTable;
                               /* Collisions-Jump Table */
UWORD Tribar[18][2] =
            {
                 {0xe000,0xc000},    /* How do the     */
                 {0xf800,0xc000},    /* VSprites look? */
                 {0xfe00,0xc000},
                 {0xff80,0xc000},
                 {0xf3e0,0xcc00},
                 {0xf0f8,0xcf00},
                 {0xf03e,0xc7c0},
                 {0xf00f,0xc1f0},
                 {0xf043,0xc07c},
                 {0xf0f0,0xc0ff},
                 {0xf3e0,0xc3ff},
                 {0xff80,0xcffc},
                 {0xfe00,0xfff0},
                 {0xf800,0xffc0},
                 {0xe000,0xff00},
                 {0x8000,0xfc00},
                 {0x0000,0xf000},
                 {0x0000,0xc000}
             };

VOID BordControler();             /* Our Routine for   */
                                  /* Border Collisions */

/*******************************************************/
/* Here we Go !                                        */
/*******************************************************/

main()
{
   long i,j,k;
   int Length;
```

```
    UWORD *Tribs, *HelpTrib,*VSCollMask,*HelpColl;
    char   *LeftMouse = (char *) 0xBFE001;
                                    /* Left Mouse Button */

    if ((GfxBase = (struct GfxBase *)
             OpenLibrary("graphics.library",0)) == NULL)
                {
                   printf (" No Graphics!\n");
                   exit(0);
                }

    if ((IntuitionBase = (struct IntuitionBase *)
             OpenLibrary("intuition.library",0)) == NULL)
                {
                   printf ("No Intuition!\n");
                   goto cleanup1;
                }

    if ((Screen = (struct Screen *)
            OpenScreen (&NewScreen)) == NULL)
              {
                 printf ("No Screen!\n");
                 goto cleanup2;
              }

    Tribs = (UWORD *)AllocMem

  (18*2*sizeof(UWORD)*MAXVSPRITES,MEMF_CLEAR|MEMF_CHIP);
                          /* Even though all VSprites    */
                          /* have the same shape you must*/
                          /* reserve a 'Chip' Buffer for */
                          /* each VSprite or DMA can     */
                          /* lose track of them.         */
    if (Tribs == 0)
       {
          printf (" No Memory for Tribs !!!\n");
          goto cleanup3;
       }

    HelpTrib = Tribs;

    for (i=0; i<MAXVSPRITES; i++)
       for (j=0; j<18; j++)
          for (k=0; k<2; k++)
             {
                *HelpTrib = Tribar[j][k];
                HelpTrib++;
             }

    VSCollMask = (UWORD *)
       AllocMem(18*sizeof(UWORD)*MAXVSPRITES,MEMF_CHIP);

    if (VSCollMask == 0)
       {
          printf (" No Memory for CollMask !!!\n");
          goto cleanup4;
```

```
    }


    SetRGB4 (&Screen->ViewPort,0,0,0,0);
    SetRGB4 (&Screen->ViewPort,1,13,7,1);
                                        /* A little Color */
    SetRast (&RP,0);

    Length = TextLength (&RP,"Tribars in Action !!!",21);
    Move (&RP,320/2-Length/2,100);
    Text (&RP,"Tribars in Action !!!",21);

    BltClear (&Start, sizeof(struct VSprite),0);
    BltClear (&Ende, sizeof(struct VSprite),0);
    BltClear (&GelsInfo, sizeof(struct GelsInfo),0);
    BltClear (VSprite, sizeof(struct
VSprite)*MAXVSPRITES,0);
    BltClear (&collTable, sizeof (collTable),0);
                        /* to be sure, clear everything */

    GelsInfo.sprRsrvd = 0xfc;
                                /* All Sprites except   */
                                /* 0 and 1 for VSprites */

    GelsInfo.nextLine = Nextlines;
    GelsInfo.lastColor = lastColors;
    GelsInfo.collHandler = &collTable;

    GelsInfo.leftmost = 23;         /* Limits for Border */
    GelsInfo.rightmost = 300;       /* Collisions        */
    GelsInfo.topmost = 9;
    GelsInfo.bottommost = 191;

    InitGels (&Start, &Ende, &GelsInfo);
                            /* Initialize GelsInfo and   */
    RP.GelsInfo = &GelsInfo;  /* link with RastPort      */

    SetCollision(0,BordControler,&GelsInfo);

    HelpTrib = Tribs;
    HelpColl = VSCollMask;
    for (i=0; i<MAXVSPRITES; i++)
       {
          VSprite[i].Width = 1;          /* 1 WORD width */
          VSprite[i].Height = 18;        /* 18 Rows high */
          VSprite[i].Flags = VSPRITE;
                                    /*VSprite is VSprite*/
          VSprite[i].Depth = 2;           /* 2 'Planes' */
          VSprite[i].ImageData = HelpTrib;/* own Tribar */
          VSprite[i].MeMask = 0;     /* no GEL Collision */
          VSprite[i].HitMask = 1;    /* but with Border  */

          VSprite[i].CollMask= HelpColl;
          VSprite[i].BorderLine = &VSBorderLine[i];

          VSprite[i].SprColors = VSCols;       /* Colors */
```

```
            VSprite[i].X = 23+i*(320/MAXVSPRITES-4);
            VSprite[i].Y = 9+i*(200/MAXVSPRITES-2);
                                              /* Position */

            VSprite[i].VUserExt.vx = 5;      /* individual */
            VSprite[i].VUserExt.vy = 5;      /* Speed      */

            HelpTrib += 2*18;
            HelpColl += 18;
                                          /* next Tribar */

            InitMasks (&VSprite[i]);/* Calculate CollMask */
                                    /* and Borderline     */

            AddVSprite (&VSprite[i], &RP);
                                 /* Sort VSprite In List */
         }

      while ((*LeftMouse & 0x40) == 0x40)
         {
            for (i=0; i<MAXVSPRITES; i++)
               {
                  VSprite[i].Y +=
                        VSprite[i].VUserExt.vy;
                  VSprite[i].X +=
                        VSprite[i].VUserExt.vx;
                                       /* move VSprites */
               }
            SortGList(&RP);                    /* new Sort */
            DoCollision(&RP);            /* Collision test */
            DrawGList(&RP, &Screen->ViewPort);
                                 /* Generate Copper-List */
            WaitTOF();
            RemakeDisplay();                /* and use it */
         }


            FreeMem(VSCollMask,
                      18*sizeof(UWORD)*MAXVSPRITES);
      cleanup4:   FreeMem(Tribs,
                      18*2*sizeof(UWORD)*MAXVSPRITES);
                                    /* Release Memory */
      cleanup3:   CloseScreen(Screen);
      cleanup2:   CloseLibrary(IntuitionBase);
      cleanup1:   CloseLibrary(GfxBase);
      return(0);
}
/*******************************************************/
/* This Routine is executed by DoCollision() when a    */
/* VSprite collides with a Border                      */
/*-----------------------------------------------------*/
/* Entry-Parameters:  VSprite, that collided with a    */
/*                    Border - and which Border        */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/
```

```
VOID BordControler (VSprite, Border)
struct VSprite *VSprite;
BYTE Border;
{
   if ((Border & TOPHIT)   == TOPHIT)          /* Top    */
                VSprite->VUserExt.vy *=-1;
   if ((Border & BOTTOMHIT)  == BOTTOMHIT)     /* Bottom */
                VSprite->VUserExt.vy *=-1;
   if ((Border & LEFTHIT)   == LEFTHIT)        /* Left   */
                VSprite->VUserExt.vx *=-1;
   if ((Border & RIGHTHIT)   == RIGHTHIT)      /* Right  */
                VSprite->VUserExt.vx *=-1;
}
```

We have to inform you about a small problem with collision control. `DoCollision`, which tests all GELs in the GEL list for collision, doesn't always work perfectly. It is possible for a collision to occur but not be detected.

The following is some information about an additional vsprite flag, `GELGONE`. This flag affects not only vsprites, but also all collision controls for bobs. When the system sets this flag (in `VSprite.Flags`) your GEL (vsprite or bob) completely disappears from the area specified in your `GelsInfo` structure.

Once you determine that this flag is set, use `RemVSprite` to remove your vsprite from the GEL list (the next use of `DrawGList` does not have to calculate for it). To do the same for bobs use `RemIBob` or the macro `RemBob`, which we will discuss in the next section.

18.2 Another GEL: the bob

Just like vsprites, bobs (Blitter objects) are also graphic elements (GELS). To display bobs, an initialized `GelsInfo` structure linked with a RastPort is required. With other types of computers these graphic elements are called "shapes". However, with the Amiga these elements are called bobs (Blitter objects) because they are controlled through the Blitter.

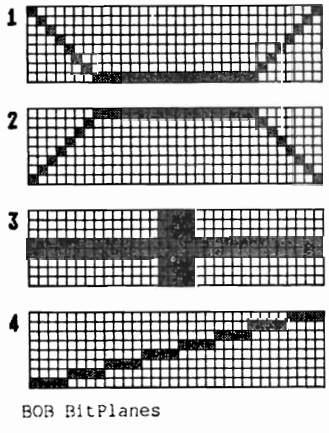
Unlike vsprites, these rectangular graphic objects can be any size you want. The only limitation to their size is that the width must be a multiple of 16 (16, 32, 48...).

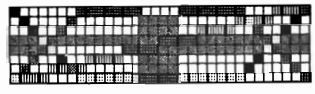
Bobs are written directly into the bit-plane memory of a RastPort. This means that bobs are automatically displayed in the same resolution and with as many possible colors as the RastPort.

Because of this, you define a bob's appearance the same way as a bit-plane. This is done by telling the Blitter which bit combinations (stored in `UWORD`s) should be written into the bit-plane memory.

Just defining the appearance of a bob in a `UWORD` array isn't enough. The Blitter also must know how large the bob is, where to position it, and how many bit-planes it has.

To accomplish this you must initialize an individual vsprite structure for each bob. However, this does not mean that bobs are displayed as vsprites. As we mentioned earlier, bobs are written directly into the bit-map.





Appearance in the Rastport

```
BOBVSprite.Width  = 2 (2 Words = 32 pixels)
BOBVSprite.Height = 8
BOBVSprite.Depth  = 4
BOB.ImageShadow   =
```

BOBVSprite.BorderLine =

The size of your bob is specified in the vsprite structure variables `BOBVSprite.Height` and `BOBVSprite.Width`. The width variable (`BOBVSprite.Width`) contains the number of words (16 bits). You must also specify the address of the first bob bit-plane in `BOBVSprite.ImageData`.

Their position is set the same way as vsprites. `BOBVSprite.X` sets the X coordinate and `BOBVSprite.Y` sets the Y coordinate for your bob.

For collision control, initialize the `MeMask` and `HitMask` of the vsprite structure. Collisions are managed much like they are for vsprites. The buffer for `Borderline` must be at least as wide as one row of the bob. The `CollMask` buffer must also be as large as one bob bit-plane. To initialize them for bobs, use `InitMasks (&BOBVSprite)`.

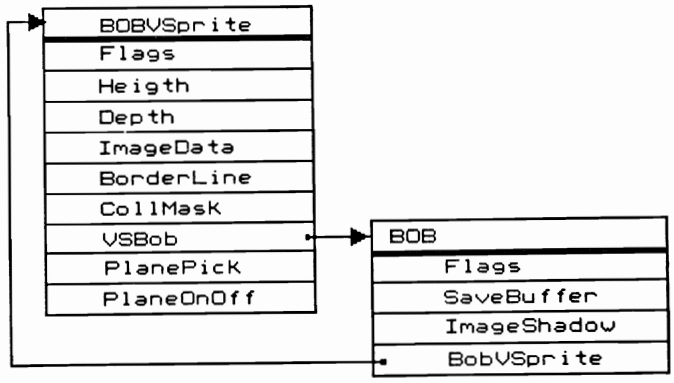
One difference between vsprites and bobs is that the vsprite flag should not be set to VSPRITE because this flag is not set when we use bobs.

Also a color definition for bobs is not required as it is with vsprites (`VSprite.SprColors`). Since bobs are written into the RastPort, they use the colors of the RastPort. We have now reached another limiting factor. If you want to use both bobs and vsprites in one RastPort you cannot use any colors from registers 17 to 31 for your bobs. In other words, do not create any bobs that are more than four bit-planes deep. For example, when displaying vsprites with changing color values, your bob could start flickering in the areas affected by the changed colors. If you need more than 16 colors you can still use color registers 16, 20, 24, and 28. These registers are used for the sprites transparent color and can also be used for vsprites.

You set the depth for your bobs the same way as the vsprites - by using `BOBVSprite.Depth`. However, with bobs this value is not constant and changes depending on how you define your bobs. Also, the more bit-planes you define for your bob, the more colors it can have.

The vsprite structure isn't the only place your bob can be described. The bob structure contains additional variables that provide more details about your bob.

You must also link the bob and corresponding vsprite structure together. First you point a pointer in the bob structure to the required vsprite structure (`Bob.BobVSprite = &VSprite`). Then you point a pointer in the vsprite structure to the bob structure (`VSprite.VSBob = Bob`). The two structures are now linked together.



The relevant variables for displaying Bobs

There is another pointer in the bob structure that is very important for color display: `Bob.ImageShadow`. `ImageShadow` is the same as the `CollMask`, a logical `OR` of all bit-plane rows of the bob.

You may be wondering why there are two pointers for the same thing. Actually, `ImageShadow` and `CollMask` are not quite the same. You can change the `CollMask` after you use `InitMasks`. For example, you could change it so that not all of the set bits of the bob's bit-plane respond to collisions. Perhaps only one bit-plane is available for collision control.

The relationship between the vsprite variables `PlanePick` and `PlaneOnOff` give `ImageShadow` a different meaning. The bits in `PlanePick` determine which bit-planes of the RastPort the bit-planes of a bob are written to. Normally `0xff` is written. This means that you copy, in order, all the bob bit-planes to all the bit-planes of the RastPort. However, a value of %00000101 = 0x05 has the following effect: You copy the first bob bit-plane to the first RastPort bit-plane and copy the second bob bit-plane to the third bit-plane of the RastPort.

The `PlaneOnOff` variable determines what happens with the inactive bit-planes of the RastPort. The default value here is 0x00. This means that nothing is written to a RastPort bit-plane that is not selected with `PlanePick`. A value of 0x2 means that `ImageShadow` is written to bit-plane two of the RastPort.

When all the variables and pointers of the vsprite structure are initialized you are almost ready to initialize the `GelsInfo` structure.

The bobs provide many more possibilities than vsprites because they are actually designed for software control. You can program the functions of the Blitter thru the 68000, but these routines are not as fast.

For example, you could set the SAVEBOB flag in the flags variables of the bob structure. This causes the bob to act like a brush, which draws over the background. However, the background is not restored after the bob moves over it.

To restore the background after the bob moves, set the SAVEBACK flag, which is located in the VSprite.Flags variables. In addition, you must reserve enough memory for as many bob bit-planes that will be written to the RastPort (please note the uses of PlanePick and PlaneOnOff). The bob software restores the background and you provide the background memory address to Bob.SaveBuffer.

The OVERLAY flag, set in the vsprite structure, is used to prevent the unset bits of the bob from being written to the RastPort. These bits allow the original background to show through. The bob is actually ORed with the bit-planes of the RastPort. It is important that, prior to this, you have initialized ImageShadow for use with the OVERLAYs use Bob.ImageShadow = BOBVSprite.Collmask after using InitMasks).

All of the flags we have discused up until now have been involved in the displaying of bobs. However, there are other flags that aren't associated with displaying bobs. When you determine that a bob is gone, this means that the GELGONE flag in VSprite.Flags is set. You can use the macro RemBob (&Bob) to set the BOBSAWAY flag.

When the system knows that this flag is set, the next DrawGList call will not draw the bob. If you want the bob to immediately disappear, call RemIBob (the I stands for immediate) (&Bob, &RastPort, &ViewPort). This removes the bob from both the GEL list and the screen.

You now know the complete procedure for displaying bobs. To design the bobs in the RastPort, use DrawGList, which not only prepares the vsprites for display but also designs the bobs in the RastPort. However, you must sort the GEL list (SortGList) before using DrawGList.

The following program demonstrates how to create, move and test bobs for collisions:

```
/*****************************************************/
/*                      Bobs.c                       */
/*                                                   */
/* Bobs test                                         */
/*                                                   */
/* Compiled with: Lattice V5                         */
/*                                                   */
/* (c) Bruno Jennrich                                */
/*****************************************************/
```

```
struct VSpriteExt {          /* Our own Data that is */
                   int vx,vy;  /* linked with the   */
                  };           /* VSprite structure */

#define VUserStuff struct VSpriteExt
                   /* Must happen before #INCLUDES ! */

#include "exec/types.h"
#include "exec/memory.h"
#include "exec/devices.h"
#include "devices/keymap.h"
#include "graphics/gfx.h"
#include "graphics/gfxmacros.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
#include "graphics/collide.h"
#include "graphics/gfxbase.h"
#include "graphics/regions.h"
#include "hardware/blit.h"
#include "hardware/custom.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
#include "libraries/diskfont.h"
#include "hardware/dmabits.h"

#define RP Screen->RastPort      /* Access to RastPort */

#define MAXBOBS 6                  /* How many Bobs ? */
struct GfxBase *GfxBase;               /* BasePointer */
struct IntuitionBase *IntuitionBase;

UWORD *SaveBuffer;
UWORD *DBufBuffer;

struct NewScreen NewScreen =              /* Our Screen */
                       {
                         0,0,320,199,3,
                         1,0,
                         0,
                         CUSTOMSCREEN | CUSTOMBITMAP,
                         NULL,
                         "",
                         NULL,NULL
                       };
struct Screen *Screen;

struct BitMap BitMap [2];

struct VSprite Start, Ende,       /* VSprites for GEL */
               BobVSprite[MAXBOBS];   /* List          */

struct Bob Bob[MAXBOBS];
UWORD *CollMask;            /* for Chip-Mem allocation */

UWORD BorderLine[MAXBOBS][2];      /*  Bobs Borderline */

struct GelsInfo GelsInfo;  /* GelsInfo must be         */
                           /* completely initialized   */
                           /* before using VSprites!   */

WORD Nextlines[8] = {0,0,0,0,0,0,0,0};
```

```
                                /* Nextline-Array for GInfo */

WORD *lastColors[8] = {0,0,0,0,0,0,0,0};
                               /* lastColor-Array for GInfo */

struct collTable collTable;
                                  /* Collisions-Jump Table */

struct DBufPacket *DBufPackets,*HelpPack;/*for Chip-Mem*/

UWORD *Image,*Help;
UWORD Baloon[46*2*2] =          /* 46 Rows of 2 WORDs   */
            {                   /* and everything twice */

                0x0007,0xf000,          /* Bob-Plane 1 */
                0x0019,0x6400,
                0x0073,0x3300,
                0x00e3,0x3880,
                0x01c7,0x1c40,
                0x03c7,0x1c20,
                0x0387,0x1e20,
                0x0787,0x1e10,
                0x078f,0x0e10,
                0x0f8f,0x0e08,
                0x0f0f,0x0f08,
                0x0f0f,0x0f08,
                0x0f0f,0x0f08,
                0x0f0f,0x0f08,
                0x0f0f,0x0f08,
                0x0f0f,0x0f08,
                0x0f0f,0x0f08,
                0x0f0f,0x0f08,
                0x070f,0x0f10,
                0x070f,0x0f10,
                0x0307,0x1f20,
                0x0307,0x1f20,
                0x0187,0x1e40,
                0x0187,0x1e40,
                0x0087,0x1e80,
                0x0087,0x1e80,
                0x0043,0x1d00,
                0x0043,0x3d00,
                0x0023,0x3e00,
                0x0023,0x3a00,
                0x0003,0x3800,
                0x001f,0xfc00,
                0x000f,0xf800,
                0x000f,0xf800,
                0x0007,0xf000,
                0x0004,0x1000,
                0x0004,0x1000,
                0x0006,0x3000,
                0x0002,0xa000,
                0x0002,0xa000,
                0x0003,0xe000,
                0x0003,0xe000,
                0x0007,0xf000,
                0x0007,0xf000,
                0x0007,0xf000,
                0x0007,0xf000,
```

```
                        0x0000,0x0000,        /* Bob-Plane 2 */
                        0x0006,0x9800,
                        0x000c,0xcc00,
                        0x001c,0xc700,
                        0x0038,0xe380,
                        0x0038,0xe3c0,
                        0x0078,0xe1c0,
                        0x0078,0xe1e0,
                        0x0070,0xf1e0,
                        0x0070,0xf1e0,
                        0x00f0,0xf0f0,
                        0x00f0,0xf0f0,
                        0x00f0,0xf0f0,
                        0x00f0,0xf0f0,
                        0x00f0,0xf0f0,
                        0x00f0,0xf0f0,
                        0x00f0,0xf0f0,
                        0x00f0,0xf0e0,
                        0x00f0,0xf0e0,
                        0x00f8,0xe0c0,
                        0x00f8,0xe0c0,
                        0x00f8,0xe180,
                        0x0078,0xe180,
                        0x0078,0xe100,
                        0x0078,0xe100,
                        0x003c,0xe200,
                        0x003c,0xe200,
                        0x001c,0xc000,
                        0x001c,0xc400,
                        0x001c,0xc400,
                        0x0000,0x0000,
                        0x0000,0x0000,
                        0x0000,0x0000,
                        0x0000,0x0000,
                        0x0004,0x1000,
                        0x0004,0x1000,
                        0x0006,0x3000,
                        0x0002,0xa000,
                        0x0002,0xa000,
                        0x0003,0xe000,
                        0x0003,0xe000,
                        0x0007,0xf000,
                        0x0007,0xf000,
                        0x0007,0xf000,
                        0x0007,0xf000,
                    };

VOID BackController();       /* Our Routine that is    */
                             /* executed for Collisions */
                             /* with the Border         */

VOID GelCol();              /* Routine, executed for    */

                            /* Gel-Gel collisions.      */

/*******************************************************/
/* Here we go  !                                       */
/*******************************************************/

main()
{
```

```
long i,j;
int Length;
long toggle;
char   *LeftMouse = (char *) 0xBFE001;
                                /* Left Mouse Button */

if ((GfxBase = (struct GfxBase *)
        OpenLibrary("graphics.library",0)) == NULL)
           {
              printf (" No Graphics!\n");
              exit(0);
           }

if ((IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library",0)) == NULL)
           {
              printf ("No Intuition!\n");
              goto cleanup1;
           }

InitBitMap (&BitMap[0], NewScreen.Depth,
             NewScreen.Width, NewScreen.Height);

InitBitMap (&BitMap[1], NewScreen.Depth,
             NewScreen.Width, NewScreen.Height);

for (i=0; i<2; i++)
      for (j=0; j<NewScreen.Depth; j++)
         {
            BitMap[i].Planes[j] = (PLANEPTR) AllocRaster
                (NewScreen.Width, NewScreen.Height);
            if ((BitMap[i].Planes[j]) == NULL)
               {
                  printf (" No Space for BitMaps\n");
                  goto cleanup2;
               }
            else BltClear (BitMap[i].Planes[j],
            RASSIZE(NewScreen.Width, NewScreen.Height),0);
         }


NewScreen.CustomBitMap = &BitMap[0];

Screen = (struct Screen *) OpenScreen (&NewScreen);
if (Screen == 0)  {
                     printf ("No Screen!\n");
                     goto cleanup3;
                  }

Image = (UWORD *)
  AllocMem (MAXBOBS*2*2*46*sizeof(UWORD),MEMF_CHIP);
                           /* Bob definition:       */
                           /* 2 WORD width, 46 high */
                           /* 2 Planes deep         */
SaveBuffer = (UWORD *)
  AllocMem (MAXBOBS*3*2*46*sizeof(UWORD),MEMF_CHIP);
                           /* 2 WORD width, 46 high */
                           /* 3 Planes (PlaneOnOff)!*/
DBufBuffer = (UWORD *)
  AllocMem (MAXBOBS*3*2*46*sizeof(UWORD),MEMF_CHIP);

CollMask = (UWORD *)
```

```
  AllocMem (MAXBOBS*2*46*sizeof(UWORD),MEMF_CHIP);
                            /* Bobs    Collision     */
                            /* Mask                  */
                            /* (46 Rows of 2 Words) */


DBufPackets = (struct DBufPacket *)
  AllocMem(sizeof(struct DBufPacket) * MAXBOBS,
           MEMF_CLEAR|MEMF_CHIP);

if ((SaveBuffer == 0) | (DBufBuffer == 0) |
    (CollMask == 0) | (DBufPackets == 0) |
    (Image == 0))
      {
         printf (" No Chip Memory for Bobs !!!\n");
         goto cleanup4;
      }

Help = Image;

for (i=0; i<2*2*46; i++)
   {
      *Help = Baloon[i];
      Help++;
   }                        /* Copy BOB to Chip-Mem */

SetRGB4 (&Screen->ViewPort,0,0,5,15);
                                  /* A little Color */
SetRast (&RP,0);
SetAPen (&RP,2);

RP.BitMap = &BitMap [1];
Length = TextLength (&RP,"Balloons in Action !!!",21);
Move (&RP,320/2-Length/2,100);
Text (&RP,"Balloons in Action !!!",21);

RP.BitMap = &BitMap [0];
Move (&RP,320/2-Length/2,100);
Text (&RP,"Balloons in Action !!!",21);


BltClear (&Start, sizeof(struct VSprite),0);


BltClear (&Ende, sizeof(struct VSprite),0);


BltClear (&GelsInfo, sizeof(struct GelsInfo),0);
BltClear (BobVSprite, sizeof(struct VSprite)*MAXBOBS,0);
BltClear (Bob, sizeof(struct Bob)*MAXBOBS,0);
BltClear (&collTable, sizeof (collTable),0);
                  /* To be sure clear everything */

GelsInfo.sprRsrvd = 0xfc;
                       /* All Sprites except       */
                       /* 0 and 1 for VSprites !!  */

GelsInfo.nextLine = Nextlines;
GelsInfo.lastColor = lastColors;
GelsInfo.collHandler = &collTable;

GelsInfo.leftmost = 1;          /* Limits for       */
```

```
GelsInfo.rightmost = 318;       /* Border Collision */
GelsInfo.topmost = 13;
GelsInfo.bottommost = 198;

InitGels (&Start, &Ende, &GelsInfo);
                        /* Initialize GelsInfo and   */
RP.GelsInfo = &GelsInfo;  /* link with RastPort      */

SetCollision(0,BackController,&GelsInfo);
SetCollision(1,GelCol,&GelsInfo);

HelpPack = DBufPackets;
for (i=0; i<MAXBOBS; i++)
   {
      BobVSprite[i].Width = 2;     /* 2 WORDs width  */
      BobVSprite[i].Height = 46;  /* 46 Rows High    */
      BobVSprite[i].Flags = OVERLAY|SAVEBACK;
                                  /* VSprite is Bob  */
      BobVSprite[i].Depth = 2;    /* 2 'Planes'      */
      BobVSprite[i].ImageData = Image;
      BobVSprite[i].MeMask = 0x2; /* GEL Collision */
      BobVSprite[i].HitMask = 0x3;/* but with Bord */
      BobVSprite[i].PlanePick = 0x05;/* Plane 1 & 3 */
      BobVSprite[i].PlaneOnOff = 0x02; /* Plane 2   */


      BobVSprite[i].CollMask = CollMask+i*2*46;
      BobVSprite[i].BorderLine = &BorderLine[i][0];

      BobVSprite[i].X = 11+i*(320/MAXBOBS-10);
      BobVSprite[i].Y = 15+i*(200/MAXBOBS-10);
                                          /* Position */

      BobVSprite[i].VUserExt.vx = 1;  /* individual */
      BobVSprite[i].VUserExt.vy = 1;  /* Speed      */

      Bob[i].Flags = 0;
      Bob[i].BobVSprite = &BobVSprite[i];
      BobVSprite[i].VSBob = &Bob[i];
      Bob[i].ImageShadow = CollMask+i*2*46;
                     /* Image Shadow must be stored */
                     /* in ChipMemory.              */

      Bob[i].SaveBuffer = SaveBuffer+i*3*2*46;
      Bob[i].DBuffer = HelpPack;

      HelpPack->BufBuffer = DBufBuffer+i*3*2*46;
      HelpPack++;

      InitMasks (&BobVSprite[i]);
                           /* Calculate CollMask   */
                           /* and Borderline       */

      AddBob (&Bob[i], &RP);
                             /* Place in List      */
   }

SetRGB4 (&Screen->ViewPort, 0x5+0x2,0,0,0);
SetRGB4 (&Screen->ViewPort, 0x1+0x2,15,0,0);
     /* Plane 2  is always written with Shadow     */

SetRGB4 (&Screen->ViewPort, 0x4+0x2,15,15,15);
```

```
    toggle = 1;

    while ((*LeftMouse & 0x40) == 0x40)
       {
          for (i=0; i<MAXBOBS; i++)
             {
                BobVSprite[i].Y +=
                      BobVSprite[i].VUserExt.vy;
                BobVSprite[i].X +=
                      BobVSprite[i].VUserExt.vx;
                                    /* Move VSprites */
             }
          SortGList(&RP);                  /* new Sort */
          DoCollision (&RP);          /* Collisions test */
          DrawGList (&RP, &Screen->ViewPort);
                                 /* Generate Copper-List */
          WaitTOF();
          RemakeDisplay();
          Screen->ViewPort.RasInfo->BitMap = &BitMap[toggle];
          RP.BitMap = &BitMap[toggle];
          toggle ^= 1;
       }

    cleanup4:
    if (Image != 0)
         FreeMem(Image,MAXBOBS*2*2*46*sizeof(UWORD));
    if (SaveBuffer != 0)
        FreeMem(SaveBuffer,MAXBOBS*3*2*46*sizeof(UWORD));
    if (DBufBuffer != 0)
        FreeMem(DBufBuffer,MAXBOBS*3*2*46*sizeof(UWORD));
    if (CollMask != 0)
        FreeMem(CollMask,MAXBOBS*2*46*sizeof(UWORD));
    if (DBufPackets != 0)
        FreeMem(DBufPackets,sizeof(struct DBufPacket) *MAXBOBS);

                CloseScreen (Screen);
    cleanup3:   for (i=0; i<2; i++)
                   for (j=0; j<NewScreen.Depth; j++)
                      if ((BitMap[i].Planes[j]) != NULL)
                        {
                          FreeRaster (BitMap[i].Planes[j],
                          NewScreen.Width, NewScreen.Height);
                        }
    cleanup2:   CloseLibrary(IntuitionBase);
    cleanup1:   CloseLibrary(GfxBase);
    return(0);
}

/*******************************************************/
/* This Routine is executed by DoCollision() when a    */
/* VSprite collides with the Border                    */
/*-----------------------------------------------------*/
/* Entry-Parameters: VSprite, colliding with the Border*/
/*                   and which Border                  */
/*-----------------------------------------------------*/
/* Returned-Values: None                               */
/*******************************************************/


VOID BackController (VSprite, Border)
struct VSprite *VSprite;
```

```
BYTE Border;
{
   if ((Border & TOPHIT)  == TOPHIT)          /* Top    */
                  VSprite->VUserExt.vy *=-1;
   if ((Border & BOTTOMHIT)  == BOTTOMHIT)    /* Bottom */
                  VSprite->VUserExt.vy *=-1;
   if ((Border & LEFTHIT)  == LEFTHIT)        /* Left   */
                  VSprite->VUserExt.vx *=-1;
   if ((Border & RIGHTHIT)  == RIGHTHIT)      /* Right  */
                  VSprite->VUserExt.vx *=-1;
}


VOID GelCol(VSpriteleftabove, VSpriterightunder)
struct VSprite *VSpriteleftabove,
               *VSpriterightunder;
{
     VSpriteleftabove->VUserExt.vx *= -1;
     VSpriteleftabove->VUserExt.vy *= -1;

     VSpriterightunder->VUserExt.vx *= -1;
     VSpriterightunder->VUserExt.vy *= -1;
}
```

18.2.1 Bobs in buffered bit-maps

In the previous example we used the `Double Buffering` technique. When moving many and/or large bobs, it is possible for you to see the bobs being drawn. After moving a bob, the background is restored and the bob is redrawn. This can produce an annoying flickering effect. To avoid this, draw your bobs in one bit-map while displaying a second bit-map. Then switch bit-maps when you have finished drawing your bob.

Our program uses an Intuition screen to display the bobs. We will now explain how Intuition manages this display mode.

Intuition automatically recognizes the hi-res, interlace, HAM and Halfbrite modes. However, the dualplayfield and double buffering modes are programmed by the user. In our example program we demonstrated the double buffering functions with Intuition. You declare the screen with a custom bit-map. Then you initialize two identically sized bit-maps and switch between them by alternating the `RasInfo` structure information for the screens. After switching, you use `RemakeDisplay` to calculate the Copper list for the new current bit-map. You can also display vsprites at the same time (you don't always have to create a new Copper list for bobs).

To install the `DUALPF` mode for Intuition screens use the following procedure. First declare the screen as a custom bit-map and put in your

own bit-maps. These bit-maps cannot have more than three bit-planes. You can create an additional `RasInfo` structure that points to one bit-map (`RasInfo2.BitMap = MyBitMap2`) and link it with (`Screen->ViewPort.RasInfo.Next = &RasInfo2;`). Then point the first bit-map to the existing `RasInfo` structure (`Screen->ViewPort.RasInfo.BitMap = &MyBitMap1;`). After `RemakeDisplay` two bit-planes are displayed (First you must set the `DUALPF` mode in the `NewScreen` structure).

You should prevent any screen movement while using these two modes because new Copper lists will be created. This conflict can cause a system crash. Also, place windows in the screens that are unmovable(we have prevented this in our program by ending the program on any mouse click).

We must set up our bobs to support the `Double Buffering` mode. When we want to save the bobs' background (`VSprite.Flags = SAVEBACK`), we must save the background from both bit-maps.

Remember to reserve more memory and save the bit-map background positions for restoring both bit-maps. Store the first bit-maps position in the variables `VSprite.OldX` and `VSprite.OldY`. The background itself is in the `Bob.SaveBuffer`. The background and the position of the second bit-map are located in `DBufPacket`. You inform your bob about the `DBufPacket` structure with `Bob.DBuffer = &DBufPacket`. When this pointer is not equal to zero, the system will know that your bobs are using `Double Buffering`. This must apply to all bobs or none.

In order for `Double Buffering` to function properly, a memory area address to the variable `DBufPacket.BufBuffer` must be provided. This area is used for the background of the second bit-map and it must be the same size as the `SaveBuffer`. The system will take care of the rest.

When restoring the background, there is another technique, besides double buffering, for displaying bobs without the flickering effect. Simply wait for the electronic beam to reach the top row of the monitor (`WaitTOF` helps us here). Before the electronic beam can reach the first row of the data on the displayed bit-map, you can restore the old background and draw the new bob with `DrawGList`. Then the electronic beam displays the rest of the picture. Anything that happens after this will not be displayed because the electronic beam is already at another location.

However, this method only works with smaller and fewer numbers of bobs, not with large bobs. Also, you shouldn't move your bobs too

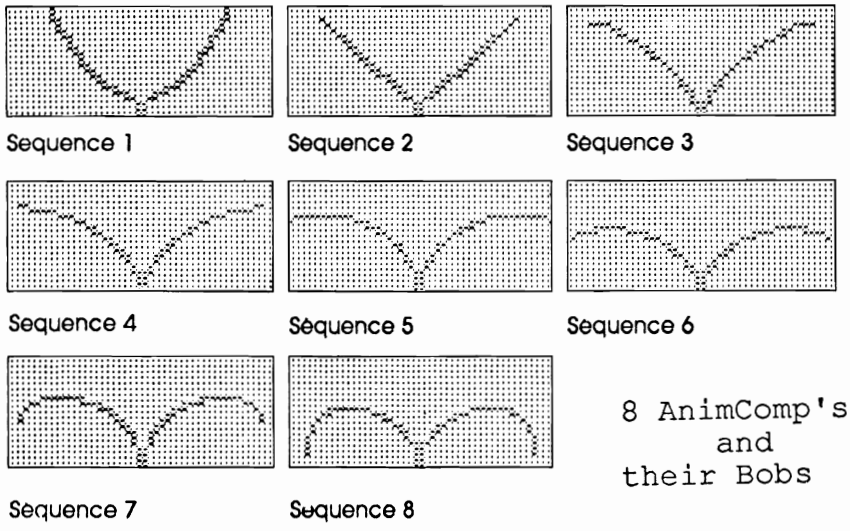
close to the upper screen border. The electronic beam is extremely fast and drawing the bobs requires some time.

18.3 AnimObs and AnimComps

Displaying only "static" bobs isn't very exciting. Because of this, a special animation system was created specifically for bobs. At the beginning of this C language section, you learned one method of animation called "color cycling" or color animation. We cycled, or scrolled, the colors in the color registers. The contents of the preceding register were exchanged for the contents of the current register, etc.

You also learned a second type of animation. When you moved your bobs and vsprites, they were also animated. You may be thinking that a Commodore 64 is able to do this too. Although this is true, the Amiga is also capable of animating bobs using the operating system software and drawing a bob in different sequences without any commands.

The Amiga can recognize `AnimationObjects` called `AnimObs`. We define `AnimObs` with `AnimationComponents`, which are named `AnimComps`. The `AnimComps` contain our defined bobs. For example, you could create the flight of a seagull with different bob sequences. Then you could link these bobs with the `AnimComps` and create an `AnimOb`.



The first step of this process is to create all your bobs as you usually do. In addition to the flags, you must determine how the bob is written into the bit-map (`OVERLAY`, `SAVEBACK`, etc.) and set the bobs

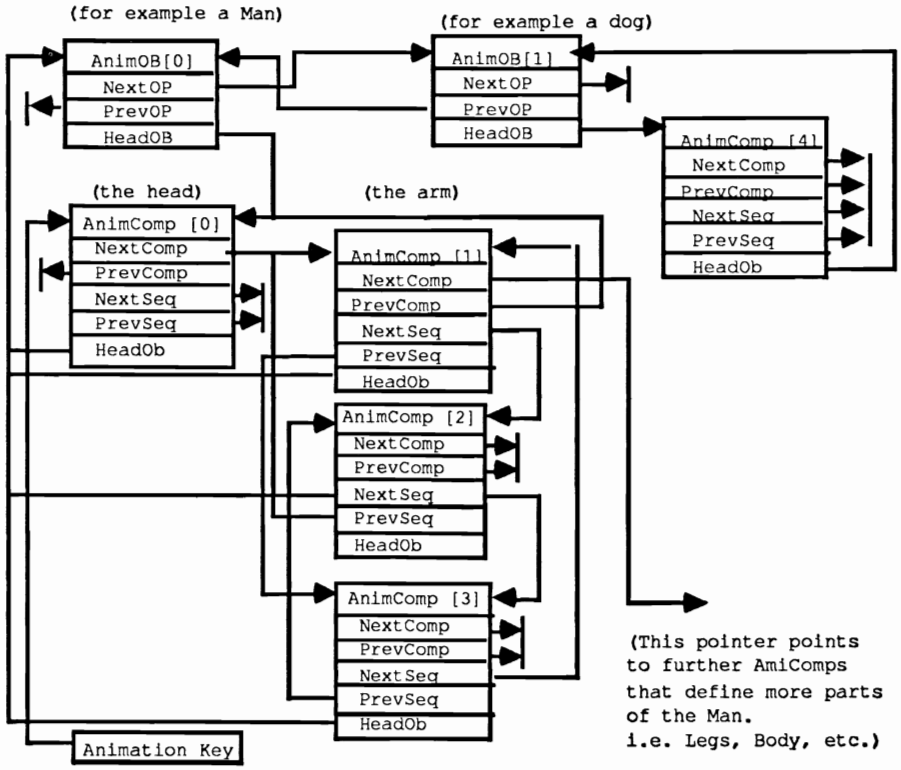
BOBISCOMP flag. This tells the animation software that your bob is an animation component.

You must also tell the system to which animation component your bob belongs. To do this, set the pointer Bob.BobComp for the corresponding AnimComp structure. Each animated bob requires its own AnimComp structure.

To link the AnimComps together, use the pointers AnimComp.NextComp and AnimComp.PrevComp. These two pointers point to the previous and next AnimComps. Be careful because when you create a loop here it should point only to a single AnimComp. Also, for all other AnimComps, you must set these pointers to zero. If you don't do this, the system will think that an AnimComp in the sequence points to a second bird instead of to the one now animated.

Next there is the pointers NextSeq and PrevSeq. From one position you can define many positions (as bobs) and then have the animation system page through your sequences. This is similar to the old "thumb movie books" which simulated animation when you rapidly paged through a series of pictures. Each new page of the bob creates the animation effect.

The next step is to set the sequence to cycle the pages. The pointers NextSeq and PrevSeq point to the next and previous page or AnimComp. These pointers create a closed ring. This means that you connect all the AnimComps to each other with the pointers NextSeq and PrevSeq. The previous AnimComp for the first AnimComp is the last AnimComp. The next AnimComp from the last AnimComp is the first AnimComp.



Setting the `RINGTRIGGER` flag in the `AnimComps` (`AnimComp.Flags = RINGTRIGGER`), which defines the ring, tells the system to flip the pages for the different `AnimComps`. You can also set the length of time a specific `AnimComp` should be displayed.

The variable `AnimComp.TimeSet` determines for how many `Animate` calls the `AnimComp` will remain visible (`Animate` is the system routine that uses the `AnimObs`). `Animate` copies the variable `TimeSet` to the variable `Timer` at the beginning of a sequence. `Timer` is decremented with each `Animate`. Whenever `Timer` equals zero, the next `AnimComp` in the sequence is displayed.

Now we will continue to define `AnimComps` and `AnimObs`. Just as a bob must know which `AnimComp` it belongs to, the `AnimComp` must also know which bob it represents. To accomplish this we point the `AnimComp.AnimBob` pointer to the specific bob.

To add something special, you can also specify one of your own routines that will be executed each time you call `Animate` to display your `AnimComps`. The pointer `AnimComp.AnimCRoutine` points to your function. Simply set this pointer to zero when you haven't

defined a function. When using this function you must remember that it returns a word value to your program. This value is not passed to the system. Because this declaration affects the AnimComp structure, you should check the returned value for any warning from the compiler.

Similar to the collision routines that are used with vsprites and bobs, your functions can use the current AnimComp structure. For example, they can check for a position and determine a specific action, or determine object distances and react.

We are almost finished with our discussion of the AnimComp structure. But before we explain our last point, we will discuss the AnimOb structure.

The AnimOb structure can contain many sequence rings. For example, it can contain the arms, legs, body and the head of a man. All of these are connected to each other and are able to move independently. We define the components as a ring sequence which is automatically sequenced by Animate. We represent the body with a single AnimComp structure because the body itself usually does not move while walking.

The two pointers PrevComp and NextComp point to the AnimComps that display the front and back view of the man. To link the AnimComps with the AnimOb we use another pointer in the AnimOb structure. The pointer AnimOb.HeadComp points to the first AnimComp that contains the rest of the objects. (You link the components together with PrevComp and NextComp). Each AnimComp contains a pointer back to the AnimOb. The pointer AnimComp.HeadOb always points to the AnimOb in which the AnimComps are included. This way you can easily access, with your routines, the required AnimOb through the AnimComp structure.

However, more is required in order to initialize an AnimOb structure. We still must determine where the structure will appear on the screen. To solve this problem we use the variables AnimOb.AnX and AnimOb.AnY. You cannot specify a coordinate using, for example, 160,100. To specify the AnimOb coordinates, you must use a type of fixed decimal method. The lower six bits are positioned after (to the right of) the decimal point. To position your AnimOb you must multiply the RastPort coordinate by 64.

You may be wondering why we are using such a strange positioning method. However, remember that these variables set not only the objects' position but also the velocity.

With each call of Animate, the values from AnimOb.XAccel and AnimOb.YAccel (X/Y movements) are added to AnimOb.XVel and

AnimOb.YVel (speed). The speed is then added to the position set by AnimOb.AnX and AnimOb.AnY.

Using normal pixel values for position, speed, and movement could make your AnimOb race across the screen. However, it wouldn't be very useful to have Animob that disappears from the screen after the fifth call to Animate. To prevent this, an AnimOb is moved only one pixel for every 64.

At first, it seems like this will be a problem. Multiplying by 64 is possible, but using normal methods to move our AnimOb anywhere in a RastPort presents a challenge. For the positions (AnX and AnY) we have only 16 bits available. Normally this would provide values from -32768 to +32768. But dividing this value by 64, gives us a range of -512 to +512. So in order to position an object at an X coordinate of 629 in a hi-res RastPort, we have to use a trick instead of the normal methods.

We can now explain the last feature of AnimComps. It is possible to specify the position of the individual animation components relative to animation objects (also in steps of 64).

When you use an offset of 128*64 in AnimComp.XTrans for your animation components you can reach any coordinate on the screen. Set positions smaller than 128 (values between -128 and zero) in AnimOb.AnX. Values between -512 and +512 for the Y position work in any resolution. To set your components' relative Y positions, use the variable AnimComp.YTrans.

Two other variables that can be used to position an object are RingXTrans and RingYTrans. Simply add, without any changes, the value of these variables to the current position. If you do not require any speed or motion, just set the corresponding AnimOb variables (XVel, YVel, XAccel and YAccel) equal to zero. Then initialize RingXTrans and RingYTrans with your desired values. Make sure your speed is uniform otherwise it will affect your animation sequences. You must synchronize the internal movements of an object (like the wings of a bird or rotation of a wheel) with any position changes. For example, the rotation of a wheel being slower than the forward movement as you increase the forward speed and the wheel always rotates at the same speed.

You can also specify a routine for your animation object that is called by Animate. The pointer for this routine is named AnimORoutine. It is also of the type word and you must specify the AnimOb, not the component.

Now that we have initialized all the animation structures, the bob, the component and the object, the only thing left to do is to display it on the screen.

Again we need a completely initialized GelsInfo structure used for bobs and linked with a RastPort.

By using AddAnimOb (&AnimOb, &Key, &RastPort) we add all bobs and components of the AnimOb to the GelsInfo structure. A required entry named Key is simply a pointer to an AnimOb (struct AnimOb *Key = 0) that must be set to zero for the first AddAnimOb call.

Because a list doesn't exist for the AnimObs, you must know which AnimOb was entered last in the GEL list. This helps ensure that the objects have been properly linked. Also link the AnimObs together by using the AnimOb.PrevOb and AnimOb.NextOb pointers. The Key always points to the last entered AnimOb.

AddAnimOb handles more than linking the objects and entering the bobs in the GEL list. It also sets the Timer variable for the AnimComps to the value previously set in TimeSet. This permits decrementing the Timer.

Once you have processed all of the AnimObs with AddAnimOb you are ready to begin. Call Animate and the Timer variable for the current AnimComps is decremented. When Timer == 0 the next sequence is activated. The position controlled by RingXTrans, RingYTrans, XVel, YVel, SAccel, YAccel, Xtrans and YTrans is calculated and then used to display the next bob.

When you call SortGList and DrawGList as usual, your AnimObs are displayed on the screen.

18.3.1 Collisions with AnimObs

Because the smallest element used to display AnimObs are bobs, you can also use them for collision control. Set the HitMask and MeMask in the vsprite structure of the bobs (using the same values for all bobs in one AnimComp loop). Then you use SetCollision with your routine and test for collisions.

We have provided another example program so that you can actually see what happens. This animation displays a flying seagull with flapping wings (in one AnimOb):

```
/*****************************************************/
/*                   LetsAnimate.c                   */
/*                                                   */
/* This program displays and moves an Amiga AmiOb    */
/* (here a Segull and uses AnmiComps (different      */
/* wing positions                                    */
/*  Compiled with Aztec C V3.6a                      */
/*   cc +L -S LetsAnimate.c                          */
/*   ln LetsAnimate.o -lc32                          */
/* (c) Prgram by Bruno Jennrich, Idea and Artwork by */
/* my little sister Ute.                             */
/*****************************************************/
#include "exec/types.h"
#include "exec/memory.h"
#include "exec/devices.h"
#include "devices/keymap.h"
#include "graphics/gfx.h"
#include "graphics/gfxmacros.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
#include "graphics/gfxbase.h"
#include "graphics/regions.h"
#include "hardware/blit.h"
#include "hardware/custom.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
#include "libraries/diskfont.h"
#include "hardware/dmabits.h"

extern WORD MoveSeagull();      /* User AnimOb-Routine */
/* extern WORD Comp();       Optional AnimComp-Routine */

#define RP Screen->RastPort     /* Pointer to RastPort */
#define MAXBOBS 8          /* Seagull in in 8 Positions */

#define MAXCOMPS (MAXBOBS*2-2)  /* Sequence consits of */
                                /* 14 pictures: 8 'to' */
                                /* and 6 'back'.       */


struct GfxBase *GfxBase;                /* BasePointer */
struct IntuitionBase *IntuitionBase;

struct NewScreen NewScreen =           /* User Screen */
                      {
                        0,0,640,2oo,1,
                        1,0,
                        HIRES,
                        CUSTOMSCREEN,
                        NULL,
                        "",
                        NULL,NULL
                      };

struct Screen *Screen;

struct VSprite Start, Ende,      /* VSprites for GEL */
               BobsVSprite[MAXCOMPS];/* List and Bobs */

struct Bob Bobs[MAXCOMPS];              /* User Bobs */
                          /* Note: 6 Bobs have the  */
                           /* same image!!!         */
```

```
UWORD *BobBuffer;
        /* MAXBOBS Bobs,  20 Lines of 3 WORDs for  */
        /* one BitPlane                            */
                    /* memory area for SAVEBACK    */
                    /* In this example one buffer  */
                    /* is enough, but because of   */
                    /* the principle and to simply*/
                    /* use new and other Bobs,     */
                    /* we have given each Bob a    */
                    /* 'SaveBuffer'                */

UWORD *BobMask;              /* Bob collisions-masks */
                             /* Memory               */
                             /* (20 Lines * 3 Words) */

UWORD BobBorderLine [MAXCOMPS][3]; /* Bobs Borderline  */
                              /* Memory              */
                              /* (logical OR of all  */
                              /* Bob lines in one    */
                              /* line (here:3 Words) */

/*extern struct Custom custom;     in custom.h */
                              /* pointer to Hardware */
                             /* Register for Copper */
struct UCopList *UCopList;    /* own Copper List    */

struct AnimComp AnimComp[MAXCOMPS];
                        /* Seagull 'back and forth    */
                        /* bit Start (Sequence 1) and */
                        /* end position (Sequence8)   */
                        /* appear in each sequence    */
                        /* only once, not twice       */
                        /* like th eother positions   */
                        /* for the Seagull            */


struct AnimOb *HeadOb = 0,          /* Animations Key */
              Seagull;               /* our Seagull  */

struct GelsInfo GelsInfo;/* GelsInfo initialization  */
                        /* must be completed before */
                        /* usinf the Animation      */
                        /* routine!                 */
UWORD *Image,*Help;
UWORD BobImage[MAXBOBS][20][3] =
        {{
             {0x0080,0x0000,0x0100},/* Data for       */
             {0x0080,0x0000,0x0100},/* Bob1. Only one */
             {0x00c0,0x0000,0x0300},/* BitPlane per   */
             {0x0040,0x0000,0x0200},/* Bob            */
             {0x0060,0x0000,0x0600},
             {0x0030,0x0000,0x0c00},/* Please use the */
             {0x0010,0x0000,0x0800},/* copy function  */
             {0x0018,0x0000,0x1800},/* of your editor */
             {0x000c,0x0000,0x3000},/* and save       */
             {0x0004,0x0000,0x2000},/* yourself a lot */
             {0x0006,0x0000,0x6000},/* of work!       */
             {0x0003,0x0000,0xc000},
             {0x0001,0x8001,0x8000},
             {0x0000,0xc003,0x0000},
             {0x0000,0x700e,0x0000},
```

```
    {0x0000,0x1c38,0x0000}, /* Here is where */
    {0x0000,0x0660,0x0000}, /* the data      */
    {0x0000,0x0180,0x0000}, /* for more      */
    {0x0000,0x0180,0x0000}, /* BitPlanes     */
    {0x0000,0x0180,0x0000}  /* follows       */
},
{
    {0x0000,0x0000,0x0000},/* Data for       */
    {0x0000,0x0000,0x0000},/* Bob2. Only one */
    {0x0200,0x0000,0x0040},/* BitPlane.      */
    {0x0100,0x0000,0x0080},
    {0x0080,0x0000,0x0100},
    {0x0060,0x0000,0x0600},
    {0x0030,0x0000,0x0c00},
    {0x0018,0x0000,0x1800},
    {0x000c,0x0000,0x3000},
    {0x0006,0x0000,0x6000},
    {0x0003,0x0000,0xc000},
    {0x0001,0x8001,0x8000},
    {0x0000,0xe007,0x0000},
    {0x0000,0x300c,0x0000},
    {0x0000,0x1818,0x0000},
    {0x0000,0x0c30,0x0000}, /* Here is where */
    {0x0000,0x0660,0x0000}, /* the data      */
    {0x0000,0x0180,0x0000}, /* for more      */
    {0x0000,0x0180,0x0000}, /* BitPlanes     */
    {0x0000,0x0180,0x0000}  /* follows       */
},
{
    {0x0000,0x0000,0x0000},/* Data for       */
    {0x0000,0x0000,0x0000},/* Bob3. Only one */
    {0x0000,0x0000,0x0000},/* BitPlane.      */
    {0x0f00,0x0000,0x00f0},
    {0x00c0,0x0000,0x0300},
    {0x0060,0x0000,0x0600},
    {0x0018,0x0000,0x1800},
    {0x0006,0x0000,0x6000},
    {0x0003,0x0000,0xc000},
    {0x0001,0x8001,0x8000},
    {0x0000,0x4002,0x0000},
    {0x0000,0x2004,0x0000},
    {0x0000,0x1008,0x0000},
    {0x0000,0x0810,0x0000},
    {0x0000,0x0c30,0x0000},
    {0x0000,0x0660,0x0000}, /* Here is where */
    {0x0000,0x0240,0x0000}, /* the data      */
    {0x0000,0x03c0,0x0000}, /* for more      */
    {0x0000,0x0180,0x0000}, /* BitPlanes     */
    {0x0000,0x0180,0x0000}  /* follows       */
},
{
    {0x0000,0x0000,0x0000},/* Data for       */
    {0x0000,0x0000,0x0000},/* Bob4. Only one */
    {0x0000,0x0000,0x0000},/* BitPlane.      */
    {0x0000,0x0000,0x0000},
    {0x3000,0x0000,0x000c},
    {0x0f80,0x0000,0x01f0},
    {0x0070,0x0000,0x0e00},
    {0x000c,0x0000,0x3000},
    {0x0003,0x0000,0xc000},
    {0x0001,0x8001,0x8000},
    {0x0000,0x6006,0x0000},
```

```
    {0x0000,0x300c,0x0000},
    {0x0000,0x1818,0x0000},
    {0x0000,0x0c30,0x0000},
    {0x0000,0x0420,0x0000},
    {0x0000,0x0240,0x0000}, /* Here is where */
    {0x0000,0x03c0,0x0000}, /* the data      */
    {0x0000,0x0180,0x0000}, /* for more      */
    {0x0000,0x0180,0x0000}, /* BitPlanes     */
    {0x0000,0x0180,0x0000}  /* follows       */
},
{
    {0x0000,0x0000,0x0000},/* Data for       */
    {0x0000,0x0000,0x0000},/* Bob5. Only one */
    {0x0000,0x0000,0x0000},/* BitPlane.      */
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
    {0x7ff0,0x0000,0x0ffe},
    {0x800f,0x0000,0x7001},
    {0x0003,0x8001,0xc000},
    {0x0000,0xc003,0x0000},
    {0x0000,0x2004,0x0000},
    {0x0000,0x1818,0x0000},
    {0x0000,0x0810,0x0000},
    {0x0000,0x0420,0x0000},
    {0x0000,0x0660,0x0000},
    {0x0000,0x0240,0x0000}, /* Here is where */
    {0x0000,0x0180,0x0000}, /* the data      */
    {0x0000,0x0180,0x0000}, /* for more      */
    {0x0000,0x0180,0x0000}, /* BitPlanes     */
    {0x0000,0x0180,0x0000}  /* follows       */
},
{
    {0x0000,0x0000,0x0000},/* Data for       */
    {0x0000,0x0000,0x0000},/* Bob6. Only one */
    {0x0000,0x0000,0x0000},/* BitPlane.      */
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
    {0x07e0,0x0000,0x07e0},
    {0x3c3e,0x0000,0x7c3c},
    {0x4003,0x8001,0xc001},
    {0x0000,0x6006,0x0000},
    {0x0000,0x300c,0x0000},
    {0x0000,0x1818,0x0000},
    {0x0000,0x0420,0x0000},
    {0x0000,0x0660,0x0000},
    {0x0000,0x03c0,0x0000}, /* Here is where */
    {0x0000,0x0180,0x0000}, /* the data      */
    {0x0000,0x0180,0x0000}, /* for more      */
    {0x0000,0x0180,0x0000}, /* BitPlanes     */
    {0x0000,0x0180,0x0000}  /* follows       */
},
{
    {0x0000,0x0000,0x0000},/* Data for       */
    {0x0000,0x0000,0x0000},/* Bob7. Only one */
    {0x0000,0x0000,0x0000},/* BitPlane.      */
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
```

```
                {0x03fc,0x0000,0x3fc0},
                {0x0c0f,0x8001,0xe030},
                {0x1000,0xe007,0x0008},
                {0x2000,0x6003,0x0004},
                {0x2000,0x1818,0x0004},
                {0x0000,0x0c60,0x0000},
                {0x0000,0x0660,0x0000},
                {0x0000,0x0240,0x0000},
                {0x0000,0x0240,0x0000}, /* Here is where */
                {0x0000,0x0180,0x0000}, /* the data      */
                {0x0000,0x0180,0x0000}, /* for more      */
                {0x0000,0x0180,0x0000}, /* BitPlanes     */
                {0x0000,0x0180,0x0000}  /* follows       */
            },
            {
                {0x0000,0x0000,0x0000},/* Data for       */
                {0x0000,0x0000,0x0000},/* Bob8. Only one */
                {0x0000,0x0000,0x0000},/* BitPlane.      */
                {0x0000,0x0000,0x0000},
                {0x0000,0x0000,0x0000},
                {0x0000,0x0000,0x0000},
                {0x0000,0x0000,0x0000},
                {0x0000,0x0000,0x0000},
                {0x01fe,0x0000,0x7f80},
                {0x0301,0xc003,0x80c0},
                {0x0400,0x6006,0x0020},
                {0x0800,0x1818,0x0010},
                {0x0800,0x0c30,0x0010},
                {0x1000,0x0420,0x0008},
                {0x1000,0x03c0,0x0008},
                {0x1000,0x0180,0x0008}, /* the data      */
                {0x0000,0x0180,0x0000}, /* for more      */
                {0x0000,0x0180,0x0000}, /* BitPlanes     */
                {0x0000,0x0180,0x0000}  /* would follow */
            }
        };

main()
{
   int  i,j,k;
   char *LeftMouse = (char *) 0xBFE001;
                                /* Left mouse button*/
   if ((GfxBase = (struct GfxBase *)
            OpenLibrary("graphics.library",0)) == NULL)
            {
               printf ("No Graphics !!!\n");
               exit(0);
            }

   if ((IntuitionBase = (struct IntuitionBase *)
            OpenLibrary("intuition.library",0)) == NULL)
            {
               printf ("No Intuition !!!\n");
               goto cleanup2;
            }

   if ((Screen = (struct Screen *)
                  OpenScreen (&NewScreen)) == NULL)
        {
          printf ("No Screen !!!\n");
          goto cleanup3;
```

```
        }

    BobBuffer = (UWORD *)AllocMem(MAXCOMPS*20*3*sizeof(UWORD),
                                  MEMF_CLEAR | MEMF_CHIP);

    BobMask = (UWORD *)AllocMem(MAXCOMPS*20*3*sizeof(UWORD),
                                  MEMF_CLEAR|MEMF_CHIP);

    Image = (UWORD *)AllocMem(MAXBOBS*20*3*sizeof(UWORD),
                                  MEMF_CLEAR|MEMF_CHIP);

    if ((Image == 0) | (BobBuffer == 0) | (BobMask == 0))
       {
          printf ("No Chip Memory !!\n");
          goto cleanup4;
       }

    Help = Image;

    for (i=0;i<MAXBOBS;i++)
       for (j=0;j<20;j++)
          for (k=0;k<3;k++)
             {
                *Help = BobImage[i][j][k];
                Help++;
             }

    SetRGB4 (&Screen->ViewPort,0,2,8,15);
    SetRGB4 (&Screen->ViewPort,1,0,0,0);

    SetRast (&RP,0);

    BltClear (&Start, sizeof(struct VSprite),0);
    BltClear (&Ende, sizeof(struct VSprite),0);
    BltClear (&GelsInfo, sizeof(struct GelsInfo),0);
    BltClear (BobsVSprite,
                  sizeof(struct VSprite)*MAXCOMPS,0);
    BltClear (Bobs,
                  sizeof(struct Bob)*MAXCOMPS,0);

    BltClear (AnimComp,
                  sizeof(struct AnimComp)*MAXCOMPS,0);

    BltClear (&Seagull, sizeof(struct AnimOb),0);
/*****************************************************/
/*            A little Copper - Power                */
/*****************************************************/

    UCopList = (struct UCopList *)
       AllocMem (sizeof(struct UCopList),
                     MEMF_CHIP | MEMF_CLEAR);

    CWAIT (UCopList, 150,0);
    CMOVE (UCopList, custom.color[0], 0x000f);  /* Sea  */
    CEND  (UCopList);

    Screen->ViewPort.UCopIns = UCopList;
                               /* Copper List linked     */
    RemakeDisplay();           /* and calculate new      */
/*****************************************************/
    GelsInfo.sprRsrvd = 0xff;/* All Sprites for VSprites*/
```

```
                /* Memory for  GelInfo reserved    */
  GelsInfo.nextLine = (WORD *)AllocMem(sizeof (WORD)*8,
                        MEMF_PUBLIC|MEMF_CLEAR);

  GelsInfo.lastColor = (WORD **)AllocMem(sizeof (LONG)*8,
                             MEMF_PUBLIC|MEMF_CLEAR);

  GelsInfo.collHandler = (struct collTable *)
                    AllocMem(sizeof (struct collTable),
                             MEMF_PUBLIC|MEMF_CLEAR);

if ((GelsInfo.nextLine == 0)|(GelsInfo.lastColor == 0) |
     (GelsInfo.collHandler == 0))
       {
          printf (" No Memory for GelsInfo !!!\n");
          goto cleanup5;
       }

  GelsInfo.leftmost = 0;       /* Boundary Collisions */
  GelsInfo.rightmost = 640;    /* Border collisions   */
  GelsInfo.topmost = 0;
  GelsInfo.bottommost = 200;

  InitGels (&Start, &Ende, &GelsInfo);
                          /* GelsInfo initialization  */
  RP.GelsInfo = &GelsInfo;  /* and in RastPort linked */

  for (i=0; i<MAXCOMPS; i++)
     {
        Bobs[i].BobVSprite = &BobsVSprite[i];
        BobsVSprite[i].VSBob = &Bobs[i];
        BobsVSprite[i].Width = 3;   /* All  Bobs are  */
        BobsVSprite[i].Height = 20; /* the same size  */
        BobsVSprite[i].Flags = SAVEBACK;
                               /* Store background    */
                               /* (in BobBuffer) and  */
                               /* restore it at a  time*/
        BobsVSprite[i].Depth = 1;
                            /* Only one Plane per Bob */

                            /* BobsVSprite[i].ImageData */
                            /* initialized in extra  loop*/

        BobsVSprite[i].PlanePick = 1;
                           /* Only first plane wriiten  */
                           /* to Rast Port              */

        BobsVSprite[i].PlaneOnOff = 0;
                          /* Remaining planes remain  0 */

        BobsVSprite[i].CollMask = BobMask+i*20*3;
        BobsVSprite[i].BorderLine = &BobBorderLine[i][0];
             /* Memory for CollMask and Borderline */
             /* prepared                           */

        Bobs[i].ImageShadow = BobMask+i*20*3;
                                  /* Shadow = CollMask */
        Bobs[i].Flags = BOBISCOMP;
                                    /* Bob is part of  */
        Bobs[i].BobComp = &AnimComp[i];
                               /* ...this   AnimComps */
```

```
            Bobs[i].SaveBuffer = BobBuffer+i*20*3;
                        /* Memory for SAVEBACK Option */

            BobsVSprite[i].Y = 0;     /* Position word is  */
            BobsVSprite[i].X = 0;     /* calculated by the */
                                      /* animation system! */

            Bobs[i].Before = 0;     /* Priorities too!!!!! */
            Bobs[i].After = 0;

            InitMasks (&BobsVSprite[i]);
                /* Initialize  CollMask and BorderLine. */
                /* (Memory must already be set aside!!  */
        }

    for (i=0; i<MAXBOBS; i++)
          BobsVSprite[i].ImageData = Image+i*20*3;
    for (i=MAXBOBS-2; i>0; i--)
          BobsVSprite[MAXCOMPS-i].ImageData =
                                        Image+i*20*3;
                                /* Order the Bobs for  */
                                /* Sequence animation */
    for (i=0; i<MAXBOBS; i++)
        {
          AnimComp[i].AnimBob = &Bobs[i];
          AnimComp[i].PrevComp = 0;     /* no further    */
          AnimComp[i].NextComp = 0;     /* AnimComp in   */
                                        /* AnimOb        */
          AnimComp[i].TimeSet = 3;/* 3 mal Animate()    */
                                  /* before new sequence*/
                                  /* is displayed.      */
          AnimComp[i].Flags = RINGTRIGGER;
                                /* Ring-Sequence-Animation*/
          AnimComp[i].XTrans = 128*64;
          AnimComp[i].YTrans = 0;
                          /* Offset to AnX/AnY in Seagull*/
                          /* Note: Fixed decimal !!!     */
          AnimComp[i].AnimCRoutine = NULL;     /* Comp; */
                                /* no AnimComp Routine     */
          AnimComp[i].HeadOb = &Seagull;
                  /* HeadOb for AnimComp (for one      */
                  /* routine as intersection mark!)    */
        }
/* Sequence:        1 2 3 4 5 6 7 8 9 10 11 12 13 14   */
/* Bobs/AnimComp:   1 2 3 4 5 6 7 8 7 6  5  4  3  2    */

    for (i=MAXBOBS-2; i>0; i--)
        {
          AnimComp[14-i].AnimBob = &Bobs[i];
          AnimComp[14-i].PrevComp = 0;
          AnimComp[14-i].NextComp = 0;
          AnimComp[14-i].TimeSet = 3;
          AnimComp[14-i].Flags = RINGTRIGGER;
          AnimComp[14-i].XTrans = 128*64;
          AnimComp[14-i].YTrans = 0;
          AnimComp[14-i].AnimCRoutine = NULL; /* Comp; */
          Bobs[14-i].BobComp = &AnimComp[14-i];
          AnimComp[14-i].HeadOb = &Seagull;
        }                                   /* see above */

    for (i=1; i<MAXCOMPS-1; i++)
        {
```

```
        AnimComp[i].NextSeq = &AnimComp[i+1];
        AnimComp[i].PrevSeq = &AnimComp[i-1];
          /* intitialize  'Ring' for  Ring/Sequence   */
          /* Animation (PrevComp and NextComp remain  */
          /* here without action and in this case     */
          /* are not initialized, but set to 0        */
      }

   AnimComp[0].NextSeq = &AnimComp[1]; /* Close 'Ring' */
   AnimComp[0].PrevSeq = &AnimComp[13];

   AnimComp[MAXCOMPS-1].NextSeq = &AnimComp[0];
   AnimComp[MAXCOMPS-1].PrevSeq = &AnimComp[12];

   Seagull.HeadComp = &AnimComp[0]; /* AnimOb's first  */
                                    /* AnimComp        */
   Seagull.RingXTrans = 2*64;       /* X/Y Translation */
   Seagull.RingYTrans = 1*64;       /* of  seagull     */
   Seagull.AnX = 0;                /* start   position */
   Seagull.AnY = 0;                /*  of XTrans       */
   Seagull.XAccel = 0x0000;       /* has no movement   */
   Seagull.YAccel = 0x0000;
   Seagull.XVel = 0x0000;      /* no movement speed    */
   Seagull.YVel = 0x0000;
   Seagull.AnimORoutine = MoveSeagull;
                          /* User control routine */

   AddAnimOb (&Seagull, &HeadOb, &RP);
                                    /* AnimOb in list  */
   while ((*LeftMouse & 0x40) == 0x40)
      {
         Animate(&HeadOb,&RP);            /* Sort       */
         SortGList(&RP);                  /* Animation  */
         WaitTOF();/* to prevent blinking the bobs!    */
         DrawGList(&RP,&Screen->ViewPort); /* Draw      */
      }
                     /* GelsInfo's memory freed        */
   cleanup5:
   if (GelsInfo.nextLine != 0)
      FreeMem (GelsInfo.nextLine, sizeof (WORD)*8);
   if (GelsInfo.lastColor != 0)
      FreeMem (GelsInfo.lastColor, sizeof (LONG)*8);
   if (GelsInfo.collHandler != 0)
      FreeMem (GelsInfo.collHandler,
                           sizeof(struct collTable));

   cleanup4:
   if (Image != 0)
       FreeMem(Image,MAXBOBS*20*3*sizeof(UWORD));
   if (BobBuffer != 0)
       FreeMem(BobBuffer,MAXCOMPS*20*3*sizeof(UWORD));
   if (BobMask != 0)
       FreeMem(BobMask,MAXCOMPS*20*3*sizeof(UWORD));

   cleanup3:   CloseScreen (Screen);
   cleanup2:   CloseLibrary(IntuitionBase);
   cleanup1:   CloseLibrary(GfxBase);
}

/*******************************************************/
/* This function is called each time by Animate()      */
/*-----------------------------------------------------*/
```

```
/* Input parameter  : AnimOb-Structur, animated with   */
/*                    Animate(). These parameters are  */
/*                    passed by Animate()              */
/*-----------------------------------------------------*/
/* Return value   : none                               */
/*******************************************************/
WORD MoveSeagull (Object)
struct AnimOb *Object;
{
   if ((Object->AnX < (-128*64)) ||/* Ist user object  */
       (Object->AnX > ((512-48)*64))) /* corner?       */
      Object->RingXTrans *= -1;

   if ((Object->AnY < (0)) ||
       (Object->AnY > (120*64)))
      Object->RingYTrans *= -1;
}
/*******************************************************/
/* This function is called each time by Animate()      */
/*-----------------------------------------------------*/
/* Input parameter  : AnimOb-Structur, animated with   */
/*                    Animate(). These parameters are  */
/*                    passed by Animate()              */
/*-----------------------------------------------------*/
/* Return value   : none                               */
/*******************************************************/
/* WORD Comp(Component)
struct AnimComp *Component;
{
   return(0);
}  */
```

19. Copper programming in C

As you already know from the previous chapters, the Copper is a co-processor of the Amiga. It is responsible for the visible display, which means that it determines what appears at a specific position of the electronic beam.

The Copper also helps display sprites and vsprites However, a more important feature of the Copper is that it can be programmed by the user. Simply insert a pointer to the user Copper list (`struct UCopList *UserCopperList`), which uses the Copper instructions to program the Copper.

However, before you can use this you must assign enough memory for your user Copper list: `UserCopperList = (struct UCopList *) AllocMem(sizeof (struct UCopIns), MEMF_PUBLIC | MEMF_CLEAR).`

You must clear the memory for this structure. Do this either directly with `AllocMem` through the `MEMF-CLEAR` option, or afterwards with `BltClear`. This allows the entry of new Copper instructions and then tells `MakeVPort` that the user Copper list is still empty.

Now that we have explained all the preliminary steps, we can proceed to the Copper language. It is very simple and consists of only three instructions, `CMOVE`, `CWAIT` and `CEND`. These three instructions are all that is needed to program pull down Intuition screens.

The `CMOVE` instruction enables you to write a value into a specific hardware register (see Appendix C). Both the hardware register and the value are specified by you with the `Custom` structure, which allows you to access the hardware registers. To create this structure first use `extern struct Custom custom` and then use `custom.<Registername>` to access the individual hardware registers.

Now give the `CMOVE` instruction as a parameter and, as you may have assumed, the absolute address of the desired hardware register. The Copper only works with offsets of the registers from $DF000 and `CMOVE` calculates the absolute address for you.

You must provide a pointer to this structure beforehand so that the `CMOVE` instruction also knows where to find the user Copper list.

Now you must supply the 16 bit value (Word) that will be written into the desired hardware register.

A complete call with the `CMOVE` instruction would look like this: `CMOVE (UserCopperList, Custom.<RegisterName>, Value);`.

There is a small restriction you should remember when using the `CMOVE` instruction. Usually you can write to any hardware register numbered higher than $20 (dskpt) without any limitations. However, you cannot, under any circumstances, address register numbers smaller than $10 (adkconr) through the Copper. It is possible to write only to registers that fall between $10 and $20 after you have set the Copper `DangerBit` in register (copcon) (number $2E).

When discussing sprite collision detection, we explained how you access the hardware registers with help from the 68000. Review that section if necessary.

Another Copper instruction is `CWAIT`. This instruction enables you to wait for the electronic scanning beam to reach a specific position. The user Copper list doesn't perform any other instructions until the scanning beam reaches the specified position (The Copper program won't have any effect on your C program). For example, you could wait for any desired position and change the contents of one of the color registers. You could also use this technique to display one of your sprites beginning in the middle of the screen.

When you use the `CWAIT` instruction you must tell the user Copper list where your instructions will be located. You must also provide the X and Y position that the electronic beam should wait for. The order of the X and Y coordinates is very important. First specify the Y coordinate and then the X coordinate. Pay close attention to the order of these coordinates because using them out of sequence can be very frustrating. Your Copper program will not function properly if the Y coordinate isn't first.

`CWAIT(&UserCopperList,Y,X)` lets you wait for a specific electronic beam position. Remember that the Y position must be smaller than 263 and the X position must be smaller than 223. Also, you must set the Y position relative to the top of your ViewPort and the X position relative to the normal scanning position. This means that for the X position you have to consider `Overscan`. The electronic beam actually covers a much larger area than is visible on the screen. A good value to use for positioning the electronic beam at the left ViewPort border is X=60 plus or minus 2 or `View.DxOffset/2` plus or minus 1.

So that the Copper program ends properly, we use the CEND instruction. The only parameter needed is the user Copper list that will be ending. CEND (&UserCopperList) waits for the electronic beam to reach row position 10000 and column position 256. Since this position is never reached, the user Copper list is no longer used.

When the electronic beam reaches the bottom of the screen, it will start at the top row again (Top Of Frame). All Copper lists for the ViewPorts and the user Copper list are run again.

The Copper can perform raster row interrupts similar to the ones from the Commodore 64. However, unlike the Commodore 64, the Copper can also change a color in the middle of a raster row. Remember that each CMOVE instruction requires a maximum resolution of 8 pixels. This means that with a normal resolution, a CMOVE instruction can only be performed every 8 pixels. Therefore, between two consecutive CMOVE instructions there must be an 8 pixel gap.

The CWAIT instruction allows you to wait for electronic beam positions that are spaced only 4 pixels apart. However, when a CMOVE instruction follows a CWAIT instruction, an 8 pixel gap is still required.

After using the above instruction to create your user Copper list, you must provide this information to the ViewPort that will use the list. To do this use ViewPort.UCopIns = UserCopperList. When using Intuition screens, use the following form, Screen->ViewPort.UCopIns = &UserCopperList.

Now open your own screens by calling MakeVPort and MrgCop.

Intuition screens are handled differently. OpenScreen generates the Copper list for the screen. Afterwards, use RemakeDisplay to ensure that the new user Copper list is added to the global View Copper list.

FreeVPortCprList and CloseScreen release the dynamically reserved memory for the UCopList, a two word pointer that contains your instructions. No additional instructions, such as FreeMem, need to be executed. Make sure you declare your user Copper list as a pointer and reserve the required memory for the UCopList structure in your program. FreeVPortCprList and CloseScreen free the memory for not only the instruction list, but also for the UCopList structure.

When you declare the UCopList as a normal structure, the memory that it uses is released twice. It is released first by FreeVPortCopLists and then by your program, which ends by

returning all used memory areas to the system. This double release of one memory area causes the familiar Guru Meditations.

To avoid this, always declare your `UCopList` as a pointer that is later used to assign your memory.

Finally, another small tip: To change your Copper list in your program you must clear your reserved `UCopList` structure (`BltClear` is best) and build a new list.

```
/*******************************************************/
/*                      Copper.c                       */
/*                                                     */
/* This Program demonstrates how you can access the    */
/* AMIGA Hardware-Registers with Help from the Copper. */
/* Compiled with: Aztec C 3.6a                         */
/*  cc +L -S Copper.c                                  */
/*  ln Copper.o -lc32                                  */
/* (c) Bruno Jennrich                                  */
/*******************************************************/

#include "exec/types.h"
#include "exec/nodes.h"
#include "exec/memory.h"
#include "graphics/gfx.h"
#include "graphics/gfxmacros.h"
#include "graphics/gfxbase.h"
#include "graphics/text.h"
#include "graphics/regions.h"
#include "graphics/clip.h"
#include "graphics/view.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
#include "hardware/blit.h"
#include "hardware/custom.h"

#define WIDTH 320
#define HEIGHT 200
 /*   PAL 256 HERE */

#define MODES 0

struct View View;
struct ViewPort ViewPort;
struct RasInfo RasInfo;
struct BitMap BitMap;
struct RastPort RastPort;

struct GfxBase *GfxBase;

struct View *oldview;

struct UCopList *UserCopperList;    /* our Copper List */
extern struct ColorMap *GetColorMap();
/* extern struct Custom custom;   in custom.h */
                                   /* For Access to the */
                                  /* Hardware-Register */
```

```
UWORD Colors[16] = {
                     0x000,0x0bbd,0x0f0,0xf00,
                     0x123,0x435,0x678,0x009,
                     0x123,0x435,0x678,0x009,
                     0x123,0x435,0x678,0x009
                    };
                                    /* Own ColorMap.    */
                                    /* Color reg. 1 is  */
                                    /* changed by the   */
                                    /* Copper.          */
char *LeftMouse = (char *)0xbfe001;

char *Texts[15] = {"COPPER-Programing with the AMIGA",
                   "",
                   "The SPECIAL-EFFECTS Processor in Action",
                   "",
                   "",
                   "Colors underneath the Text",
                   "(a standard feature of Games)",
                   "and how it is done.",
                   "",
                   "",
                   "",
                   "This Effect is especially spectacular",
                   "when used with Moving BOBS !",
                   "",
                   "(MOUSE BUTTON)"};



/*******************************************************/
/* Here we go  !                                       */
/*******************************************************/
main()
{
   long i,Len;

   if ((GfxBase = (struct GfxBase *)
       OpenLibrary("graphics.library",0))==NULL)
      {
         printf (" No Graphics !!!\n");
         Exit(10);
      }

   oldview = GfxBase->ActiView;  /* Build a Screen as  */
   InitView(&View);              /* usual              */
   InitVPort(&ViewPort);

   View.Modes = MODES;
   View.ViewPort = &ViewPort;
   ViewPort.DWidth = WIDTH;
   ViewPort.DHeight = HEIGHT;
   ViewPort.Modes = MODES;

   RasInfo.RyOffset = 0;
   RasInfo.RxOffset = 0;
   RasInfo.Next = 0;

   ViewPort.RasInfo = &RasInfo;
   ViewPort.ColorMap = GetColorMap(16);
```

```
LoadRGB4(&ViewPort,&Colors,16);

InitBitMap(&BitMap,4,WIDTH,HEIGHT);
for (i=0; i<4; i++)
   /
      BitMap.Planes[i] = (PLANEPTR)
                         AllocRaster(WIDTH,HEIGHT);
      if (BitMap.Planes[i] == NULL)
         {
            printf("No BitMap - Space !!!\n");
            Exit(10);
         }


InitRastPort(&RastPort);

RastPort.BitMap = &BitMap;
RasInfo.BitMap = &BitMap;

SetRast(&RastPort,0);

SetAPen(&RastPort,1);

for (i=0;i<17;i++)
   {
      Len = WIDTH/2-
        TextLength(&RastPort,Texts[i],strlen(Texts[i]))/2;

      Move(&RastPort,Len,i*9+63+RastPort.TxBaseline);

      /* Y-Coordinate should be divisible by 9 ! */

      Text (&RastPort,Texts[i],strlen(Texts[i]));


MakeVPort(&View,&ViewPort); /* Display ViewPort.    */
MrgCop(&View);              /* Makes the difference */
LoadView(&View);            /* visible.             */

UserCopperList = (struct UCopList *)
          AllocMem(sizeof(struct UCopList),MEMF_CHIP);
                /* Reserve the required Memory    */
                /* for the UCopList-Structure     */

BltClear(UserCopperList,sizeof(struct UCopList),0);
                        /* Clear UCopList-Structure  */
   for (i=0; i<256; i+=9)
      {
         CWAIT (UserCopperList,i,View.DxOffset/2);
         CMOVE(UserCopperList,custom.color[1],0x0fff);
                                        /* white  */

         CWAIT (UserCopperList,i+3,View.DxOffset/2);
         CMOVE(UserCopperList,custom.color[1],0x0bbd);
                                  /* light purple  */

         CWAIT (UserCopperList,i+5,View.DxOffset/2);
         CMOVE(UserCopperList,custom.color[1],0x088b);
                                        /* purple  */

         CWAIT (UserCopperList,i+7,View.DxOffset/2);
         CMOVE(UserCopperList,custom.color[1],0x0558);
```

```
                                        /* dark purple  */
    }
CEND (UserCopperList);                  /* End UCopList */

Delay (100);                          /* Wait 2 Seconds  */

ViewPort.UCopIns = UserCopperList;
                    /* Link ViewPort and UCopList */

MakeVPort(&View,&ViewPort);
              /* Calculate ViewPort-Copper-List  */
              /* with a Intuition Screen using   */
              /* RethinkDisplay()                */
MrgCop(&View);
LoadView(&View);

while ((*LeftMouse & 0x40) == 0x40);
                           /* Wait for Mouseclick */
LoadView(oldview);

for (i=0; i<4; i++)
     FreeRaster(BitMap.Planes[i],WIDTH,HEIGHT);

FreeColorMap (ViewPort.ColorMap);

FreeVPortCopLists(&ViewPort);
          /* UCopList automatically released   */

FreeCprList(View.LOFCprList);
FreeCprList(View.SHFCprList);
                         /* View-Copper List FREE */

CloseLibrary(GfxBase);
return (0);
}
```

Appendix A: Structures and include files

This Appendix contains a listing of all symbolic constants (`#define xyz Numerical Value`) and the include files in which these constants are located. Also listed are all used structures (`struct`) and all used C macros (`#define xyz()`). At the end of this Appendix, we explain the structure functions and provide more details for the most important structure elements.

Declaration	Include file
struct AnimOb	"graphics/gels.h"
struct AnimComp	"graphics/gels.h"
struct AreaInfo	"graphics/rastport.h"
#define AREAOUTLINE	"graphics/rastport.h"
struct AvailFonts	"libraries/diskfont.h"
struct AvailFontsHeader	"libraries/diskfont.h"
struct BitMap	"graphics/gfx.h"
struct Bob	"graphics/gels.h"
#define BNDRYOFF()	"graphics/gfxmacros.h"
#define BOBISCOMP	"graphics/gels.h"
#define BOBSAWAY	"graphics/gels.h"
#define BOTTOMHIT	"graphics/collide.h"
#define CEND()	"graphics/gfxmacros.h"
#define CMOVE()	"graphics/gfxmacros.h"
struct ColorMap	"graphics/view.h"
#define COMPLEMENT	"graphics/rastport.h"
struct Custom	"hardware/custom.h"
#define CUSTOMBITMAP	"intuition/intuition.h"
#define CUSTOMSCREEN	"intuition/intuition.h"
#define CWAIT()	"graphics/gfxmacros.h"
struct DBuffPacket	"graphics/gels.h"
#define DUALPF	"graphics/view.h"
#define EXTRA_HALFBRITE	"graphics/view.h"
struct GelsInfo	"graphics/gels.h"
#define GELGONE	"graphics/gels.h"
struct GfxBase	"graphics/gfxbase.h"
#define HAM	"graphics/view.h"
#define HIRES	"graphics/view.h"
#define INVERSVID	"graphics/rastport.h"
struct IntuitionBase	"intuition/intuitionbase.h"
struct IntuiMessage	"intuition/intuition.h"

Declaration	Include file
#define JAM1	"intuition/intuition.h"
#define JAM2	"intuition/intuition.h"
#define LACE	"graphics/view.h"
#define LEFTHIT	"graphics/collide.h"
struct NewScreen	"intuition/intuition.h"
struct NewWindow	"intuition/intuition.h"
#define MEMF_CHIP	"exec/memory.h"
#define MEMF_PUBLIC	"exec/memory.h"
struct Menu	"intuition/intuition.h"
struct MenuItem	"intuition/intuition.h"
#define OVERLAY	"graphics/gels.h"
#define PFBA	"graphics/view.h"
#define RASSIZE()	"graphics/gfx.h"
struct RasInfo	"graphics/view.h"
struct RastPort	"graphics/rastport.h"
#define RemBob()	"graphics/gels.h"
#define RINGTRIGGER	"graphics/gels.h"
#define RIGHTHIT	"graphics/collide.h"
struct Screen	"intuition/intuition.h"
#define SAVEBACK	"graphics/gels.h"
#define SAVEBOB	"graphics/gels.h"
#define SUSERFLAGS	"graphics/gels.h"
#define SELECTDOWN	"intuition/intuition.h"
#define SELECTUP	"intuition/intuition.h
#define SetOPen()	"graphics/gfxmacros.h"
#define SetDrPt()	"graphics/gfxmacros.h"
#define SetWrMsk()	"graphics/gfxmacros.h"
#define SetAfPt()	"graphics/gfxmacros.h"
struct SimpleSprite	"graphics/sprite.h"
#define SPRITE_ATTACHED	"graphics/sprite.h"
#define SPRITES	"graphics/view.h"
struct TextAttr	"graphics/text.h"
struct TextFont	"graphics/text.h"
struct TmpRas	"graphics/rastport.h"
#define TOPHIT	"graphics/collide.h"
struct UCopList	"graphics/copper.h"
typedef ULONG	"exec/types.h"
typedef UWORD	"exec/types.h"
struct View	"graphics/view.h"
struct ViewPort	"graphics/view.h"
#define VP_HIDE	"graphics/view.h"
struct VSprite	"graphics/gels.h"
#define WBENCHSCREEN	"intuition/intuition.h"
struct Window	"intuition/intuition.h"

```
struct AnimComp
    {
    WORD Flags;
                /*  These variables enable you to determine the type of
                    animation to use. When you set the RINGTRIGGER flag it
                    means you have set up a ring of AnimComps. You link the
                    ring through AnimComp.NextSeq and
                    AnimComp.PrevSeq. Animate can automatically display
                    different sequences (like a flying bird) by using the
                    AnimComp ring.

                    Unless you set RINGTRIGGER, animation is not possible.
                */
    WORD Timer;
                /*  This variable is loaded with the TimeSet value and
                    decremented (if set to decrement to zero) with each
                    Animate() call. Whenever Timer reaches zero, depending
                    on your setup (RINGTRIGGER), a new sequence is displayed.
                */

    WORD TimeSet;
                /*  The value contained here is written into Time and
                    decremented there by every Animate() call. TimeSet
                    determines how long an animation sequence is active. The
                    next sequence starts after a certain number of Animate ()
                    calls. */

    struct AnimComp *NextComp;
    struct AnimComp *PrevComp;
                /*  These two variables allow you to link many animation
                    objects (AnimOb) together for display by Animate(). For
                    example, the AnimObs for the arms, legs and head of a man
                    can be linked together. Please remember that NextComp
                    and PrevComp should not be used to animate a sequence for
                    moving an arm while walking. You must use NextSeq and
                    PrevSeq for these type of animations. */

    struct AnimComp *NextSeq;
    struct AnimComp *PrevSeq;
                /*  When you want to repeatedly change an animation
                    component, define different display (sequences) of an
                    object (for example, an arm). Then use the above two
                    pointers to tell Animate() that you want the arm
                    displayed in different positions. You program the various
                    arm movement positions and then they are displayed. */

    WORD (*AnimCRoutine)();
                /*  You either set this pointer to zero or point it to a function
                    you have defined. Each call of Animate() that displays
                    your components also calls this function. Your routine is
                    passed to the current AnimComp structure. */
```

```
    WORD YTrans, XTrans;
                /*  These two pointers contain the position of the AnimComps
                    relative to the AnimObs defined earlier. (Please remember to
                    use the fixed decimal arithmetic for these variables). */

    struct AnimOb *HeadOb;
                /*  This pointer points to the previously defined AnimOb of
                    which the AnimComp is a part. */

    struct Bob *AnimBob;
                /*  Naturally Animate() must also know what should be
                    displayed. For this purpose the AnimComp structure
                    contains a pointer to the bob that is associated with the
                    AnimComp. (Please make sure that the bob also has its own
                    vsprite structure). */

}
```

The AnimComp, or the animation components, determine the connection between bob and animation object (AnimOb). This is especially important with Sequence Animation (RINGTRIGGER) because it connects the individual components together in a ring and sets the amount of time each sequence is active.

---

struct AnimOb

```
{
    struct AnimOb *NextOb, *PrevOb;
                /*  These variables make it possible for you to link many
                    animation objects together. The linked objects are then
                    animated with Animate() and then with SortGList(),
                    DrawGList(), etc. are all displayed on the screen (for
                    example, several men). */

    LONG Clock;
                /*  This variable contains a count of the calls to Animate()
                    that have been used for a AnimOb. */

    WORD AnOldY, AnOldX;
                /*  These variables contain the old position of an animation
                    object. We save the old position because the current
                    position (AnX, AnY) does not change until Timer reaches
                    zero for the current AnimComp. The user can change the
                    position between movements which causes some
                    components to be displayed in the wrong positions. For
                    this reason, we store the old position. After numerous
                    Animate() calls the actual position is calculated using the
                    old position and the following variables. */

    WORD AnX, AnY;
                /*  These variables contain the actual position of the
                    AnimObs. They do not contain values for pixel and row
                    positions within a RastPort. Depending on the speed and
```

motion variables, they contain values in steps of 64 for pixels and lines.

This means you must use `AnX = Width/2 * 64` and `AnY = Height/2 *64` to move an `Animob` to the middle of the screen. Again we have a problem using horizontal values plus or minus 32768 versus positions over 512 (= 32768/64). The following trick can help: The variable `XTrans` (and `YTrans`) in the `AnimComp` structure set the position of an `AnimComp` relative to the previously defined `AnimOb`. When you simply initialize `XTrans` with a value of 128*64 you can move an object around the entire screen (512+128 = 640) even in hi-res mode. For horizontal positions smaller than 128 you simply use negative values with `AnX`. */

```
WORD YVel, XVel;
        /*  The speed of an AnimOb is contained in these two
            variables. The values in XVel and YVel are added to AnX
            and AnY after every Animate() call. Because Animate()
            is normally called many times per second it is possible for
            your object to move erratically on the screen. To prevent
            this, velocity and acceleration are set using step values of
            64 for Animate() (this is the reason for the unusual
            method used for the values in AnX and AnY). This means
            that with a value of one in XVel, 64 Animate() calls have
            to occur before the object moves one pixel. */

WORD YAccel, XAccel;
        /*  These variables determine the motion of your AnimOb.
            These values must also be set in steps of 64. The values in
            XAccel and YAccel are added to XVel and YVel. */

WORD RingYTrans, RingXTrans;
        /*  These variables set the speed of the AnimOb. They are
            added directly to AnX and AnY. Motion isn't a factor here.
            */
WORD (*AnimORoutine)();
        /*  The routine whose address is specified here is called once
            by every call of Animate(). This routine is passed to the
            current AnimOb structure so that the actual position of the
            current AnimOb is controlled and the proper reaction is
            received. */
struct AnimComp *HeadComp;
        /*  This pointer points to the first animation component of an
            AnimOb. */

AUserStuff AUserExt;
        /*  Link your own structures (see VSprite.VUserStuff) here.
            */
}
```

The `AnimOb` structure contains a complete animation object. You use `AddAnimOb()` to make this structure available to the system and then use `Animate()` for animation.

struct AvailFonts

```
{
    UWORD af_Type;
                /*  These variables determine whether the TextAttr structure
                    specified below uses a font in memory (AFF_MEMORY) or a
                    disk font in the "SYS:fonts" directory (AFF-DISK). This
                    is important for determining whether you should use
                    OpenFont() or OpenDiskFont() to open a font. */

    struct TextAttr af_Attr;
                /*  This is the TextAttr structure returned by AvailFonts().
                    */

}
```

This structure is only used by the routine `AvailFonts()` (and by you, of course). It is written after the `AvailFontsHeader` is located in a memory area where it is available for use by `AvailFonts()`.

struct AreaInfo

This structure is required for the `Area` function. It is initialized with `InitArea` and used for the coordinates of a polygon corner points.

struct AvailFontsHeader

This structure can only be created by using the routine `AvailFonts()` and contains only one variable: `afh_NumEntries`. This variable contains the count of the `AvailFonts` structures written after the `AvailFontsHeader` in the memory area specified by the `AvailFonts()` routine.

struct BitMap

```
{
    UWORD BytePerRow;
                /*  This variable contains the byte count required for one bit-
                    map row (BytesPerRow = Width/8). */

    UWORD Rows;
                /*  The line count for a bit-map (height) is contained in this
                    variable. */

    UBYTE Flags;
                /*  This variable is only used by the system. */
```

```
    UBYTE Depth;
                /*  The number of bit-planes in a bit-map are contained in this
                    variable. Please remember that the depth determines the
                    number of available colors (Colors = 2^Num-
                    Bitplanes). */

    UWORD pad;
                /*  This is one full WORD so the following pointer begins
                    with a LONG WORD address. */

    PLANEPTR Planes[8];
                /*  These eight pointers contain the addresses of the individual
                    bit-planes for the bit-map. Although now you can only use
                    six of the eight pointers, you can see there are
                    possibilities for future expansion. */

}
```

The `BitMap` structure contains the addresses for the individual memory areas where graphics are stored. In addition, information for the height, width and depth of the bit-map are contained here.

```
struct Bob
  {
    WORD Flags;
                /*  Through these variables, the user determines how the bob
                    is handled by the system. The SAVEBOB flag tells the
                    system that once the bob is drawn it won't be cleared from
                    the RastPort (Brush function). You use BOBISCOMP to
                    make the bob part of an animation component. Make sure
                    that you also point the pointer BobComp to the
                    corresponding AnimComp structure.

                    Not only the user can set the bob flags; the system can
                    also set the so called status flags. These flags provide
                    information about the status of a bob. For example, the
                    flag BOBNIX tells us that the bob has disappeared from the
                    RastPort, the background has been restored, and the bob
                    was removed from the GEL list. */

    WORD *SaveBuffer;
                /*  This pointer points to a memory area, reserved by you,
                    where the background will be stored when a bob is drawn.
                    Because bobs are normally written directly into the bit-
                    map, the background where the bob appears is destroyed.
                    When you set the SAVEBACK flag in the vsprite structure
                    the background is saved to the memory area. This area must
                    be at least as wide and as high as the bob being drawn.
                    You must also make sure that for every bit-plane of the
                    RastPort being written to (see PlanePick and
                    PlaneOnOff in the vsprite structure) a buffer in chip
```

memory (lower 512 KByte) is reserved. The background is saved into these memory buffers. */

```
WORD *ImageShadow;
```
/* As you may already know, you define a bob bit-plane by bit-plane and use `PlanePick` and `PlaneOnOff` to determine in which planes the bob is drawn. `ImageShadow` points to a memory buffer that is large enough to store one bit-plane of your bob. All the set pixels in the individual bit-planes for your bob are stored in `ImageShadow`. In other words, all the `Bob-Planes` are `OR`ed and stored in `ImageShadow` (chip memory). */

```
struct Bob *Before;
struct Bob *After;
```
/* These two pointers determine the order the bobs are drawn. They can be used to make the GEL routines use your bobs in a specific order. However, since this function removes pointers, you must do this after `AddBob()`. */

```
struct VSprite *BobVSprite;
```
/* Each bob requires a vsprite structure because the bob structure doesn't contain any variables for positioning. In addition, the GEL list is made up of only vsprite structures and the bob must be in this list somehow. To make this work, this pointer points to the vsprite structure for the bob (every bob has its own).

```
struct AnimComp *BobComp;
```
/* This pointer points to the `AnimComp` structure to which the bob belongs. You set this pointer only after setting the bob flag `BOBISCOMP`. */

```
struct DBufPacket *DBuffer;
```
/* When you want to use bobs with a double buffered bit-map, which saves both bit-maps for the background, you must initialize this pointer to `Double Buffer Packet`. This makes it possible for the GEL software to easily display the bobs in both bit-maps, without any extra work (chip memory). */

```
BUserStuff BUserExt;
```
/* Here you, the user, can add your own extensions to the bob structure. Simply use `#define BUserStuff` to define the type of extension in your program. This extension can then be accessed either by the `AnimCRoutines`, `AnimORoutines` or a collision routine. When you don't define `BUserStuff`, it is automatically defined as a `SHORT` variable. */

The bob structure describes the bob (Blitter object). Bobs can be as large as you want and can contain as many colors as the RastPort in which they are used.

struct ColorMap

You use this structure to set the colors for a ViewPort. To change them afterwards use `LoadRGB4()`, `GetRGB4()` and `SetRGB4()`.

struct Custom

This structure provides a picture of the hardware registers which helps you effectively access them with C. The Appendix for the hardware registers describes all the symbol registers for this structure and how to use them (chip memory).

struct DBuffPacket

This structure was designed for using bobs in double buffered bit-maps. The backgrounds from both bit-maps, which are selected by you, are saved here when the bob is drawn over them. When you set the SAVEBACK flag in the vsprite structure the background is saved from only one of the bit-maps. By installing a `DBufPacket (Bob.DBuffer = &DBufPacket)` for every bob, you can also save the background of the second bit-map. However, a `DBuffPacket` cannot be generated for only one bob in a GEL list. Either all or none of the bobs in the GEL list have a `DBuffPacket`.

You must also provide the `DBufPacket` structure and the address of an additional memory area in chip memory `(DBufPacket.BufBuffer = &Memory)`. This area must be the same size as the `Bob.SaveBuffer`. The other variables for the `DBufPackets` are handled by the GEL software.

struct GfxBase

`GfxBase` is your pointer to the graphic function library. You initialize this library with `GfxBase = OpenLibrary ("graphics.library", VERSION_NUMBER)`. Now you have access to all the graphic functions that the Amiga has available.

Also, `GfxBase` contains a pointer to the currently active View. When you use a program that creates its own View, without Intuition, you should save the current view. By using `OldView = GfxBase->ActiView` you can save this pointer, restore the Intuition View and, thereby, the workbench screen.

Another `GfxBase` structure variable `GfxBase->SpriteReserved` provides information on the hardware sprites that are currently available for use.

There are a few more pointers in `GfxBase`, such as the `Systemfontlist`, etc., that are exclusively for system use. These pointers can also be affected by the graphic functions.

---

struct GelsInfo

```
{
    BYTE sprRsrvd;
                /*  This variable provides information about the sprites
                    available to the vsprite generator. When your program does
                    not require hardware sprites, simply set all the bits in
                    sprRsrvd. This tells the Amiga to use all the hardware
                    sprites as vsprites. If you want to use both hardware sprites
                    and vsprites, you must ensure that your hardware sprites are
                    not used as vsprites. To accomplish this, clear the
                    corresponding bit in sprRsvrd (bit 0 for sprite 0, bit 1
                    for sprite 1, etc.). */

    UBYTE Flags;
                /*  These variables are used only by the system */

    struct VSprite *gelHead, *gelTail;
                /*  The GELs (vsprites and bobs) are organized in a GEL list.
                    These two pointers point to the beginning and end of this
                    list and are initialized with InitGels(). */

    WORD *nextLine;
                /*  This pointer points to a memory area that is eight words
                    long. It contains information on the highest vertical
                    position at which a hardware sprite can be displayed by
                    using the vsprite software.

    WORD **lastColor;
                /*  This 8 pointer array helps the Copper. It stores the address
                    of the last color definition that was saved for the hardware
                    sprites. This address is compared to the color table address
                    for a new sprite that will be displayed. If they are the
                    same, a color change is not performed by the Copper
                    because these colors have already been displayed. When all
                    eight pointers point to the same vsprite color table you
                    can display up to eight vsprites in one raster row instead
                    of only four. */

    str .ct collTable *collHandler;
                /*  This is where you store the various addresses for the
                    collision routines using SetCollision(). Whether or not
                    these are called depends on which GELs collide with which
                    collision masks (MeMask and HitMask).

    short leftmost, rightmost, topmost, bottommost;
                /*  These four variables are used to set the rectangle boundaries
                    within which your GELs are confined without having a
                    Border-Collision. When your GEL exceeds these
```

```
                boundaries, collision routine zero is called. In order for
                this to function, you must set bit zero in the GELs
                HitMask. */

    APTR  firstBlissObj,lastBlissObj;
                /*  These two pointers are only used by the operating system.
                    */
}
```

The `GelsInfo` structure contains very important variables and pointers for the GELs (graphic elements). It must be initialized with `InitGels()` before the GEL routines (`AddBob()`, `AddVSprite()`, `Animate()`, `AddAnimOb90`) can be used. This structure must be passed to the initialized RastPort (`RastPort.GelsInfo = &GelsInfo`).

---

struct IntuiMessage

This structure allows you to intercept and check messages received by Intuition, such as those from a window. With their help you can determine whether or not a menu item has been selected, the mouse has moved or a mouse button has been pressed.

---

struct IntuitionBase

Just as any library, the Intuition library also has a `BasePointer`. This base pointer is used in the same way, as the starting address for accessing the library functions.

---

struct NewWindow

```
{
    SHORT  LeftEdge, TopEdge;
    SHORT  Width, Height;
                /*  These four variables determine the position of a window
                    (LeftEdge (X coordinate), TopEdge (Y coordinate)),
                    width and height. */

    UBYTE  DetailPen, BlockPen;
                /*  These two variables are used to set the color for your
                    BlockPen and the Title. The values you use here are the
                    same ones used for SetAPen() (the number of the color
                    register). */

    ULONG  IDCMPFlags;
                /*  The IDCMP flags (Intuition Direct Communication Message
                    Ports) determine the type of communication between the
                    user and Intuition. You can decide which messages are sent,
```

by Intuition, to your program. You may only want to receive messages for a mouse click (MOUSEBUTTON) or a key press (RAWKEY / VANILLAKEY). Unfortunately, we do not have enough room here to provide a complete listing of all the IDCMPs. */

```
ULONG Flags;
```
/* By using these flags it is possible to describe a window in more detail. For example, you can set BORDERLESS so the window has no border. With ACTIVATE, a window can be active as soon as you open it and will immediately become the in/output window. Again, a complete list of all the possibilities is not possible here. */

```
struct Gadget *FirstGadget;
```
/* This pointer points to the first gadget you have created. Gadgets are similar to the small boxes seen in the upper right corner of system windows, which place the window in the background when they are clicked. */

```
struct Image *CheckMark;
```
/* This pointer points to the Image structure used for the checkmark, which is used to show static menu items. By entering a zero here you can use the default checkmark. */

```
UBYTE *Title;
```
/* When you want to use several windows it is easier to give each one a name or title. This name is displayed in the top row of the window. Title is the pointer to the first character of your title string (for example, the system window is named AmigaDOS). */

```
struct Screen *Screen;
```
/* In order for a window to exist, it needs a screen. A window must be displayed at some time. This pointer points to an opened screen where the window is later displayed. */

```
struct BitMap *BitMap;
```
/* When you have set the SUPER_BITMAP flag in the flags variables, this pointer must point to the bit-map you have created. It is possible to create a bit-map with 1024x1024 pixels and display part of it as large as the window on your screen. */

```
SHORT MinWidth, MinHeight;
SHORT MaxWidth, MaxHeight;
```
/* Once you have set the WINDOWSIZING flag in the flags variables (and also set SIZEBRIGHT (Size Border Right) or SIZEBOTTOM (Size Border Bottom) the window sizing gadget will appear. This gadget, located in the bottom right corner of your window, allows you to change the size of your window. You can set the sizing limits by using the variables MinHeight, MinWidth, MaxHeight, and MaxWidth. */

```
    USHORT Type;
                /*  This variable lets you determine whether or not this
                    window will appear in the Workbench area
                    (WBENCHSCREEN). Here you must set the screen structure
                    pointer to zero otherwise the window will appear in its
                    own screen (CUSTOMSCREEN). You must also provide the
                    NewWindow structure with an address for the screen where
                    the window should appear. */

}
```

These structures let you describe a window. After setting all the variables and pointers, call `Window = OpenWindow (&NewWindow)`. As an example, you can now use your window for graphic output. */

---

struct NewScreen

```
{
    SHORT LeftEdge, TopEdge, Width, Height, Depth;
                /*  These variables allow you to set the position, size and
                    depth (number of bit-planes) for your screen. While doing
                    this, you must also ensure that your window cannot be
                    moved horizontally (in the X direction). Otherwise, values
                    for LeftEdge that are not equal to zero will have the same
                    effect as if they were equal to zero. */

    UBYTE DetailPen, BlockPen;
                /*  The colors for text (DetailPen) and title (BlockPen) for
                    the top screen row are also set here in the same way as the
                    windows. */

    USHORT ViewModes;
                /*  In this variable you can set the display resolution mode for
                    your screen. */

    USHORT Type;
                /*  This variable sets the type of screen and must always be
                    specified as CUSTOMSCREEN. When you initialize your own
                    bit-map you also have to set the CUSTOMBITMAP flag.

    strut TextAttr *Font;
                /*  If you want to use a different font with your screen you can
                    select it here. Simply define a TextAttr structure
                    describing the new font and provide the address. This
                    screen and any windows within it will use the selected font.
                    To use the Default font (Topaz) set this variable to zero. */

    UBYTE *DefaultTitle;
                /*  As with windows, this pointer points to a title text string
                    for the top screen row of your screen. (The Workbench
                    screen has the title "Workbench Screen"). If you do not
                    want a title, set this pointer to zero. */
```

```
    struct Gadget *Gadgets;
                /*  Currently this pointer is not used and should be set to zero
                    for guaranteed upward compatibility. */

     struct BitMap *CustomBitMap;
                /*  This pointer points to your bit-map when Type is set to
                    CUSTOMBITMAP. */

}
```

The `NewScreen` structure is used to describe a screen. It is opened with `Screen = OpenScreen (&NewScreen)`.

---

struct RastPort

```
{
   struct Layer *Layer;
                /*  This pointer points to the Layer structure of the RastPort.
                    Layers are data structures that help manage windows. They
                    prevent a window from overwriting another by mistake. */

   struct BitMap *BitMap;
                /*  This is the pointer to the bit-map used by the RastPort.
                    For screens other than Intuition this must be initialized
                    later. */
   USHORT *AreaPtrn;
                /*  This pointer points to the fill pattern of the RastPort.
                    Normally an area is filled without any special pattern, but
                    by using the macro SetAfPt() you can change the fill
                    pattern. */

   struct TmpRas *TmpRas;
                /*  This pointer points to an additional memory area that is
                    used for the fill functions Area...() and Flood. This area
                    must be large enough to store the entire area that is being
                    filled. */

   struct AreaInfo *AreaInfo;
                /*  This pointer is only used by the Area...() commands.
                    The points for a polygon set by AreaDraw() and
                    AreaMove() must be stored in some location. Use
                    InitArea() to initialize an AreaInfo structure that
                    contains sufficient memory (five bytes per coordinate).
                    Then link this structure to the RastPort for get AREA
                    (RastPort.AreaInfo = &AreaInfo;). */

   struct GelsInfo *GelsInfo;
                /*  This structure is used to display vsprites and bobs in a
                    RastPort. It contains a linked list of all the graphic
                    elements for the vsprites. You can sort and display this list
                    by using SortGList() and DrawGList(). */
```

```
UBYTE Mask;
        /*  This variable contains information about which bit-planes
            of a RastPort are affected by a graphic operation. The
            normal value is 0xff which means all bit-planes are affected
            (each set bit represents an on bit-plane). You can change
            this variable as desired with SetWrMsk (&RastPort,
            Mask). */
BYTE FgPen;
        /*  This variable contains the number of the color register that
            is responsible for setting the APen color. FGPen ==
            APen, the APen was previously named ForegroundPen. */

BYTE BgPen;
        /*  BgPen == BPen. (BPen was the BackgroundPen). */

BYTE AOlPen;
        /*  AOlPen (AreaoutlinePen) == OPen */

BYTE DrawMode;
        /*  This variable contains the actual drawing mode set by the
            macro SetDrMd().*/

BYTE AreaPtSz;
        /*  This variable contains the number of rows that are in the
            fill pattern. This can always be changed by using
            SetAfPt(), but remember that the height must be set in
            powers of two. */z

BYTE linpatcnt;
        /*  This help variable is used for drawing lines. */

BYTE dummy;
USHORT Flags;
        /*  These variables contain various flags. For example, they
            can determine whether the first pixel of a line is drawn
            (Flags \=FRST_DOT) or if only one pixel per raster row is
            drawn (ONE_DOT). Another example is whether Area...()
            frames an area with the color of the OPen (Flags \=
            AREAOUTLINE).

USHORT LinePtrn;
        /*  This variable contains the 16 bit line pattern that can be
            set with the macro SetDrPt(). */

SHORT cp_x, cp_y;
        /*  These two variables contain the X and Y position of the
            graphic cursor, which you can position within the bit-map
            with the Move() command. */

UBYTE minterms[8];
        /*  We do not have much information for this and the
            following two variables. The reason for this is that these
            parameters don't provide any visible results when they are
            changed. */
```

```
SHORT PenWidth;  /* See minterms */
SHORT PenHeight; /* See minterms */

struct TextFont *Font;
            /*  This pointer points to the TextFont structure for the font
                currently in use. When the normal font ("topaz.font") becomes
                tiresome you can change fonts by using OpenFont() and
                SetFont(). */

UBYTE AlgoStyle;
            /*  This variable contains the text style type that you set with
                SetSoftStyle(). */

UBYTE TxFlags;
            /*  This variable contains the flags that define your font in
                detail for the RastPorts. Here you can determine whether a
                font supports proportional characters (TxFlags ==
                FPF_PROPORTIONAL), is loaded from RAM (FPF_ROMFONT) or from
                disk (FPF_DISKFONT). */

UWORD TxHeight;
            /*  This variable provides the character height of the current
                RastPort font. */

UWORD TxWidth;
            /*  This is the average width of the individual characters. */

UWORD TxBaseline;
            /*  This variable contains the position of the baseline for the
                font. With the style FPF_UNDERLINED the baseline is drawn in
                with each character for underlining. The most important
                aspect of the baseline is text positioning with Text().
                Strings are not positioned by using a Y position for the top
                line of the text. The string is positioned by a Y position of
                the graphic cursor and the baseline. */

WORD TxSpacing;
            /*  This variable sets the pixel width for each character (the
                width of a character). This applies to the display of single
                characters but not strings. */

APTR *RP_User;
            /*  This variable is reserved for the user. You can use this
                variable, for example, to link your own data structures with
                the RastPort for special purposes. */
```

```
    UWORD wordreserved[7];    /* Reserved */
    ULONG longreserved[2];    /* Reserved */
    UBYTE reserved[8]; /* Reserved */
}
```

When changing a bit-map, the RastPort structure is the most important structure. Most graphic commands require a RastPort structure because the actual values of the foreground pen and many other variables are available.

After initializing the RastPort with `InitRastPort (&RastPort)`, simply initialize the bit-map pointer with `RstPort.BitMap = &BitMap`.

---

struct RasInfo

```
{
    struct RasInfo *Next;
                /*  When you have set your ViewPort for DUALPF display mode
                    (Dual Playfields) you also have to specify two bit-maps
                    that overlap in the ViewPort. Here you link two RasInfo
                    structures that point to one of the two bit-maps by using
                    RasInfo1.Next = &RasInfo2. Then the first of the two
                    RasInfo structures is made available to the ViewPort with
                    ViewPort.Rasinfo = &RasInfo1;. The rest happens as
                    usual when opening the View and ViewPorts. */

    struct BitMap *BitMap;
                /*  This is the pointer to the bit-map of the RasInfo structure
                    that will be displayed in the ViewPort. */

    SHORT RxOffset, RyOffset;
                /*  These two variables determine which pixel of the bit-map
                    lines up with the upper left corner of the ViewPort. They
                    are normally set to zero which means that the upper left
                    corner of the bit-map and the ViewPort line up exactly. By
                    changing these values and then calculating a new Copper
                    list you can achieve a scrolling bit-map. */

}
```

The `RasInfo` structure is the referee between ViewPort and bit-map.

---

struct Screen

This structure, which is similar to the window structure, provides access to an already opened screen (`Screen = OpenScreen (&NewScreen)`). All system structures, such as those used for graphic output (`Screen->Rastport.xxx, Screen->ViewPort.xxx`, etc), are available to you through this structure.

struct SimpleSprite

```
{
   UWORD *posctldata;
                /*  This pointer points to a memory area in this form:

          struct SpriteData
                  {
                  UWORD posctl[2];
                     /* Represents the Hardware-Register
                        'spr[x].pos' and 'spr[x].ctl' } */

                  UWORD Appearance[Height*2];
                     /* This Array contains the Appearance of
                        the Sprites Row for Row defined in
                        two UWORDs. */
                  UWORD Reserved[2] = {0,0};
                   }

                     You must create the SpriteData structure yourself because
                     it does not exist in an include file. */

   UWORD height;
                /*  This variable contains the height of the sprites. Sprites are
                    always 16 pixels (one WORD) wide. */

   UWORD x,y;
                /*  These two variables contain the current position of the
                    sprites. */

   UWORD num;
                /*  This variable contains the number of the hardware sprites
                    (0-7) that are described and changed by the SimpleSprite
                    structure. */

}
```

The `SimpleSprite` structure permits the use of a hardware sprite. Sprites are always 15 pixels wide and can be any desired height.

You, as the programmer, must provide the structures that determine the appearance of the sprites (`SpriteData`). Then use `SimpleSprite.posctldata = (UWORD *) &SpriteData` to pass this data to the `SimpleSprite` structure.

struct TextAttr

```
{
   STRPTR ta_Name;
                /*  This pointer points to the name of the font ("name.font")
                    that you want to open with OpenFont() or
                    OpenDiskFont().*/
```

```
UWORD ta_YSize;
        /*  This is where the height in rows is stored for your font. */

UBYTE ta_Style;
        /*  You use this variable to set a beginning style for the font
            you opened above. */

UBYTE ta_Flags;
        /*  This variable tells you whether or not your font can use
            proportional characters. */
```

This structure is used by the command `OpenFont()` and `OpenDiskFont()`. The variables `ta_Name` and `ta_YSize` are used to try to load a specific font and also find a font that best fits your selected parameters.

---

struct TextFont

This structure is used to access a font opened with `OpenFont()` or `OpenDiskFont()`. It can be linked to a RastPort with `SetFont()` or added to the Systemfontlist with `AddFont()` and removed with `RemFont()`.

---

struct TmpRas

The `TmpRas` structure is used by the fill commands `Flood` and `Area...`. It must be initialized with `InitTmpRas` and linked to a RastPort. Now you can use the `Area...` and `Flood` commands with this RastPort.

The `TmpRas` structure is used to make an area of memory available for use. It must be large enough to store a bit-plane of the largest element you want filled. This is a requirement of the recursive full algorithm.

---

struct View

```
{
  struct ViewPort *ViewPort;
        /*  This is the pointer to the first ViewPort of the View. */

  struct cprlist *LOFCprList;
        /*  This is the pointer to the Copper list created with
            MrgCop(). */

  struct cprlist *SHFCprList;
        /*  This is also a pointer to a Copper list, but this list is only
            used with interlace mode and is required because
            LOFCprList is always used. */
```

```
    short DyOffset, DxOffset;
                /*  These two variables determine the position of your View
                    on the monitor. They are automatically set by InitView()
                    so you don't have to worry about the View positioning.

    UWORD Modes;
                /*  This variable contains the resolution mode of the View. In
                    order to use interlace mode in any particular View, you
                    must set it here. */

}
```

The View structure is the manager of graphic displays and provides the most important link between you and the system.

---

struct ViewPort

```
{
    struct ViewPort *Next;
                /*  Since it is possible to display more than one ViewPort in a
                    View, this is the pointer to the next ViewPort. The
                    ViewPorts must also be linked together. After using
                    InitVPort() this pointer equals zero which means that
                    there are no other ViewPorts. */

    struct ColorMap *ColorMap;
                /*  This pointer determines the colormap of the ViewPort.
                    Because every ViewPort has its own colormap it is
                    possible to set different colors for each ViewPort within a
                    View. */
    struct CopList *DspIns;     /* Display Instructions */
    struct CopList *SprIns;     /* Sprite Instructions */
    struct CopList *ClrIns;     /* Sprite Instructions */
    struct CopList *UCopList *UCopIns;
                /*  These are the pointers to the intermediate or ViewPort
                    Copper list created with MakeVPort(). */

    SHORT DWidth, DHeight;
                /*  These two variables determine the height and width of the
                    ViewPort in pixels. For example, the number of lines used
                    for this ViewPort's resolution (hi-res or lace).*/

    SHORT DxOffset, DyOffset;
                /*  These two variables determine the position of the ViewPort
                    within the View. InitVPort() sets these equal to zero, so
                    you will have to change them. */

    UWORD Modes;
                /*  This is where you set the resolution mode of the ViewPort.
                    You are not limited in choice since HAM, Extra Halfbrite,
                    sprites and all the modes are available. Remember to
```

```
                    always use the highest resolution mode, that you will be
                    using, in your View. */

    UWORD reserved;
    struct RasInfo *RasInfo;
                /*  This pointer makes the link between ViewPort and bit-map
                    through the Rasinfo structure. */

}
```

The ViewPort, that is described by the ViewPort structure, is the window through which you can see your bit-map on your monitor.

---

struct VSprite

```
{
    struct VSprite *NextVSprite;
    struct VSprite *PrevVSprite;
                /*  These two pointers are used to create the GEL list which is
                    sorted using SortGList() by Y and X values (see below
                    for display). */

    struct VSprite *DrawPath;
    struct VSprite *ClearPath;
                /*  These pointers are used to display the bobs in an organized
                    manner. DrawPath writes the bobs into the RastPort.
                    ClearPath is calculated from DrawPath and, as desired,
                    removes the bobs from the RastPort and restores the
                    background. */

    WORD OldY, OldX;
                /*  These two pointers are used to restore the background
                    covered by a bob when the bob moves. By setting the
                    SAVEBACK flag in the vsprite structure for a bob and
                    defining a memory buffer area, you can save the
                    background that is hidden when a bob is displayed. When
                    you move the bob and call DrawGList(), the old
                    background is displayed again. The old bob position is
                    stored in OldX and OldY so the computer knows where to
                    restore the background. */

    WORD Flags;
                /*  This variable determines how the system handles the
                    vsprite. When the vsprite structure is being used to display
                    vsprites then Flags = VSPRITE. However, bobs also use
                    this structure. */

    WORD Y,X;
                /*  These two variables determine the position of the vsprite
                    or bob on the screen. */
```

```
WORD Height;
```
/* This variable sets the number of vertical rows for a vsprite or bob. */

```
WORD Width;
```
/* This variable determines how many words are used to display one row of a bob. A value of one is used for width when using vsprites because they cannot be wider than 16 pixels (= 16 bits = 1 word). */

```
WORD Depth;
```
/* This is where you set the bit-plane depth for your bob. This is actually how many bit-planes you have defined for the bob. Please note that your bob cannot have a depth greater than the RastPort in which it is used. However, a bob can have less bit-planes than the RastPort. */

```
WORD MeMask;
WORD HitMask;
```
/* You use these two variables to determine which collision routine (if any) is executed upon a collision with another GEL. The `MeMask` of one and the `HitMask` of the other GEL are `AND`ed and the result bit determines which collision routine is used. (bit 1 => routine1, bit2 => routine2, ... bit15 => routine15).

When you set bit zero in the `HitMask`, a GEL border collision calls routine zero. */

```
WORD *ImageData;
```
/* This pointer points to the bob/vsprite data that determine their appearance. This data must be stored in chip memory. */

```
WORD *BorderLine;
WORD *CollMask;
```
/* These two pointers point to memory buffer areas, defined by you, for the `BorderLine` and `CollisionMask`. They are also used to detect collisions.

The `BorderLine` contains as many words as the width of a bob or vsprite (vsprites are always one word wide). `BorderLine` is a logical `OR` of all the GEL rows. The `ColMask` is exactly the same size as your GEL, but is only one bit-plane deep. `ColMask` contains a logical `OR` of all plane data. (Both of these buffers must be created by you and are initialized with `InitMasks()`). They must also be located in chip memory. */

```
WORD *SprColors;
```
/* This pointer points to a three `UWORD` memory area that contains the vsprite colors. */

```
    struct Bob *VSBob;
                /*  If you don't set the VSPRITE flag in the VSprite.flags
                    variables, the system assumes that your vsprite structure is
                    being used to describe a bob. This pointer points to the
                    bob structure of the vsprite. */

    BYTE PlanePick;
    BYTE PlaneOnOff;
                /*  These two variables determine which bit-planes are on
                    (PlanePick) for displaying your bobs and which are
                    passive and written with the bobs ImageShadow. */

    VUserStuff VUserExt;
                /*  With the help of this "extension", the user can combine
                    his own data into the vsprite. The VUserStuff function
                    can be defined at any point before the #INCLUDE
                    statements are executed (e.g., #define VUserStuff
                    struct Speed {vx,vy};). */

}
```

The vsprite structure is needed for both types of GELs. Both the vsprite itself and the bob are applications of this structure.

---

struct Window

When you open a new window using `WINDOW = OpenWindow (NewWindow)`, you can write in the window, bypassing the window structure. This is because you have a RastPort available `(Window->RPort)`.

In addition, you can get the window information from Intuition. However, a detailed description of this will go far beyond the scope of this Appendix.

Appendix B: The library-functions

This Appendix provides the C programmer with information about the library routines used in this book.

First you must open the required libraries (with `OpenLibrary()`) before you can use the library routines.

We have organized the following routine listing by library and in alphabetical order.

GfxBase First the `GfxBase` routines:

AddAnimOb (&AnimOb, &AnimKey, &RastPort)

These routines are used to organize, in the GEL list of a specific RastPort, all the bobs for an animation object. This makes it possible for `DrawGList()` to draw the bobs in an orderly manner. The `AnimKey` points to the last added `AnimOb`. `AnimKey` must be equal to zero for the first `AddAnimOb()` call (`struct AnimOb *AnimKey = 0`).

AddBob (&Bob, &RastPort)

This routine adds the specified bob structure for the defined bob (Blitter object) to the GEL list for the selected RastPort. You must do this so the bob can later be drawn with `DrawGList()`.

See: `AddVSprite()`, `DrawGList()`, `SortGList()`, `InitGList()`

AddFont (&TextFont)

This function links the specified `TextFont` structure, which defines your font, with the Systemfontlist.

Once this is done, your font is available to any program you are using.

See: RemFont()

AddVSprite (&VSprite, &RastPort)

This routine adds the specified vsprite structure, which defines your vsprite (virtual sprite), to the GEL list of the selected RastPort. This is needed so that DrawGList() can later display your vsprite.

See: AddBob(), DrawGList(), SortGList(), InitGList()

Pointer = **AllocRaster** (Width, Height)

This function enables you to allocate a memory area for a bit-plane in a size specified by the width and height. After calling this routine, the first word of the memory address is returned in pointer.

See: FreeRaster()

Animate (&AnimOb,&RastPort)

This routine applies to all animation objects that are linked to an AnimOb. Animate calculates the new object positions and resets Timer for the animation components. In a ring animation (AnimComp.Flags = RINGTRIGGER), when Timer for a component is equal to zero the next sequence is activated.

See: AddAnimOb()

AreaDraw (&RastPort, x, y)

This function adds a polygon point to the AreaInfo structure for the specified RastPort. X and Y determine the coordinate of this point in the bit-map.

See: AreaMove(), AreaEnd(), InitArea()

AreaEnd (&RastPort)

This function executes the drawing of a polygon, which is defined with `AreaMove()` and `AreaDraw()`. It also fills the polygon with the current fill pattern controlled by the current `APens`, `BPens`, `OPens` and `Draw` modes.

The `APen` and `BPen` determine the color of the fill pattern. The `OPen` sets the framing color for the polygon.

Remember that you must first initialize an `Area` and `TmpRas` structure in the RastPort where the polygon will be drawn. Unless you do this first, filling an area using the `Area...` function will not function.

See: `AreaDraw()`, `AreaMove()`, `InitArea()`, `InitTmpRas()`

---

error = **AreaEllipse** (&RastPort, XMiddle, YMiddle, XRadius, YRadius)

This function allows you to draw an ellipse in the selected RastPort. The center is at XMiddle,Ymiddle. Xradius and Yradius set the radius for the ellipse. (When XRadius = YRadius a filled circle is drawn. Remember that XRadius and YRadius must be greater than zero.)

`AreaEllipse` is an expansion of the `Area...` function. This means that a filled ellipse is drawn using the current fill pattern. It also means that an `AreaInfo` and `TmpRas` structure must exist in the selected RastPort.

The variable `error` provides information about the vector table, which contains the coordinates for the polygon plot points in the `AreaInfo` structure. It tells you whether or not there is enough room (minimal (2+1)*5 bytes) for the data required for an ellipse (error = 0). If `error` returns a -1 there isn't enough room.

You must call `AreaEnd()` to execute the drawing of the ellipse.

See: `AreaDraw()`, `AreaEnd()`, `AreaMove()`, `InitArea()`, `InitTmpRas()`

AreaMove (&RastPort, x, y)

After building your polygon with `AreaDraw()`, close it with this routine. Any additional `AreaDraw()` functions will start a new polygon with the above coordinates.

See: `AreaDraw()`, `AreaEnd()`, `InitArea()`

AskFont (&RastPort, &TextAttr)

This function initializes, in the selected RastPort, the specified `TextAttr` structure with the values of the current font. You can use this function to check the RastPort for the currently active font.

See: `SetFont()`

Style = **AskSoftStyle** (&RastPort)

This function moves, into the selected RastPort, all the possible font styles that can be selected with `SetSoftStyle`. Each set bit in style represents a font style:

FSF_UNDERLINED	= Bit 0	= 1 (Underlined)
FSF_BOLD	= Bit 1	= 2 (Bold)
FSF_ITALIC	= Bit 2	= 4 (Italic)
FSF_NORMAL	= No Bit set	= 0

See: `SetSoftStyle()`

BitPlanes = **BltBitMap** (&SourceBitMap, X1, Y1, &TargetBitMap, X2, Y2, Width, Height, Minterm, Mask, Buffer)

This function blitters a rectangle from one bit-map into another bit-map. You must specify the source rectangle coordinates from the SourceBitMap (X1, Y1) and the position for the rectangle in the TargetBitMap (X2, Y2). Since the height and width are the same for both rectangles, they are set only once. You must reference both the source and target bit-maps. Remember that these can also be references within the same bit-map.

You also specify the minterm which determines how the source and target rectangles are logically mixed. What the minterms do and which ones exist is discussed in Chapter 16.

The `Mask` parameter determines which bit-planes are blittered. The default value is $ff which means that all bit-planes are referenced. You can also filter out specific bit-planes by clearing the corresponding bit (bit zero for bit-plane zero, etc.).

Buffer points to a memory area that is used when a target and source rectangle overlap in the same bit-map. This buffer must have enough memory to store one row of the rectangle. When you are sure that the two rectangles do not overlap, you can set this pointer to zero.

The result returned by this function is from the "Blit" affected bit-planes (see `Mask`).

Remember that this routine doesn't test whether the target rectangle fits completely within the target bit-map. It is possible to blitter over the edge of a bit-map. This can cause errors which result in less bit-planes being affected than expected. With extreme error conditions you will encounter the familiar Guru Meditations.

See: `ClipBlit()`

BltBitMapRastPort (&SourceBitMap, X1, Y1, &TargetRastPort, X2, Y2, Width, Height, Minterm)

This function performs exactly the same operation as `BltBitMap()`. This time you are moving a rectangle from a bit-map into a RastPort.

The `Mask` and `Buffer` parameters are not required here.

The result returned by this function is TRUE = successful, FALSE = error.

See: `BltBitMap()`

BltClear (&Memory, NumBytes, Flags)

As the name indicates, this function erases a selected memory area. You simply specify the starting address.

The flag parameter determines how the `NumBytes` value is interpreted:

With bit one of the flag parameter set to 1:

The upper 16 bits of `NumBytes` is the number of lines to erase.

The lower 16 bits of `NumBytes` is the number of bytes per line to erase.

With bit one set to zero `NumBytes` represents the actual number of bytes to erase.

Bit zero of the flags parameter determines whether the function is to immediately return to the program or wait until the Blitter has erased everything.

BltTemplate (&Source, BitPosition, Modulo, &RastPort, X, Y, Width, Height)

This function uses the Blitter to transfer data from a packed array into a RastPort. You provide the address of the packed array and the bit position, within this array, where the data read should start.

Modulo specifies the number of bytes for one row in the data array. X, Y, width and height set where and how much data from the array is moved to the RastPort.

ChangeSprite (&ViewPort, &Sprite, &SpriteData)

This function changes the appearance of the specified hardware sprites. Remember that the sprite structure must be initialized already. This ensures that the height and screen position are already set and that the correct hardware sprite (`SimpleSprite.num`) is affected.

The data used by this function have the following format:

```
struct SpriteData
       {
          UWORD posctl[2];
          UWORD Data[Height][2];
          UWORD Reserved[2]; /* = 0,0 */
       }
```

`posctl` represents the sprite hardware registers `spr[x].pos` and `spr[x]ctl`. This allows the user to use `GetSprite()` to reserve both the even and odd numbered sprite and set the `SPRITE_ATTACHED` bit in `posctl[1]` for attached sprites. When

the bit pattern of both sprites overlap, they are merged. This permits the use of 15 colors instead of only three (plus transparent).

See: `GetSprite()`,`FreeSprite()`,`MoveSprite()`

ClearEOL (&RastPort)

This function deletes one row, in the selected RastPort, that has the height of a character in the current font. This deletion starts at the current graphic cursor position (`RastPort.cp_x`, `RastPort.cp_y`).

See: `ClearScreen()`,`Move()`

ClearScreen (&RastPort)

In contrast to the `ClearEOL()` function, which only clears one row, `ClearScreen` deletes the entire RastPort from the current graphic cursor position.

See: `ClearEOL()`,`Move()`

ClipBlit (&SourceRastPort, X1, Y1, &TargetRastPort, X2, Y2, Width, Height, Minterm)

This function performs exactly the same function as `BltBitMap()`. Most of the parameters have the same meaning. But with this function we blit from one RastPort into another and any overlapping problems are handled by the routine.

This function is much better because it is easy to use (you do not have to reserve a buffer) and it prevents you from creating any error conditions. Blitting over the border of a RastPort does not create the problems possible with `BltBitMap()`.

See: `BltBitMap()`,`BltBitMapRastPort()`

CloseFont (&TextFont)

When you have opened a font with `OpenFont()` or `OpenDiskFont()` you also must close it again before ending your

program. Otherwise the memory used by your font is not released back to the system.

See: `OpenFont()`

DisownBlitter ()

This function releases the Blitter from your control and allows other programs to use it again.

See: `OwnBlitter()`

DoCollision (&RastPort)

This routine test all GELs in the GEL list for a RastPort, for collisions and also executes any collision routines you have set up with `SetCollision()`.

However, we must inform you that `DoCollision()` does not function all the time.

See: `SetCollision()`

Draw (&RastPort, x, y)

This is one of the most important graphic functions. `Draw` allows you to draw a line from the current graphic cursor position to the selected X/Y coordinate in the specified RastPort.

See: `Move()`

DrawEllipse (&RastPort, XMiddle, YMiddle, XRadius, YRadius)

This function allows you to draw ellipses or, with XRadius = YRadius, circles. These are not filled as with `AreaEllipse`; only the outline is drawn.

Remember that XRadius and YRadius must contain values that are greater than zero.

DrawGList (&RastPort, &ViewPort)

`DrawGList()` performs two functions. First it can draw the bobs, from the GEL list, in the selected RastPort. Second, it can link the vsprites, from the ViewPort Copper list, with the selected ViewPort.

However, vsprites are not drawn after calling `DrawGList()`. First you have to use `MakeVPort()` or `MakeScreen()` to set up the new Copper list. Then you use `MrgCop()` and `LoadView()` or `RethinkDisplay()` to display them.

See: `MakeVPort()`, `MrgCop()`, `LoadView()`, `MakeScreen()`, `RethinkDisplay()`

Flood (&RastPort, Mode, x, y)

This function enables you to fill an area in a RastPort. All the pixels around the area from the specified X/Y position are tested and filled according to the mode selected. Mode = 0 tests for the color of the `OPen` and Mode = 1 tests for the color of the starting pixel.

Any pixels around your starting point that don't meet the requirements of either mode are filled with the current fill pattern. When the pixel meets one of the mode criteria it is not changed.

When all pixels within a closed border meet the mode criteria and are already filled, the fill is canceled.

Remember that for the `Flood()` function a `TmpRas` structure must be initialized and must have enough memory to be used by the `Flood()` function.

See: `InitTmpRas()`

FreeColorMap (&ColorMap)

You use this function to release the memory, which was reserved through `GetColorMap()` for the colormap structure, back to the system.

See: `GetColorMap()`

FreeCopList (&CopList)

Call this routine in order to release a single intermediate Copper list for a ViewPort (struct CopList), which was created by using `MakeVPort()`. `FreeVPortCopList()` uses this routine to release all intermediate lists from a ViewPort.

See: `FreeVPortCopList()`

FreeCprList (&cprlist)

Use this function to release the memory reserved, through `View.LOFCprList` and `View.SHFCprList`, for the hardware Copper list. These hardware Copper lists are calculated from the individual ViewPort Copper lists by using `MrgCop()`.

See: `MrgCop()`

FreeRaster (&BitPlane, Width, Height)

This function produces the opposite effect of `AllocRaster()`. It releases all reserved bit-plane memory, that was allocated with `AllocRaster()`, back to the system. This makes the memory available to other programs again.

`BitPlane` points to the memory area that you reserved with `AllocRaster`. Width and height must be the same for free and `AllocRaster()`, otherwise it is possible to release too much or too little memory.

See: `AllocRaster()`

FreeSprite (SprNumber)

With `FreeSprite` you can release a sprite, that was reserved specifically for your use through `GetSprite()`, back to the system. This makes the sprite available again for full system use (especially the vsprites).

You just provide the number of the sprite that was reserved with `GetSprite()`.

See: GetSprite(),ChangeSprite(),MoveSprite()

FreeVPortCopLists (&ViewPort)

With this function you can release, for the specified ViewPort that you generated with MakeVPort(), all the Copper lists for color, display, sprites, etc.

See: MakeVPort(),FreeCopList()

---

*ColorMap = **GetColorMap** (NumberColors)

This function creates a complete ColorMap structure. All of the required ColorMap structure variables are initialized and enough memory is reserved for the specified NumberColors. When sufficient memory is not available for the new ColorMap structure or color buffer, a ZERO (0) is returned by the routine.

You must link this ColorMap structure to the ViewPort with ViewPort.ColorMap = ColorMap (before using MakeVPort() to calculate the Copper list for the ViewPort).

See: FreeColorMap(),LoadRGB4(),GetRGB4(), SetRGB4()

Color = **GetRGB4** (&ColorMap, ColorRegister)

This function enables you to retrieve the UWORD that determines the color of the selected color register. The 16 bit UWORD which is returned is divided as follows: bits 0-3 are the blue component, bits 4-7 the green, and bits 8-11 the red.

See: SetRGB4(),LoadRGB4()

SprNumber = **GetSprite** (&SimpleSprite, getSprNumber)

This function allows you to reserve any hardware sprite for your personal use. You provide the SimpleSprite structure, that the sprite will be associated with, to the function. Also select the desired number (0-7) (if it doesn't matter a -1) of the reserved sprite.

The number of the reserved sprite is returned in SpriteNumber. If the selected sprite is not available, SpriteNumber contains a -1.

See: `FreeSprite()`,`MoveSprite()`,`FreeSprite()`

InitArea (&AreaInfo, &Buffer, NumCoord)

This function initializes the selected `AreaInfo` structure for you. The structure contains the specified Buffer and the NumCoord+1 coordinates required for `AreaDraw()` and `AreaMove()`. The buffer should contain at least five times the number of bytes that are required for the polygon coordinates which are used with `AreaMove()` and `AreaDraw()`.

Declaring the buffer as a `char` array is the easiest method. Since programs created with the compiler automatically release arrays, you don't have to release the used buffer areas again.

See: `AreaDraw()`,`AreaMove()`,`AreaEnd()`

InitBitMap (&BitMap, Depth, Width, Height)

This routine initializes the specified bit-map structure. The depth (number of bit-planes), width (in pixels), and the height (in rows) is easily set.

See: `AllocRaster()`

InitGels (&Begin, &End, &GelsInfo)

This function initializes the selected `GelsInfo` structure. You need this structure in order to display vsprites and bobs in an organized manner.

All the graphic elements (=GEL) are linked in a list through the vsprite structures (`&Begin`, `&End`). `&Begin` and `&End` mark the beginning and end of this GEL list.

After you have used `InitGels()` to initialize the `GelsInfo` structure, then simply make them available to your RastPort (`RastPort.GelsInfo = &GelsInfo`). Once this is accomplished, you can start displaying your bobs and vsprites.

See: AddVSprite(),AddBob()

InitMasks (&VSprite)

You initialize the BorderLine and CollMask for the selected vsprite structure with this routine. The memory buffer for both masks must be reserved already and stored in the variables VSprite.BorderLine and VSprite.CollMask.

InitRastPort (&RastPort)

You use this routine to initialize the specified RastPort structure. This RastPort structure is the most important connection between user and bit-map.

The RastPort contains, for example, the color of the current foregroundpen (APen). It also contains a pointer to the bit-map where the graphic functions will be executed and many other important variables and pointers used by the graphic functions.

TmpRas = **InitTmpRas** (&TmpRas, &Buffer, Buffersize)

When you use the Area...() or Flood() functions within a RastPort, an extra memory area is required for these functions. This buffer must be at least as large as the object being filled. Because of the recursive algorithm used by fill, extra memory is required as work space.

This buffer only requires one bit-plane which is as large as the largest element. To ensure that you have enough, it is best to reserve one complete bit-plane for temporary raster.

Now simply pass the initialized TmpRas structure to your RastPort. There are two possible ways to do this. You can either use the result returned by the initialized TmpRas structure (RastPort.TmpRas = InitTmpRas (...)) or pass the initialized TmpRas structure yourself (RastPort.TmpRas = &TmpRas).

InitView (&View)

This function is used to initialize the specified View structure. All variables and pointers for the View structure are first set to zero. Then the `DxOffset` and `DyOffset` variables are set so that the View is positioned approximately two centimeters below the top right monitor border. This assumes that the monitor is properly adjusted.

See: `MrgCop()`, `MakeVPort()`

InitVPort (&ViewPort)

This function is used to clear all variables and pointers for the selected ViewPort structure. This is done so that the individual Copper lists for this ViewPort will be correctly calculated with `MakeVPort()`. When a pointer to a ViewPort Copper list is not equal to zero, `MakeVPort()` assumes that a ViewPort Copper list exists and does not calculate a new one.

Undefined Copper lists could be created without this `MakeVPort()` feature.

See: `MakeVPort()`

LoadRGB4 (&ViewPort, &Colorpalette[0], Colorentry)

This routine allows you to load new color values, `Colorentry`, from the `Colorpalette` (`UWORD Colorpalette[Colorentry];`) into the colormap for the selected ViewPort.

Using this routine is an easy way to change all the colors of a ViewPort at one time. However, the change is not immediate. First you have to calculate a new color Copper list for the ViewPort. You do this with `MakeVPort()`, `MrgCop()`, and then `LoadView()`. When using Intuition screens substitute `RemakeDisplay()` for `LoadView()`.

See: `SetRGB4()`, `GetRGB4()`

LoadView (&View)

After calling this function, the selected View and all ViewPorts it contains are displayed on the screen. You must first use `MakeVPort()` for each ViewPort that will be displayed in the View. `MakeVPort()` creates the ViewPort Copper lists that `MrgCop()` merges into a single View Copper list.

The View Copper list starting address is written by `LoadView()` in the necessary hardware registers (`cop1lc`, `cop2lc`).

See: `InitView()`, `InitVPort()`, `MakeVPort()`, `MrgCop()`

MakeVPort (&View, &ViewPort)

This routine calculates the intermediate or ViewPort Copper lists for the selected ViewPort of the specified View. For example, there are ViewPort Copper lists for color display for all colors which are written through the Copper to the hardware registers. These color Copper lists are calculated from the ViewPort colormap. When the color list pointer is equal to zero, the Default colormap is used to calculate the ViewPort color Copper list.

These lists also contain the various resolution modes for the ViewPorts (many ViewPorts in one View can have different resolutions).

See: `MrgCop()`, `FreeVPortCopLists()`

Move (&RastPort, X, Y)

You use this function to position the graphic cursor for the selected RastPort to the specified position (`RastPort.cp_x = x; RastPort.cp_y = y;`).

At this position you could print text or draw a line to another position.

MoveSprite (&ViewPort, &SimpleSprite, X, Y)

This function lets you position the selected hardware sprite within the selected ViewPort when ViewPort is not equal to zero. Otherwise, the sprite will be positioned relative to the View.

X and Y specify the new position. Remember that the position values in hi-res or lace ViewPorts must be multiplied by two so that the sprite actually moves.

See: `ChangeSprite(),GetSprite(),FreeSprite()`

MrgCop (&View)

This function calculates the View hardware Copper list from the ViewPort Copper lists created with `MakeVPort()`. Then this list is used by the hardware (`cop1lc`, `cop2lc`).

Any change to the colortable or the hardware sprite positions requires a newly calculated Copper list. This happens automatically when the hardware sprites are moved. When changing the colorpalette with `LoadRGB4()` you must update the Copper list.

See: `MakeVPort(),FreeCprList()`

---

*TextFont = **OpenFont** (&TextAttr)

This function searches through the Systemfontlist for a font described by the `TextAttr` structure. When a font cannot be found to match the named `TextAttr` structure, a value of zero is returned. However, when a font is found but has different styles or sizes, a font that best meets the specifications is selected.

`OpenFont()` increases user choices when accessing the Systemfonts. `OpenFont()` and `CloseFont()` must maintain a balance in order to guarantee a clean memory. When a font is no longer required you can remove it from memory with `RemFont()`.

See: `Closefont(),AddFont(),RemFont(),`
`OpenDiskFont(),AvailFonts()`

OwnBlitter ()

This function informs the Blitter that it is only able to work for you. Other tasks (programs) that are running at the same time, can no longer use the Blitter.

After `OwnBlitter()`, you should use `WaitBlit()` to allow the Blitter to complete any current processing.

See: `DisownBlitter()`,`WaitBlit()`

PolyDraw (&RastPort, NumCoords, &PixelArray)

This routine enables you to draw into the selected RastPort by using coordinates from an array you created. Lines are drawn from pixel to pixel as plotted from the array.

The pixels are stored in an integer array (`int PixelArray [NumCoords][2]`) with each coordinate consisting of two array elements. The first element is the X and the second is the Y coordinate for the stored polygon pixels.

`NumCoords` specifies the number of pixels that will be drawn from the pixel array.

Colorregister = **ReadPixel** (&RastPort, X, Y)

This function allows you to read the color value of a specific pixel at the coordinates you selected.

When your coordinates are outside the selected RastPort, a value of -1 is returned in `Colorregister`.

See: `WritePixel()`,`SetAPen()`

RectFill (&RastPort, X1, Y1, X2, Y2)

With this function you can fill any desired rectangular area within the selected RastPort. The current fill pattern, `DrawModes`, `APens`, etc. affect how your fill is performed.

You just specify the top left corner (X1, Y1) and the lower right corner (X2, Y2) of the rectangle you want filled. Make sure that the lower right corner really is lower and to the right of the upper left corner. When your coordinates are incorrect the computer can crash because your memory will be filled without the proper controls.

This function does not require a `TmpRas` structure like the `Flood()` and `Area...()` functions.

RemFont (&TextFont)

This function removes the selected font from the system font list. The font remains available to any programs currently using it. After all programs signal, with `CloseFont()`, that this font is no longer required, the font is physically removed from memory.

After you have used `RemFont()` a program that has used `CloseFont()` cannot access the font again.

See: `AddFont()`, `OpenFont()`, `CloseFont()`

RemIBob (&Bob, &RastPort, &ViewPort)

This function immediately removes the selected bob from the GEL list for its RastPort and deletes it from the RastPort.

See: `DrawGList()`, `SortGList()`, `InitGList()`

RemVSprite (&VSprite)

This routine deletes the selected vsprite from the current GEL list. However, the sprite does not immediately disappear from the screen. This occurs only after the calls to `SortGList()`, `DrawGList()`, `MakeVPort()`, `MrgCop()`, `LoadView()`.

See: `DrawGList()`, `SortGList()`

ScrollRaster (&RastPort, DeltaX, DeltaY, X1, Y1, X2, Y2)

This function scrolls the rectangle, specified by X1, Y1, X2, Y2 (see `RectFill`), for DeltaX pixels and DeltaY rows. Positive delta values scroll up and left. Negative delta values scroll right and down. Combining negative and positive 'Delta' values allows you to scroll in any desired direction.

The areas uncovered by the scrolled rectangle are filled with the color of the BPen.

Please note that this routine is not very fast and can cause some flickering during the scrolling.

ScrollVPort (&ViewPort)

After you have changed the RxOffset and RyOffset values in the RasInfo structure of the ViewPort, use ScrollVPort() to display a different portion of the bit-map. ScrollVPort calculates the new Copper list for the new bit-map position.

This routine is not fast enough to prevent some flickering on the screen. Because of this, you should calculate the new Copper list since this can be performed as a background process.

SetAPen (&RastPort, Colorregister)

This function allows you to select, in the selected RastPort, a new color for the APen. Colorregister selects the number of the color register in the ViewPort colormap.

See: SetBPen()

SetBPen (&RastPort, Colorregister)

Just like SetAPen, the color of the BPen is changed in the selected RastPort. Again, Colorregister specifies the number of the color register to be used from this point on.

APen and BPen functions differently depending on which drawing mode you are using. With JAM2 mode, the BPen functions as the background color pen. For example, text would be highlighted with the BPen. The BPen has no effect in other draw modes.

See: SetAPen()

SetCollision (Number, Routine, &GelsInfo)

After comparing the `MeMask` and `HitMask`, this function sets the collisions routines that are used for collisions between GELs or a GEL with a border.

With GEL to GEL collisions you must pass both vsprite structures to your collision routine. The first vsprite structure passed represents the upper left positioned GEL (bob or vsprite).

With border collisions, you pass the involved vsprite structure and a flag to your routine. The flag contains the code for the border (`TOPHIT`, `BOTTOMHIT`, `LEFTHIT`, `RIGHTHIT`) the GEL collided with.

See: `DoCollision()`

SetDrMd (&RastPort, Mode)

Use this function to set the `DrawMode` for your RastPort. This sets up how lines are drawn, how pixels are set and how text is displayed. The following modes exist:

`JAM1 (0)`: Only the `APen` color is used for drawing.

`JAM2 (1)`: The `BPen` color is used for the background.

`COMPLEMENT (2)`: Pixels to be drawn are first `XOR`ed with the existing pixels (by bit-plane).

`INVERSVID (4)`: Pixels are inverted before being drawn; set pixels become unset and vice versa (this also is performed by bit-plane).

The modes `COMPLEMENT` and `INVERSVID` only function with `JAM1` or `JAM2`.

SetFont (&RastPort, &Font)

`SetFont()` makes a font available to the selected RastPort. This font should have already been opened with `OpenFont()` or `OpenDiskFont()`. Any text you now output with `Text()` will appear in the new font.

See: `OpenFont()`, `CloseFont()`

SetRast (&RastPort, Colorregister)

This function sets all pixels, in the selected RastPort bit-map, to the color of the color register from the ViewPort's colormap for this RastPort.

When `Colorregister` equals zero, then all pixels of the ViewPort's colormap are cleared to color zero (= background color).

SetRG4CM (&ColorMap, Colorregister, Red, Green, Blue)

This function works similar to `SetRGB4`. However, the color change is not immediately visible. This happens only after you calculate the Copper list again with `RemakeDisplay`.

The color change first occurs in the selected colormap before it is allowed to be visible.

Like `LoadRGB4`, the colormap is initialized with new values in the background.

See: `LoadRGB4()`

SetRGB4 (&ViewPort, Colorregister, Red, Green, Blue)

This function allows you to change the color of one color register in the specified ViewPort. You pass the number of the color register and the new red, green and blue components (0-15) for the new color value.

`SetRGB4()` also changes the color entry in the ColorMap structure of the ViewPort and in the Copper list. This color change is immediately visible on the screen.

See: `LoadRGB4()`, `GetRGB4()`

NewStyle = **SetSoftStyle** (&RastPort, Fontstyle, Style)

This function sets your desired, algorithmically generated font style in the selected RastPort. You set the desired style in `Fontstyle` (see `AskSoftStyle()`) and use the value returned by `AskSoftStyle()` in `Style`. `AskSoftStyle()` provides the available styles that can be used for the `Style` variable. For example, it is possible that a font is already italic and after `AskSoftStyle()` provided that information, it would be useless to try to make the font italic.

`SetSoftStyle()` changes a font only when the style-bits are also set in `Style`. After the change, `NewStyle` contains the font generated by `SetSoftStyle()`.

See: `AskSoftStyle()`

SortGList (&RastPort)

This routine sorts the GEL list of the selected RastPort. The vsprite structures used for positioning vsprites and bobs are sorted by their Y and X coordinates. The sort is performed first by the Y and then by the X coordinate.

After using `SortGList()` you use `DrawGList()` to display the GEL list again.

See: `DrawGList()`

Text (&RastPort, &"String", NumCharacters)

This routine lets you display, at the graphic cursor position, any desired text in the selected RastPort. You simply provide the starting address of the output string and the number of characters to be displayed.

See: `TextLength()`

Length = **TextLength** (&RastPort, &"String", NumCharacters)

This function has the same parameters as `Text()`. The length in pixels for the text width is returned and the RastPorts currently active font is used to display the text.

See: `Text()`, `SetFont()`

Position = **VBeamPos** ()

This function returns the current vertical position of the electronic raster beam in "Position". The values that are returned fall between zero and 255 on PAL systems. Because the electronic beam actually scans 262 lines on PAL systems, the positions between 256 and 262 are returned as values between zero and 6.

After your program has received it, the value returned by `VBeamPos()` is usually useless. Due to the Amiga's multi-tasking capabilities, the millisecond reaction time, required by your program to react, can be used up by another program or task.

WaitBlit ()

This function pauses your program until the Blitter has completed its task. Be careful because a processor error can cause `WaitBlit()` to return early before completing the desired task. Your program is then able to continue its tasks before they should occur. Most often, this error occurs when you are using hi-res with four bit-planes.

WaitBOVP (&ViewPort)

This function pauses your program until the electronic beam reaches the last row of the selected ViewPort (Bottom Of ViewPort).

See: `WaitTOF()`

WaitTOF ()

This function is similar to `WaitBOVP()` except that the pause doesn't happen until the end of the ViewPort. `WaitTOF()` waits for the electronic beam to reach "Vertical Blank" (Top Of Frame) or the start of a new scan after completing any cyclic routines (Interrupts).

See: `WaitBOVP()`

WritePixel (&RastPort, X, Y)

This function sets a pixel, at coordinates X and Y, in the `APen` color in the selected RastPort. Naturally, the current drawing mode of the RastPort is used.

See: `ReadPixel()`

`DiskFontBase` Now to the `DiskFontBase` functions (you must first open the library "diskfont.libraries"):

Error = **AvailFonts** (&Buffer, BufferSize, Typ)

This function provides a complete list of all available system fonts. This list is written into the memory area (`&Buffer`), which contains the `BufferSize` bytes that you selected.

The first thing stored in the buffer is the `AvailFontsHeader`. This contains the number of font entries that are stored (in `AvailFontsHeader.afh_NumEntries`). Then the various `AvailFonts` entries, which contain the type of font (either in the Systemfontlist (`AFF_MEMORY`) or on disk (`AFF_DISK`)) and the more descriptive `TextAttr` structure are stored.

The `TextAttr` structure can be used to open a font. However, this must be done by type (`AFF_MEMORY / AFF_DISK`) in order to open either one with `OpenFont()` or `OpenDiskFont()`.

The `AvailFonts()` parameter `Typ` determines whether the Systemfontlist (`AFF_MEMORY`), SYS:Fonts (`AFF_DISK`) or both are searched for available fonts.

Please note that `AvailFonts()` does not verify the `TextAttr` structure contents for SYS:Fonts from disk. When you open fonts using `OpenDiskFont()` with the returned `TextAttr` structure of `AvailFonts()`, you can encounter such undesired results as a system crash.

The results returned by `AvailFonts()` provide information on whether or not enough memory is available to store all the `AvailFonts` structures. When error equals zero, there is enough memory. However, when error is not equal to zero the number of additional bytes that are required to contain all the `AvailFonts` structures is returned instead.

---

*TextFont = **OpenDiskFont** (&TextAttr)

This function opens a font described by the `TextAttr` structure. The SYS:Fonts directory is searched for a font that best matches the `TextAttr` structure in style and size. If a matching font in the `TextAttr` structure cannot be found, a value of zero is returned.

See: `OpenFont()`

`IntuitionBase`

The following are the `IntuitionBase` routines (IntuitionBase is the pointer to the Intuition library that you open with `IntuitionBase = (struct IntuitionBase *) OpenLibrary ("intuition.library".0).`

CloseScreen (&Screen)

Use this function to close a screen that was opened earlier with `OpenScreen()`. The memory used for the bit-map and all Intuition screen structures (ViewPort, RastPort, etc.) is released.

When you close the last user-opened screen, the workbench screen is automatically active again. `CloseScreen()` does not check whether all windows for a screen have been closed. If any windows are still open, a system crash will occur.

See: `Openscreen()`

CloseWindow (&Window)

This function closes a window you have opened with `OpenWindow()`. Make sure that you close all the windows in a screen before attempting to close the screen window.

See: `OpenWindow()`

DisplayBeep (&Screen)

This function blinks the screen by quickly changing the background color of the screen. This technique is used in order to make the user aware of small errors.

By using a zero instead of a screen address you can blink all the Intuition screens. However, this should only be used to indicate a rather extreme user error.

MakeScreen (&Screen)

This function executes `MakeVPort()` for the ViewPort of the specified screen. You can make changes to the Intuition screens' ViewPorts (such as color changes with `LoadRGB4()`) and inform the Copper afterwards.

See: `MakeVPort()`

ModifyIDCMP (&Window, IDCMPFlags)

This function is used to change the IDCMP flags (Intuition Direct Communication Message Ports) for a window. These flags are used for various messages that are sent by Intuition to your window. However, it is impossible to explain all the possibilities of the IDCMPs in this Appendix.

OpenScreen (&NewScreen)

This function opens an Intuition screen that is further described by the selected `NewScreen` structure. This function creates a ViewPort and all the other required structures (bit-map, RastPort, etc.). The new screen is linked to the Intuition View and then displayed.

See: `CloseScreen()`

RemakeDisplay ()

This function calculates a new Copper list for all Intuition screens. This means that for each screen, a `MakeScreen()` and then a `RethinkDisplay()` is called. When using this function, you should disable multi-tasking for a short time.

See: `MakeScreen()`, `RethinkDisplay()`

RethinkDisplay ()

This function executes `MrgCop()` and `LoadView()` for the Intuition View. Multitasking is turned off shortly while this is executing.

See: `MakeScreen()`, `RemakeDisplay()`, `MrgCop()`, `LoadView()`

DosBase Here are the `DosBase` routines (Please note that the `DosBase`, just like the `ExecBase`, don't have to be opened because in these C programs they are always open).

Delay (Time)

This routine pauses your program "Time" * 1/60 seconds. Other running programs (tasks) are not paused as with a pause using `for()`.

Exit (ReturnValue)

This function allows you to exit a program loop at any time. When you run your program as an "Overlay-Process", the "ReturnValue" is passed to your "Main-Process". We did not use the program overlay technique in our example programs.

ExecBase The following are the `ExecBase` routines:

*Memory = **AllocMem** (NumBytes, Use)

This routine is used to declare NumBytes of memory for your use. You must also specify what memory you require. When you use `MEMF_CHIP (MEMF = MemoryFlag)` the memory is allocated from the lower 512 KByte area. This is especially important since this memory must be used by all the Amiga processors (for example, for bit-planes, etc.).

With `MEMF_PUBLIC` you use a memory area from anywhere. This can be an area above 512 KByte if you have more than 512K.

`MEMF_CLEAR` allows you to immediately clear the selected memory area. You will receive the address of a completely cleared memory area.

See: `FreeMem()`

CloseLibrary (&BasePointer)

Use this function to close a library you opened earlier with `OpenLibrary()`. Provide the address pointer, that you received when using `OpenLibrary()`, in "BasePointer".

Please make sure you close all open libraries before exiting your programs.

See: `OpenLibrary()`

FreeMem (&Memory, Size)

You use this routine to release a memory area that was reserved. Simply specify the starting address of the memory area and the number of bytes reserved.

see: `AllocMem()`

Message = **GetMsg** (&Port)

By using this function you can intercept messages from Intuition. This function waits until a message is sent.

See: `ReplyMsg()`

OpenLibrary (&LibName, Versionnumber)

Use this function to open the selected library which gives you access to specific operating system routines.

For example, to use the graphic functions, open the graphic library with `GfxBase = (struct GfxBase *)OpenLibrary ("graphics.library",0).GfxBase` is the pointer to the requested library and must be used with `CloseLibrary()` at the end of your program.

When you don't know the actual version number of the library, use zero, which uses the current library version. However, when a function is available only in a specific version of a library, you should quickly exit your program if the library is not found.

ReplyMsg (&Port)

Use this function to tell Intuition, for example, that you have received and processed its messages.

Appendix C: The hardware registers

This Appendix provides an overview of the Amiga hardware registers. These registers are useful with Copper programming because they can be easily changed through the Copper.

Use the `CMOVE()` instruction with the register number that you want to change. If you want to change a register by using the 68000, you should only write to specific memory areas.

The hardware registers are located in a memory area starting at $DFF000. This means that you must use this address as an offset and then add the value of the specific register address in order to use the 68000 to access it.

To access the hardware registers with C, use the `Custom` structure that you create with `extern struct Custom custom`. As with the Copper or 68000, you can access the individual registers by using `custom.NAME`.

The following list describes hardware registers in detail. We have provided the register names as you would define them in your include file `hardware/custom.h`. We also show the offset for each register from $DFF000. So, both C and machine language programmers have the background information on the registers and how to use them.

adkcon	$09E	Audio and disk control (write)
adkconr	$010	Audio and disk control (read)

As shown, you access these registers differently. You can either write (`adkcon`) to one, or read (`adkconr`) from the other.

The bit structure of both registers is the same. However, even though bit 15 is not affected by read operations, it is very important for write operations.

Bit 15 SET/CLR:

When writing a word with this bit set, all set bits (14-0) are written into the register. Any unset bits in the word being written do not change the corresponding bit already in the register.

If bit 15 is unset in the word being written, all set bits of the written word clear any matching set bits in the register. Again, any unset register bits are unaffected.

In order to clearly explain this, we have provided the following short example: When writing the binary value %1111111111111110 all bits in the register are set except bit zero. With a value of %0111111111111110 all bits in the register are cleared except bit zero. This type of register control appears very often. Whenever a CLR/SET bit is used, the above rules will apply.

Bits 14,13 PRECOMP1/2:

These bits set the Precomp for disk operations. The value %00 in both registers selects zero ns (nanoseconds). A value of %01 selects 140 ns, %10 selects 280 ns, and %11 selects 560 ns.

Bit 12 MFMPREC:

This bit selects the read format for a disk. If it is set, this bit selects the normal MFM format. When it is unset, the GCR format is selected as for Apple or Commodore 64 disks.

Bit 11 UARTBRK:

When this bit is set, all lines of the RS-232 interface are set to zero (Universal Asynchronous Receiver/Transmitter Break).

Bit 10 WORDSYNC:

Setting this bit starts the data transfer over the DMA channel and each word read is synchronized with the word in dsksync.

Bit 9 MSBSYNC:

This bit determines whether the high bit of a word is synchronized in GCR format.

Bit 8 FAST:

This bit allows you to set the timing for reading a bit from disk. Setting this bit selects two microseconds, which is the required timing for the MFM format. A value of zero for this bit sets the timing to four microseconds, which is the correct timing for GCR type formats (Apple, Commodore 64).

Bits 7-4 ATPER3-0:

These bits allow you to modulate the sampling rate of the individual audio channels. Setting ATPER3 stops the sound channel output. By setting ATPER2, you can modulate the sampling rate of audio channel three with audio channel two. Setting ATPER1 modulates the sampling rate of audio channel two with audio channel one. ATPER0 sets the modulation sampling rate between audio channels one and zero.

Bits 3-0 ATVOL3-0:
These bits determine the volume modulation of the individual audio channels. The pattern used is the same as `ATPER3-0` except that the volume is modulated.

---

aud[X].ac_ptr $0A0 Audio channel X start address

The address for the data that will be sent over the sound DMA channel (direct memory access = without use of the 68000) is stored here.

This is actually a register pair where the data addresses' lower 15 bits are stored in $0A2 and the upper three bits are stored in $0A0. Based on the custom declaration in the include file, the `*ac_ptr` is already reserved by the compiler as a long word.

To directly access this register pair from the C language, you must use an index value between zero and three for the corresponding audio channel (`aud[X]`).

Machine language programmers must always add X*12 words to access a specific sound channel register. This register pair and the following five registers represent a block divided into four parts, one after the other. At the end of this Appendix we have provided a table showing all the register addresses relative to $DFF000.

Remember that here you can only use 18 bits for addressing. The sound chip, as with the Blitter, only uses the lower 512 KBytes of memory.

---

aud[X].ac_len $0A4 Audio channel X length

This register contains a count of the number of words to be sent through the DMA channel for sound generation. You can set the starting address for these words in the register that is described above.

---

aud[X].ac_per $0A6 Audio channel X sampling frequency

In this register you specify how many bytes per second are used for sound generation. Since the sound DMA channel is synchronized with the screen DMA channel, only two bytes per raster row can be made audible. Theoretically it is possible to make two (bytes) * 262.5 (raster rows) * 59.94 (screen displayed) = 31468.5 bytes per second audible in PAL systems.

Because the DMA controller is responsible for other tasks besides sound generation, it is only possible to sample 28867 bytes per second.

This means that you could send one byte through the sound channel in 1/28867 seconds = 34.642 microseconds. This provides the value for

this register: `ac_per` = 34.642 microseconds / Num_Microseconds for one scan.

The hardware determines the limits for the scan per timing cycle. Since the timing cycle is 280 nanoseconds or 0.28 microseconds, a value of 124 (34.643 microseconds / 280 nanoseconds) is the smallest possible value for this register. If you select a smaller value, the DMA channel can no longer function properly.

---

aud[X].ac_vol $0A8 Audio channel X volume

Use this register to set your audio volume level for channel X. Only the lower five bits are essential to volume.

The maximum volume level is 64; setting this register to zero turns off your sound.

---

aud[X].ac_dat $0AA Audio channel X Data

This register is the audio DMA data buffer. It contains two bytes in two compliments which are sequentially sent through the sound hardware.

The DMA controller automatically writes the values read from `ac_ptr` into these registers. To create sound without using the DMA channels, you can write directly to these register by using the 68000. When doing this, make sure that you don't enable DMA access at the same time because this will disturb the interrupt timing.

---

bltXpt $050 Blitter pointer

This register pair pointer contains an 18 bit pointer to the DMA data that the Blitter will act upon. X represents a, b, c and d, which are the three sources (a, b, c) and the target (d) for the Blitter operation (Blits).

The data for a blit operation is read from `bltapt`, `bltbpt` and `bltcpt`; it is operated on and stored at `bltdpt`.

When the Blitter operation is complete, the pointer contains the last data address (plus increment and modulo) for the data written and read. To include the target (d) in the blit operation, one of the sources (a, b, or c) must contain the same address as the target (d).

The Blitter is not only responsible for blitting, but also for drawing lines. Because this additional task is integrated into this processor, it is easy to understand why the Amiga can draw lines so quickly.

How the `bltapt`, `bltbpt`, `bltcpt` and `bltdpt` registers affect line drawing is not relevant here because the `Draw()` instruction was designed for this function.

---

bltXmod $064 Blitter modulo X

This register contains the modulo for the Blitter sources (a, b and c) and the Blitter target (d). Since it is possible to use different sized bit-maps, which we can blit between, we must know the size of each bit-map that the Blitter will work with. A blit operation is not limited to only one bit-map.

You must provide the width for each bit-map, that is used for the blit operation, as the modulo parameter. This is how a rectangle to be blitted is transferred from one bit-map 1:1 to another.

Remember that the Blitter can only operate on data by bit-plane. This means that when you control the Blitter yourself and blit a rectangle from one bit-map to another, you must provide, with `bltxpt`, the rectangle position for each bit-plane. The modulo is read last and added to the written word to locate the start of the next row.

Commodore has provided some very powerful instructions that fully use the Blitter (`BltBitMap()`, `BltTemPlate()`, `ScrollRaster()`).

As with drawing lines, the modulo registers also have a specific purpose. However, as we said earlier, the `Draw()` function does all the work for you.

---

bltafwm $044 Blitter First word mask for source A
bltalwm $046 Blitter Last word mask for source A

In order to blit a rectangle that starts in the middle of a word instead of at a word address, the first and last word of a row to be blitted (the source A) is ANDed with a specific value. The remaining bits are changed, reworked and written into the row.

---

bltXdat $074 Blitter X data register

Similar to sound generation, the individual words from source a, b and c are read here before they are logically merged. This action is performed by the DMA channel. After the blit of these three words, the resulting word is written, by the DMA channel, to `bltddat` and then to the target. The entire blit operation is reduced to operating on three words.

bltcon0	$040	Blitter control register 0
bltcon1	$042	Blitter control register 1

These two registers are used to control the Blitter operation (blits). There are two modes, `Area` and `Line`, that are selected through bit zero of register `BLTCON1`:

The Area mode bltcon0

Bits 15-12 ASH3-0:

If you have ever moved a bob by pixels, you probably wondered how this was possible since the Blitter can only access word addresses.

By using the bits `ASH3-0`, you can select how many pixels the data from source A is rotated before being written to the target.

Bits 11-8 USEx:

These bits determine which source the Blitter can access. If you have only two sources, it would be a waste of time to access all three possible sources. Besides, there wouldn't be any data in source C.

With the `USEx` bits you can select your source; X stands for the a, b, c and d. Make sure you activate the target, otherwise the data will have no place to go.

Bits 7-0 LF7-0:

These bits determine the type of logic operation that will be performed, by the Blitter, on your source data. The bit patterns written here are also called minterms, which you learned about in connection with the Blitter instructions. Only two sources are considered, b and c.

However, there are three sources you can use. What happens with each source is determined by the individual bits `LF7-0`:

Bits: LF 7 6 5 4 3 2 1 0

Minterm: ABC $AB\overline{C}$ $A\overline{B}C$ $A\overline{BC}$ $\overline{A}BC$ $\overline{A}B\overline{C}$ $\overline{AB}C$ $\overline{ABC}$

ABC: D = A and B and C

$AB\overline{C}$: D = A and B and !C

$A\overline{B}C$: D = A and !B and C

$A\overline{BC}$: D = A and !B and C

$\overline{A}BC$: D = !A and B and C

.
.
.

The Area mode **bltcon1** has the following bit pattern in Area mode:

Bits 15-12 BSH3-0:
These bits have the same meaning as `ASH3-0` with `bltcon0`, except that source b is shifted.

Bits 11-5: Commodore has not assigned a function to these bits yet.

Bits 4, 3 EFE/IFE:
Earlier we mentioned that the Blitter is capable of performing our fills. These two bits determine the fill mode for an area already set for the Blitter. When bit EFE is set (Exclusive Fill Enable), the area outline to the left of the filled area is erased after the fill operation. If EFE is unset and IFE is set (Inclusive Fill Enable), the area is filled normally. To achieve a clean fill, the area border must fulfill specific requirements: Only one single pixel may be set on each horizontal line (see SING).

Bit 2 FCI: This bit is the starting value for the "Fill Flip-Flop". Just like an electronic flip-flop, the Blitter also switches status back and forth while filling. Whenever a specific external event occurs, the Blitter switches and then keeps this status until a new event of the same type happens.

An event for the Blitter looks like this: The Blitter finds a set bit in a bit-plane row and the status of the Fill Flip-Flop is toggled. All the following bits that are clear are written with the value of FCI. You can also determine whether an area is filled inside or outside.

Bit 1 DESC:
This bit determines the direction in which the Blitter works in the specified data area (source a, b, c, d). When `DESC` is set, the direction is

downward from the provided address. When DESC is clear, the Blitter works upwards. This bit is very important when target and source overlap.

Bit 0 LINE:

(=0) When this bit is clear the Blitter operates in Area mode.

When drawing lines all of these bits have new meanings:

bltcon0

Bits 15-12 START3-0:

These bits set the code for the horizontal position of the first pixel of the line.

The exact format used by Commodore for START3-0 was not available when this book was written. For this reason, we have provided only limited information on drawing lines with the Blitter.

Bits 11, 10, 9, 8 %1011:

These bits must be initialized to a value of %1011, (only Commodore knows why).

Bits 7-0 LF7-0:

When drawing patterned lines, the Blitter sources are used in some capacity. Again, only Commodore has the complete information.

bltcon1

Bits 15-12 TEXTURE3-0:

A value (0-15) in these four bits sets the starting bit position for the pattern in a line.

Bits 11-7 %00000:

All these bits must be cleared.

Bit 6 SIGN:

This bit must contain the sign for the rise of the line.

Bit 5 %0: This bit is reserved for a new mode and should be kept clear for upward compatibility with other programs and machines like the 2000.

Bits 4-2 SUD, SUL, AUL:

These bits are used for super fast line drawing. This super fast method is based on the symmetric characteristics of a line that the Blitter uses. You control the drawing of a line with these three bits named "Sometimes Up or Down", "Sometimes Up or Left" and "Always Up or Left". Whether they can be set by you or only by the Blitter is something we were unable to find out.

Bit 1 SING:

If this bit is set, each line is displayed by only one pixel in each bit-plane row. This is important for the Blitter when filling areas.

Bit 0 LINE:

(=1) This bit determines the Blitter mode. When set, the Blitter operates in line mode.

---

bltsize $058 Blitter start and size (width, height)

This register contains the height and width of the Blitter's operating area. All operations occur in this defined window. The starting address is set in `bltXpt` and the operations are determined by `bltcon0/1` and other registers.

When you access these registers (through the Copper or 68000), whether reading or writing, the Blitter immediately executes the blit.

This is why you should first initialize all the other registers (`BLTAF/LWM`, `BLTxDAT`, `BLTCON0/1`, `BLTxMOD`, etc.) and then perform the write. This start effect also applies to drawing lines.

Bits 15-6 H9-H0:

These ten bits determine the number of rows for the Blitter's operating area and can contain a value between zero and 1024.

Bits 5-0 W5-W0:

These bits determine the width, in words (64 * 16 bits = 1024), of the Blitter's operation area. Theoretically the Blitter can operate with a memory area of 1024 x 1024 pixels (a superbitmap).

This register also has a different effect when it used for line drawing. `BLTSIZE` controls the line length and accessing this register tells the Blitter to start drawing the line. The H9-H0 bits control the line length (up to 1024 pixels) and bits W5-W0 must be initialized to %00010.

Since we could not provide complete information about the Blitter registers, we recommend that you utilize the libraries and BASIC instructions.

---

bplpt[6] $0E0 Bit-plane X pointer

This register pair (see `ac_ptr`) contains the 18 bit pointer to the starting address for bit-plane x (x= 1, 2, 3, 4, 5, 6) DMA data. This register, which must be initialized by the 68000 or the Copper after every raster return, is responsible for making your graphics visible.

Since there are a few more than the 480 visible rows on your screen (with interlace and overscan), a scrolling effect can be achieved through this register. However, instead of manipulating the hardware yourself, you can use the `RxOffset` and `RyOffset` variables in the `RasInfo` structure and the library instruction `ScrollVPort()` to scroll an entire bit-map in a ViewPort.

---

bpldat[6] $110 Bit-plane X data

As with DMA operations, registers are also required for temporary data storage. After the DMA controller stores data in these six registers, they are used by the video hardware (Videoshifter).

---

bpl1mod $108 Bit-plane modulo (odd planes)
bpl2mod $10A Bit-plane modulo (even planes)

These registers contain the modulo (the width) for the even (2, 4, 6) and odd (1, 3, 5) bit-planes (see `DBLPF` and `bplXmod`). The modulos are essential for a correct display.

---

bplcon0 $100 Bit-plane control register
bplcon1 $102 Bit-plane control register
bplcon2 $104 Bit-plane control register

You can control the video shifter like the Blitter. Use these three registers, which are the core of the entire graphic display.

bplcon0

Bit 15 HIRES:
If this bit is set, the display is in high resolution mode (640 * ... pixels). In this mode (4 bit-planes), you can only use 16 colors, which are determined by the lower 15 color registers (see `COLORxx`).

Bits 14-12 BPU2-0:
These bits determine how many bit-planes you are using (0-6). Six bit-planes are only required for dual playfields (DBLPF), hold-and-modify (HAM), and extra halfbrite modes. The number of bit-planes determines the number of possible colors: colors = 2BPU. So, it is possible to use five bit-planes and 32 colors without hold-and-modify. Setting all of these bits to zero displays only the background color (COLOR00).

Bit 11 HAM:
This bit switches on the hold-and-modify mode.

Bit 10 DBLPF:
This bit switches on the dual playfield mode.

Bit 9 COLOR:

Setting this bit sets your display output for a composite monitor. This means that the normal three color (RGB) is no longer used, and a single output for composite is activated.

All older model monitors are of the composite type. Since the colors must be separated from the single input signal, the picture on a composite isn't as good as one that uses RGB.

Bit 8 GENLOCK:

With a genlock interface it is possible to display, instead of the background color, the output from a videorecorder, videocamera or laser disk. This enables you to overlay the background video with your own titles and graphics. In order to use such an interface, your software has to set this bit.

Bit 7 EXTRA_HALFBRITE:

This bit switches on the halfbrite mode.

Bits 6-4: Not used

Bit 3 LPEN:

In order to use a light pen, this bit must be set. This tells the Amiga to test for the light pen position through the control port. Light pens can only be used with control port one (see VPOSR).

Bit 2 LACE:

This bit switches on the interlaced mode, which doubles the vertical display resolution from 200 rows to 400 rows. With interlace the screen is actually displayed twice. The first display is the even numbered rows and the second display is the odd numbered rows of the bit-map.

Bit 1 ERSY:

This bit enables external synchronization of the electronic scanning beam.

Bit 0: Not used

bplcon1

Bits 15-8: Not used

Bits 7-4 PF2H3-0,
Bits 3-0 PF1H3-0:

These bits set the number of pixels that a playfield is horizontally scrolled when displayed. You can use any value between zero and 15 pixels.

By using `bplpt[x]`, you can achieve word type scrolling. These bits are also available for finer scrolling.

However, you can also use `ScrollVPort()` and `RasInfo.RxOffset`. So this register is not too important.

bplcon2

Bits 15-7: Not used

Bit 6 PFsPRI:
Setting this bit gives playfield 2 complete video priority over playfield 1. Clearing this bit puts playfield 1 in the foreground (PFBA).

Bit 503 PF2P2-0,
Bits 2-0 PF1P2-0:
The bit combination in these bits set the video priorities between playfield 1 and the sprites and between playfield 2 and the sprites:

Value	Priority
000	PF1/2 > SP0/1 > SP2/3 > SP4/5 > SP6/7
001	SP0/1 > PF1/2 > SP2/3 > SP4/5 > SP6/7
010	SP0/1 > SP2/3 > PF1/2 > SP4/5 > SP6/7
011	SP0/1 > SP2/3 > SP4/5 > PF1/2 > SP6/7
100	SP0/1 > SP2/3 > SP4/5 > SP6/7 > PF1/2

('>' means: has Priority over)

You don't have to set the video priority for sprites because the hardware automatically knows that the lower numbered sprites have priority over the higher numbered sprites. The mouse pointer has sprite number zero with priority over all the other sprites.

When different ViewPorts are displayed, these registers are usually changed many times for each display. If there is only one ViewPort, the Copper changes these every sixtieth of a second.

---

clxcon $098 Collision control

This register determines what type of collisions (between bit-planes, sprites and bit-planes, or sprites and sprites) are looked for by your Amiga. Through the value stored in `clxdat`, you can discover which objects have collided. The individual bits for `clxcon` have the following functions:

Bits 15-12 ENSP7, 5, 3, 1:
Setting one of these bits allows you to check the odd numbered sprites (7, 5, 3, 1) for collision control. The even numbered sprites (0, 2, 4, 6) are always checked for collisions, but the odd ones aren't.

However, unless you are using attached sprites that are at the same position, you cannot determine through CLXDAT whether an odd or even numbered sprite has collided with an object.

Bits 11-6 ENBPx:
Use these bits to set which bit-planes (x = 6-1) are tested for collisions. Clearing these bits always indicates one bit-plane collision.

Bits 5-0 MVBP6-1:
These bits determine whether bit-plane collisions with sprites are registered by set or cleared bits. For example, only one bit-plane collision is registered when a pixel is set in bit-plane 1 and not registered for bit-plane 2 at the same position, etc. This makes it possible to determine if a sprite has collided with a specific colored pixel.

---

clxdat $00E Collision data register

This register helps you to determine which collisions have occurred.

Bit 15	unused (usually = 1)
Bit 14	Sprite 4 (or 5) with sprite 6 (or 7)
Bit 13	Sprite 2 (or 3) with sprite 6 (or 7)
Bit 12	Sprite 2 (or 3) with sprite 4 (or 5)
Bit 11	Sprite 0 (or 1) with sprite 6 (or 7)
Bit 10	Sprite 0 (or 5) with sprite 4 (or 5)
Bit 9	Sprite 0 (or 1) with sprite 2 (or 3)
Bit 8	Playfield 2 with sprite 6 (or 7)
Bit 7	Playfield 2 with sprite 4 (or 5)
Bit 6	Playfield 2 with sprite 2 (or 3)
Bit 5	Playfield 2 with sprite 0 (or 1)
Bit 4	Playfield 1 with sprite 6 (or 7)
Bit 3	Playfield 1 with sprite 4 (or 5)
Bit 2	Playfield 1 with sprite 2 (or 3)
Bit 1	Playfield 1 with sprite 0 (or 1)
Bit 0	Playfield 1 with playfield 2

(Playfield 1 is the odd bit-planes (1, 3, 5) and playfield 2 the even (2, 4, 6).

This register provides many possibilities for determining sprite collisions. Since bit-plane collisions provide less information, specific determinations are impossible.

Remember that sprite collisions are first registered when two pixels overlap instead of when the edges collide.

Also, when you read this register, it is immediately cleared and not written to again until the next raster scan (vertical blank).

---

color[32] $180 Color register

There are 32 color registers ($180 - $1BE) that contain a 12 bit color code for every color that exists in the bit-planes.

The color register that determines the color information for a specific pixel is calculated by ORing - the value for all bit-planes for that pixel position.

The color code is composed of 12 bits which makes 4096 different colors possible. You can use a maximum of 32 of the 4096 colors for one screen (hold-and-modify is an exception to this rule). Your pixel is then displayed in the color calculated for its color register. Here is the bit pattern used in all of the color registers:

Bits 15-12		Unused
Bits 14-8	Red3-0	Red component of color
Bits 7-4	Green3-0	Green component of color
Bits 3-0	Blue3-0	Blue component of color

The fourth bit determines the intensity of a color component (red, green, blue). It is possible to create almost any color by mixing the three different colors and their intensities (0 = dark, $0f = light).

---

copcon $02E Copper control register

This register is a one bit register. Only bit one is used, all the other bits are unused.

Some computer users refer to this register as the Copper danger bit. Register zero (`bltddat`) to register 30 (`strlong`) are exempt from Copper manipulation.

Usually, the Copper cannot access registers 31 (`bltcon0`) to 49 (`dsksync`). Setting the danger bit allows you to use these registers. Any reset clears this bit.

---

copjmp1 $088 (Strobe) Copper Newstart the first list
copjmp2 $08A (Strobe) Copper Newstart the second list

By accessing these registers, either through the Copper or 68000, causes the Copper to execute a Copper list. The starting list address should be in `cop1lc` or `cop2lc` and the Copper-Program-Counter (PC) is directed to this address.

The Copper list, whose start address is in `copllc`, is executed with each raster scan (vertical blank).

For your own experiments with Copper lists it is safer to use the user Copper list. The entire display is controlled by the Copper and direct access can cause serious problems unless you take control of the entire display.

---

cop1lc	$080/$082	Address of first Copper list
cop2lc	$082/$084	Address of second Copper list

When you access registers `copjmp1` or `copjmp2`, they contain the starting addresses of Copper lists for execution.

The address of the hardware Copper list is loaded into these registers through `LoadView().Cop2lc` is only used with interlace mode for the second screen scan. After the first scan the second Copper list is started by the first.

---

copins	$08C	Copper instruction register

Like the DMA channel, the Copper also uses a temporary buffer. Two 16 bit instruction words are stored in this register and executed sequentially.

The first instruction word consists of one of the three possible Copper instructions, the affected hardware register and the position of the electronic beam that will be waited for.

However, the three possible instructions, `CMOVE()`, `CWAIT()` and `CEND()` aren't the only instructions you can use. The instructions `Move`, `Wait` and `Skip` are also available.

The reason for this is that the `CEND()` instruction is actually a sub-instruction of `CWAIT()` and the `Skip` instruction wasn't considered useful enough to be included in C.

`Move` and `Wait` have the same meaning as `CMOVE()` and `CWAIT()`.

The `Skip` instruction actually skips the following Copper instructions when the electronic beam has reached the selected position.

A complete Copper instruction consists of two 16 bit words that are loaded and executed sequentially in these registers.

Instruction: **MOVE**

1. Instruction Word

Bits 15-9 Unused

Bits 8-1 DA8-1:
DAx (Destination Address), which is used by the Move instruction, selects the register the value of the second instruction will be written to. DA is specified as an offset from $DFF000 for the selected hardware register.

Bit 0 0: This bit is used to identify a Copper move instruction. `Move` must always be zero (The first bit of the second instruction word isn't required for decoding this instruction).

2. Instruction Word

This word contains the 16 bit value that will be written to the register selected by DA.

WAIT

1. Instruction Word

Bits 15-9 VP7-VP0:
VP (Vertical Position) contains the vertical (Y) position of the electronic beam that will be waited for.

There are 262 different vertical beam positions that can be waited for. Since VP has a data width of only eight bits, a small trick must be used in order to access positions 256 through 262.

Usually, first you wait for the vertical position 255. Then you use a second wait instruction for a position between zero and six.

Bits 8-1 HP8-HP1:
HP (horizontal position) contains the horizontal (X) position for the electronic beam that will be waited for. It can contain a position value between zero and 226. Since the available data width is only seven bits, the Copper can access only 113 positions. This limits you to a pixel resolution of four with low-res (HIRES = 0) or eight with hi-res (HIRES = 1).

It is possible for you to wait for every fourth pixel position. However, this doesn't mean that you can change a register for every four pixels. When you execute two moves, one after the other, there is a gap of eight low-res pixels (with hi-res this value doubles).

Bit 0 1: This bit and bit zero of the second instruction word are used to identify the Copper instructions `Wait` and `Skip`. When bit zero of the first

word is clear, the Copper knows that it is working with a `Move` instruction so it doesn't test the second instruction word.

When bit zero of the first word is set, the Copper knows that it must either `Wait` or `Skip`. Which action is performed is set by you in the instruction word.

2. Instruction Word

Bit 15 BFD:
Setting this bit puts the `Wait` instruction on hold until the Blitter says "OK" (Blitter Finished Disable).

Bits 14-8 VE6-VE0:
VE (Vertical Comparison Enable) and HP (Horizontal Comparison Enable) determine which bits are used for comparing the electronic beam position.

Bits 7-1 HE-8-HE-2:
This bit has the same meaning for horizontal position as VE does for vertical positioning.

Bit 0 %0: The Copper recognizes the `Wait` instruction at this location.

SKIP

Except for the set or unset status of bit zero of the second instruction word, the bit pattern of the instruction words for `Skip` and `Wait` are very similar. The meaning of all the other bits is the same.

---

diwstrt	$08E	Display window start (top, left position)
diwstop	$090	Display window stop (lower, right position)

Use these registers to set the actual display (ViewPort) size for your screen. The values in these registers determine the upper left corner and the lower right corner of the screen. Besides the background color (in color[0]), nothing can be displayed outside this screen.

These positions are specified in normal (not interlaced) raster row and low resolution pixel values. You don't have to change these values to use interlace or hi-res modes.

The bit pattern of these registers is as follows:

diwstrt

Bit	Function	
15-8	VSTART	vertical start position
7-0	HSTART	horizontal start position

The starting position of your display is set in these two registers. However, you cannot set the starting position of a ViewPort here.

Because of the eight bit data width of these registers, you can only use horizontal and vertical starting and ending positions between zero and 256 raster rows.

diwstop

Bit	Function	
15-8	VSTOP	vertical start position
7-0	HSTOP	horizontal start position

`Diwstop` also has the same eight bit pattern. In order to display a window wider than 256 pixels, a value of $100 must be added to the X end coordinate to maintain the actual end position.

Bit VSTOP7 is very important for calculating the actual vertical end coordinate. The nonexistent VSTOP8 bit is actually the compliment of VSTOP7. If VSTOP7 is cleared, VSTOP8 is set. When VSTOP7 is zero, you simply add 256 to calculate the actual vertical end position.

If VSTOP7 is set you can disregard VSTOP8 since it is zero and not active in the calculation.

When writing to these registers, be aware of "Vertical Blank". The top visible raster is not raster zero just as the first visible column is not the first column raster wise.

The electronic beam needs some time to recover after reaching the bottom screen border. This is similar to when the beam finishes displaying a raster row.

Normal values for `diwstrt` and `diwstop` are:

diwstrt = $2c81
diwstop = $f4c1

This provides a screen window size of:

HSTOP+$100-HSTART = $c1+$100-$81 = $140 = 320 Pixels
VSTOP+(VSTOP8 * $100)-VSTART = $F4(= %11110100)+(0 * $100) - $2c = 200 Normal-Raster rows

Also remember not to use a vertical starting value smaller than $20 because, while creating the screen, the vertical blank needs an area for the beam return. Otherwise you wouldn't see anything on the screen because the electronic beam wasn't given enough recovery time.

ddfstrt	$092	Screen data fetch start
ddfstop	$094	Screen data fetch stop

These two registers determine when to get the data from memory and display it.

One timing cycle is used to display two pixels. To determine when to start and stop displaying data, use the values from the `diwstrt` and `diwstop` registers. Eventually you will add a small offset to these values and then the desired values for `ddfstrt` and `ddfstop` will be available.

The offset is different for lo-res and hi-res modes. For lo-res the offset is 8.5, and for hi-res it is 4.5.

Start position values are:

ddfstrt = ($81/2 - 8.5) = $38
(hi-res: ddfstrt = ($81/2 - 4.5) = $3C)

You calculate the value for `ddfstop` like this:

ddfstop = ddfstrt + (16/2 *(Number of words per line -1))
(low-resolution)
ddfstop = ddfstrt + (16/4 *(Number of words per line -2))
(high-resolution)

Again the number of displayed pixels (1 word = 16 pixels) is halved and quartered. The quartered value for hi-res is natural because hi-res displays twice as many pixels in the same amount of time.

You cannot select these values because they are limited from the hardware side. `DDFSTRT` cannot be smaller than $18 and `DDFSTOP` cannot be larger than $D8. These values permit a display of 376 pixels with lo-res (752 with hi-res). By using a `DDFSTRT` value smaller than $38, you can prevent the display of several sprites.

dmacon	$096	DMA control register (write)
dmaconr	$002	DMA control register (read)

These registers are used by the DMA controller to manage the information fed to it by the various processors.

With `dmacon`, set which DMA channel is open.

This is the bit pattern which is the same for both registers:

Bit 15 SET/CLR:
(see ADKCON)

Bit 14 BBUSY:

This bit shows the Blitter status. When the Blitter is working this bit is set.

Bit 13 BZERO:

This bit is set when all result bits of a blit (screen area copy, etc.) were zero.

Bits 12, 11: Unused

Bit 10 BLTPRI:

This bit determines the priority between the Blitter and the 68000. How the 14 MHZ clock frequency is divided between the two processors is set here.

When set, the Blitter takes every possible clock cycle, even those belonging to the 68000. When clear, the 68000 receives one clock cycle for every third memory refresh cycle.

Bit 9 DMAEN:

When this bit is set, it is possible to activate all the DMA channels. If this bit is clear, the values of all the following bits have no effect on the status of the DMA channels that they activate. So, DMAEN is the controller of DMA activity.

Bit 8 BPLEN:

When this bit and DMAEN are set, the DMA channel for screen data transfer from the bit-planes is activated.

Bit 7 COPEN:

This bit sets the Copper status. If it is set, the Copper can work and if it is clear, the Copper is shut off.

Bit 6 BLTEN:

What applies to the Copper with COPEN applies here to the Blitter.

Bit 5 SPREN:

Setting this bit enables transfer of sprite data through the DMA channel.

Bit 4 DSKEN:

This bit determines the status of the disk DMA channel. Clearing this bit stops all data transfer between computer and floppy.

Bits 3-0 AUD3-0EN:

These bits determine the status for the sound DMA channels. Clearing these bits prevents any sound activation through the DMA channels.

dskpt $020 Disk pointer

This register pair (also an 18 bit pointer) determines the data read storage address for the disk DMA data that will will be written to disk.

dsklen $024 Disk data length

This register contains the word count of the words to be sent or received through the disk DMA channel. One bit of this register determines the data direction (RAM -> disk or disk -> RAM). Another bit determines whether access to the DMA channel is possible.

Accessing this register executes the data transfer. The register must be written to twice with the same data. When all data has been transferred a "Disk Block Interrupt" (see INTREQ) is sent that stops the data transfer.

Just setting the DMA register isn't the only requirement for executing a complete disk operation such as loading a program. You must also set a pair of I/O registers for the CIAs. However, we won't discuss these registers because they belong in a different book, such as "**Amiga System Programmers Guide**".

Bit 15 DMAEN:
Setting this bit enables the data transfer through the DMA channel.

Bit 14 WRITE:
This bit determines the write/read direction. When set, the data is read from RAM through the DAM channel and written to disk. When clear, the data is read from disk.

Bits 13-0 LENGTH:
These bits contain the number of words that will be read or written.

dskdat $026 Disk DMA data (write)
dskdatr $008 Disk DMA data (read)

These registers contain the data that is either read or written through the disk DMA channel.

You should remember that the `dskdatr` register is protected from Copper access.

dskbytr $01A Disk databyte and status

This register is a data buffer that is directly linked to the disk microprocessor. From this register, data is sent to the DMA controller

or received from the disk controller. There are also a few status bits in this register.

Bit 15 DSKBYT:
When bytes are read from disk, this bit is set as soon as the byte has been completely read.

Bit 14 DMAON:
This bit, which has the same value as DMAEN in dislen, is ANDed with bit DMAEN from dmacon. When the result is a one, data transfer from the disk device is possible.

Bit 13 DISKWRITE:
This bit has the same value and meaning as WRITE in dsklen.

Bit 12 WOREQUAL:
This bit is set only when the dsksync register byte data will be synchronized with the word data.

Bits 11-8: Unused

Bits 7-0: These bits contain the data byte.

---

dsksync	$07E	Disk synchronization register

This register contains the synchronize code for disk operations.

---

intreq	$09C	Interrupt request bits (write)
intreqr	$01E	Interrupt request bits (read)
intena	$09A	Interrupt enable bits (write)
intenar	$01C	Interrupt enable bits (read)

These are the interrupt requests, interrupt enable or mask bits that determine which interrupts (cyclic breaks) are allowed.

This is where you can decide between request and enabling. Interrupts can only occur when the corresponding enable bits are set.

All four registers have the same bit pattern. The request and enable registers are included twice, once for writing and again for reading.

Here is the bit pattern:

Bit 15 SET/CLR:
(see ADKCON)

Bit 14 INTEN (Master Interrupt enable):
When this bit is clear all interrupts for the following bits are disabled.

Bit 13 EXTER:

This bit is used to generate an external interrupt through CIAB (level 6).

Bit 12 DSKSYN:

This interrupt is executed when data is synchronized with the DSKSYNC register (level 5).

Bit 11 RBF (receive buffer full):

This is where the interrupt for the serial port is tested. It is executed when the serial input buffer is full and can be read by the user (level 5).

Bits 10-7 AUD3-0:

An interrupt is executed when sound data will be processed through the DMA channel. In manual mode this interrupt is executed when the audio data registers are ready to operate on new data (level 4).

Bit 6 BLIT (blitter finished):

This bit is set when the Blitter has completed its work (level 3).

Bit 5 VERTB:

An interrupt routine, which resets many pointers and executes other system tasks, is performed for every raster return.

Bit 4 COPPER:

COPPER indicates that a Copper interrupt will be executed. Just as it can change almost any register, the Copper can change any of these bits. After the entire screen has been displayed and the electronic beam has reached the DIWSTOP position, the Copper interrupt occurs. This enables the 68000 to process special tasks during the raster return.

Bit 3 PORTS:

PORTS indicates an interrupt that is executed during the low processor cycle INT2 (level 2).

Bit 2 SOFT:

This bit is reserved for software interrupts.

Bit 1 DSKBLK (Disk block finished):

DSKBLK indicates that a data transfer through the disk DMA channel is complete (level 1).

Bit 0 TBE (transmit buffer empty):

TBE indicates that the UART (Universal Asynchronous Receiver/Transmitter) output buffer can receive data and you can write into this buffer (level 1).

Usually by setting the processor lines INT0, INT1 and INT2 low, there could be seven different interrupts. These would only be system interrupts.

However, the Amiga further specifies these system interrupts within the interrupt handler, which produces more than seven possible interrupt levels.

---

joy0dat	$00A	Joystick/mouse 1 (left port)
joa1dat	$00C	Joystick/mouse 2 (right port)

These two registers provide information for the status of the entry device (joystick, mouse, lightpen, potentiometer, etc.) that is connected to the related control port. The following applies to the specific entry devices:

Mouse or trackball:

For these devices the register contains a value that is proportional to the actual movement. This movement is split between the horizontal (X direction) and vertical (Y direction) in the register. Bits 15-8 contain the vertical movement and bits 7-0 contain the horizontal movement. The register values come from the rotation of the ball, which turns the raster sliders between light sensors.

Breaking the light sensor contact and then making contact again represents one impulse. The Amiga mouse sends 200 impulses per inch. These two bytes are then counters of the light impulse breaks.

The counters increase when you move the mouse left or down and decrease when you move the mouse right or up. To determine the movement direction, compare the difference of the current value to the new value. When the difference is negative, the mouse is moving down or right.

In order to make the mouse position available to Intuition, this register is read and cleared after each vertical blank.

The left mouse button is read through bit six of the CIAA register $BFE001. The right mouse button is read from the `pot0dat` register.

The Joystick:

Testing for joystick positions is easier than testing mouse movements. When bits one or nine for `joy0/1dat` are set, the joystick has been moved left or right. To check for up and down movement, you must XOR bits nine and eight with bits one and zero. The fire button for a joystick in the left port is tested the same way as the left mouse button. The fire button for the right joystick is read from bit five of register $BFE001.

The typical programmer shouldn't try direct reading of the joystick and mouse because the system software handles this for you. However, when the system does not satisfy your control requirements, you could experiment with the information presented here.

---

joytest $036 Write all mouse counters

The values written to this register are transferred to both `joy0dat` and `joy1dat` registers. This allows you to initialize both of them at the same time.

---

pot0dat $012 Pot counter left (vert, horiz).
pot1dat $014 Pot counter right (vert, horiz).

Like most computers, paddles can also be used with the Amiga. These are simply potentiometers that are read every sixtieth of a second for their current position.

When you set the starbit in `potgo`, electricity is sent through the resistor via a capacitor. The capacitor loads for a short time (about eight raster rows) and then unloads. Then the actual load of the capacitor is compared to a standard value.

If the current load is higher than the standard, a previously cleared counter is incremented. When the load is smaller, the compare operation is canceled and the proportional resistance value is stored in this register. You can connect two of these paddles to each control port, which is allowed for proportional joysticks. Test the paddle fire buttons as usual through the left and right joystick positions. Here is the bit pattern for these registers:

Bits 15-8 Y7-Y0:
Potentiometer position of pot at pin 9

Bits 7-0 X7-X0:
Potentiometer position of pot at pin 5

The electrical resistance, which must be at least 470K Ohms, plus or minus ten percent, is sent through pin seven (+5 V, 125mA).

---

potinp $034 Pot port data write and start
potgo $016 Pot port data read

The control ports also function as 4 bit bidirectional I/O ports. These registers are used to control them:

Bit 15 OUTRY (right Port, Pin 9)
Bit 14 DATRY

Bit 13 OUTRX (right Port, Pin 5)
Bit 12 DATRX
Bit 11 OUTLR (left Port, Pin 9)
Bit 10 DATLR
Bit 9 OUTLX (left Port, Pin 5)
Bit 8 DATLX

The OUTxx bits set the line status. When these bits are set, data DATxx is output. If these bits are cleared, data DATxx is read.

Bits 7-0 0: Reserved

Bit 0 START:

When this bit is set (`potgo`), the capacitors are loaded and the counters start counting.

---

refptr $028 Write refresh pointer

This pointer is used as a RAM refresh pointer. Instead of constantly receiving electricity, the memory bits are loaded every 280 nanoseconds (= one memory cycle). Therefore, their old value is constantly kept fresh. Because of this technique, it is possible to use five volt power for most ICs.

These registers should never be accessed by the 68000 or Copper because this can confuse the internal timing, which can cause loss of data.

---

serdat $030 Serial port: data and stop bits (write)

The data that is written to this register is further written to a shifter, which transfers a byte, bit by bit, through the serial bus. When this buffer can receive data, it sends a TBE (Transmit Buffer Empty) interrupt (see INTREQ). Depending on the setting of the LONG bit in the serper register, either eight or nine data bits are used.

Bits 15-10: 0

Bit 9: Stopbit

Bits 8-0: Databits

---

serdatr $018 Serial port: data and status (read)

This register, which is the opposite of `serdat`, contains the data bits received from the buffer through the shifter. Various interrupt request bits (INTREQ) are also in this register.

Bit 15 OVERRUN:
This bit is set when there is an input buffer overflow. Clearing RBF in INTREQ clears this bit.

Bit 14 RBF:

Bit 13 TBE:
see INTREQ

Bit 12 TSRE:
This bit is set when the shifter register is empty.

Bit 11 RXD:
The RXD processor pin receives data directly from the UART. This bit can be tested directly from the 68000.

Bit 10: 0

Bit 9: Stopbit

Bit 8: Stopbit or D8

Bits 7-0 D7-D0:
(Databits)

---

serper $032 Serial port: rate and control

With this register you set the word data width (8 or 9 bits), which the serial port will send or receive. The data transfer speed is also set here.

Bit 15 LONG:
Setting this bit makes your data word nine bits wide.

Bits 14-0 RATE:
These bits specify the speed for sending or receiving one bit. To calculate the time, use the following formula: (value of the lower 14 bits + 1) * 2794 microseconds.

---

sprpt[8] $120 Sprite x Pointer

This register pair contains the starting data address for every sprite (x = 0, 1, 2, 3, 4, 5, 6, 7). In order to use sprites, this register must be reset by the Copper or 68000 after every raster return. To display sprites, the DMA channel (dmacon/SPREN) must also be active.

spr[8].pos	$140	Sprite X vertical & horizontal start position
spr[8].ctl	$142	Sprite X vertical stop position

These registers work together to control the size, position and other functions connected with hardware sprites. They are usually loaded by the DMA controller during the horizontal raster beam return.

The positions are again specified in raster rows or normal pixels.

pos

Bits 15-8 SV7-SV0,
Bits 7-0 SH8-S8H1:
SV7=SV0 sets the vertical (Y direction) start position of sprite x. SH7=SH0 contains the horizontal (X direction) start position of sprite x. There are only eight bits available for each position coordinate. So `spr[x].ctl` contains the high value bit SV8 and the SH0 contains the low value bit. This allows position values between zero and 512 in both horizontal and vertical directions. The movement of sprites is only possible in steps of low-res pixels and in normal raster rows.

ctl

Bits 15-8 EV7-EV0:
These bits contain the vertical (Y) end position of sprite x. This clearly reveals that sprites can have any desired height (sprites are always 16 pixels wide). Remember that the highest bit (EV8) is bit one.

Bit 7 ATT: Setting this bit selects paired sprites (0/1, 2/3, 4/5, 6/7) that provide 15 colors per sprite instead of only three. Each sprite in a pair can be moved independently. The 15 color bit pattern is visible only when the sprites overlap.

Bits 6-4: Unused

Bit 2 SV8: High value bit for SV7-SV0 in `spr[x].pos`

Bit 1 EV8: see EV7-EV0

Bit 0 SH0: High value bit for SH7-SH0 in `spr[x].pos`

spr[8].dataa	$144	Sprite X Image data register A
spr[8].datab	$146	Sprite X Image data register B

These registers contain the rows of a sprite that will be displayed. `dataa` contains the first word and `datab` contains the second word of any sprite. When `dataa` is written into the register, it is compared

with the actual raster position in SH8-SH0. When both values are the same, the sprite row is displayed.

Changing `spr[x].ctl` affects this comparison. The beam position is compared only after `dataa` is accessed and eventually the sprite is displayed.

strequ	$038	Strobe for horiz. synchronization with VB and EQU
strvbl	$03A	Strobe for horiz. synchronization with VB
strhor	$03C	Strobe for horiz. synchronization
strlong	$03E	Strobe for identification of a long horizontal.

These registers are used internally by the video shifter. Unfortunately, we are unable to provide more detailed information about them. However, the typical programmer really doesn't need these registers.

vposr	$004	Highest value vertical positions bit (read)
vposw	$004	Highest value vertical positions bit (write)

To synchronize actions of the 68000 with the electronic scanning beam, these registers are used. So, the 68000 can wait for a specific Y coordinate of the electronic beam and then take whatever action is required by the program (see WaitTOF).

These registers only contain the highest value bit for the vertical position and an interlaced flag.

Bit 15 LOF:
This bit indicates whether or not interlace is switched on.

Bits 14-0: Unused

Bit 0 V8: This is the highest value bit for the vertical electronic beam position. You can also test for rows higher than 256 (313 for PAL and 262 for NTSC).

vhposr	$006	Vertical and horizontal electronic beamposition (read)
vhposw	$02C	Vertical and horizontal electronic beamposition (write)

These registers are used to read the current beam position or to set (write) a new position, which is sometimes necessary when performing tests. Usually these registers aren't used except for diagnostic purposes.

The bit pattern is as follows:

Bits 15-8 V7-V0:

These bits contain the lowest bit values for the vertical beam position. The high bit is stored in VPOSR.

Bits 7-0 H8-H1:

These bits contain the actual horizontal position of the beam. The resolution value consists of 1/160 of the screen width.

The following register list is in offset address order (please remember the offset from $DFF000):

Nr.	Name	Address
000	bltddat	$000
001	dmaconr	$002
002	vposr	$004
003	vhposr	$006
004	dskdatr	$008
005	joy0dat	$00a
006	joy1dat	$00c
007	clxdat	$00e
008	adkconr	$010
009	pot0dat	$012
010	pot1dat	$014
011	potinp	$016
012	serdatr	$018
013	dskbytr	$01a
014	intenar	$01c
015	intreqr	$01e
016	dskpt	$020
017	dsklen	$024
018	dskdat	$026
019	refptr	$028
020	vposw	$02a
021	vhposw	$02c
022	copcon	$02e
023	serdat	$030
024	serper	$032
025	potgo	$034
026	joytest	$036
027	strequ	$038
028	strvbl	$03a
029	strhor	$03c
030	strlong	$03e
031	bltcon0	$040
032	bltcon1	$042
033	bltafwm	$044
034	bltalwm	$046
035	bltcpt	$048
036	bltbpt	$04c
037	bltapt	$050
038	bltdpt	$054
039	bltsize	$058
	-------	$05a
	-------	$05c

Nr.	Name	Address
	------	$05e
040	bltcmod	$060
041	bltbmod	$062
042	bltamod	$064
044	bltdmod	$066
	------	$068
	------	$06a
	------	$06c
	------	$06e
045	bltcdat	$070
046	bltbdat	$072
048	bltadat	$074
	------	$076
	------	$078
	------	$07a
	------	$07c
049	dsksync	$07e
050	cop1lc	$080
051	cop2lc	$084
052	copjmp1	$088
053	copjmp2	$08a
054	copins	$08c
055	diwstrt	$08e
056	diwstop	$090
057	ddfstrt	$092
058	ddfstop	$094
059	dmacon	$096
060	clxcon	$098
061	intena	$09a
062	intreq	$09c
063	adkcon	$09e
aud[0]		
064	ac_ptr	$0a0
065	ac_len	$0a4
066	ac_per	$0a6
067	ac_vol	$0a8
068	ac_dat	$0aa
	-----	$0ac
	-----	$0ae

Nr.	Name	Address
aud[1]		
069	ac_ptr	$0b0
070	ac_len	$0b4
071	ac_per	$0b6
072	ac_vol	$0b8
073	ac_dat	$0ba
	------	$0bc
	------	$0be
aud[2]		
074	ac_ptr	$0c0
075	ac_len	$0c4
076	ac_per	$0c6
077	ac_vol	$0c8
078	ac_dat	$0ca
	------	$0cc
	------	$0ce
aud[3]		
079	ac_ptr	$0d0
080	ac_len	$0d4
081	ac_per	$0d6
082	ac_vol	$0d8
083	ac_dat	$0da
	------	$0dc
	------	$0de
084	bltpt[0]	$0e0
085	bltpt[1]	$0e4
086	bltpt[2]	$0e8
087	bltpt[3]	$0ec
088	bltpt[4]	$0f0
089	bltpt[5]	$0f4
	--------	$0f8
	--------	$0fa
	--------	$0fc
	--------	$0fe
090	bplcon0	$100
091	bplcon1	$102
092	bplcon2	$104
	-------	$106
093	bpl1mod	$108

Nr.	Name	Address
094	bpl2mod	$10a
	-----	$10c
	-----	$10e
095	bpl1dat	$110
096	bpl2dat	$112
097	bpl3dat	$114
098	bpl4dat	$116
099	bpl5dat	$118
100	bpl6dat	$11a
	-----	$11c
	-----	$11e
101	sprpt[0]	$120
102	sprpt[1]	$124
103	sprpt[2]	$128
104	sprpt[3]	$12e
105	sprpt[4]	$130
106	sprpt[5]	$134
107	sprpt[6]	$138
108	sprpt[7]	$13c
spr[0]		
109	pos	$140
110	ctl	$142
111	dataa	$144
112	datab	$146
spr[1]		
113	pos	$148
114	ctl	$14a
115	dataa	$14c
116	datab	$14e
spr[2]		
117	pos	$150
118	ctl	$152
119	dataa	$154
120	datab	$156

Nr.	Name	Address
spr[3]		
121	pos	$158
122	ctl	$15a
123	dataa	$15c
124	datab	$15e
spr[4]		
125	pos	$160
126	ctl	$162
127	dataa	$164
128	datab	$166
spr[5]		
129	pos	$168
130	ctl	$16a
131	dataa	$16c
132	datab	$16e
spr[6]		
133	pos	$170
134	ctl	$172
135	dataa	$174
136	datab	$176
spr[7]		
137	pos	$178
138	ctl	$17a
139	dataa	$17c
140	datab	$17e
141	color00	$180
142	color01	$182
143	color02	$184
144	color03	$186
145	color04	$188
146	color05	$18a
147	color06	$18c
148	color07	$18e
149	color08	$190
150	color09	$192

Nr.	Name	Address
151	color10	$194
152	color11	$196
153	color12	$198
154	color13	$19a
155	color14	$19c
156	color15	$19e
157	color16	$1a0
158	color17	$1a2
159	color18	$1a4
160	color19	$1a6
161	color20	$1a8
162	color21	$1aa
163	color22	$1ac
164	color23	$1ae
165	color24	$1b0
166	color25	$1b2
167	color26	$1b4
168	color27	$1b6
169	color28	$1b8
170	color29	$1ba
171	color30	$1bc
172	color31	$1be
...		
xxx	NO-OP	$1fe

Index

3-D effects 39
1024x1024 paint program 293

AbortDoubleBuffer 180
absolute address 9
absolute coordinates 9
access counter 216
action-verbs 51
activity menu 1 5
AddAnimOb 483, 527
AddBob (&Bob, &RastPort) 527
AddFont (&TextFont) 527
addressing 8
AddVSprite (&VSprite, &RastPort) 528
AddVSprite 455
AllocMem (NumBytes, Use) 553
AllocMem() 157, 203, 319
AllocRaster (Width, Height) 528
AllocRaster() 170, 203, 204, 319
Amiga libraries 316
AND 53
Angle parameters 24
Animate (&AnimOb,&RastPort) 528
Animated bit-planes 80
Animation 21, 55
animation 479, 481
AnimationComponents 479
AnimationObjects 479
AnimComp 481
AnimComp structure 480
AnimComp.AnimBob 481
AnimComp.AnimCRoutine 481
AnimComp.NextComp 480
AnimComp.PrevComp 480
AnimComp.TimeSet 481
AnimComp.XTrans 482
AnimComp.YTrans 483
AnimComps 479, 482
AnimOb 481, 482
AnimOb.AnX 482
AnimOb.AnY 482
AnimOb.HeadComp 481
AnimOb.PrevOb 483
AnimOb.XAccel 482
AnimOb.XVel 482
AnimOb.YAccel 482
AnimOb.YVel 482
AnimObs 479, 484
AnimORoutine 483
AREA 31
Area... functions 356
AreaDraw (&RastPort, x, y) 528
AreaEllipse 529
AreaEnd (&RastPort) 529
AREAFILL 31, 32, 33, 124
AreaInfo 124
AreaInfo structure 356
AreaMove (&RastPort, x, y) 530
AreaPtSz 126
ASCII 249
ASCII codes 216
AskFont (&RastPort, &TextAttr) 530
AskSoftStyle (&RastPort) 530
aspect ratio 20, 23
asynchronous 274
AvailFonts (&Buffer, BufferSize, Typ) 550
AvailFonts 228
AvailFonts structure 393

Backdrop (Layerbackdrop) 197
backdrop window 99
Backup-Register 117
bar graph 19
BASIC interpreter 93
binary numbers 35
binary pattern mask 36
bit pattern 355
Bit-map 117, 123, 166, 319, 402,417
bit-map structure 321

bit-planes 46, 80, 81, 149, 319, 355, 402, 415
BitMap 149, 344
Bitmap Offset 153
Blitter 52, 349, 399, 400, 465
Blitter objects 465
BltBitMap 278, 400, 530
BltBitMapRastPort 531
BltClear (&Memory, NumBytes, Flags) 531
BltClear 399
BltTemplate 407, 412, 532
bmap file 87
Bob.BobComp 480
Bob.SaveBuffer 477
BOBISCOMP 480
bobs 55, 85, 458, 465, 466, 467, 468, 476, 479, 484
BOBVSprite.Depth 466
bold print 215
border color 347
borderless window 99
BorderLine 456, 466
brush effect 9
byte field 95

C language 315
CEND 497
ChangeSprite 436, 532
Character spacing 128
Character width 127
CharData 235, 412
charLoc 235
checkmark 69, 105
CHR$(0) 100
CIRCLE statement 24, 30
circle 20
ClearEOL (&RastPort) 533
ClearPointer 109
ClearScreen (&RastPort) 533
ClipBlit 402, 533
ClipRect 192, 194
CLOSE 50
Close gadget 99
CloseFont (&TextFont) 533
CloseLibrary (&BasePointer) 554
CloseRequest 199
CloseScreen (&Screen) 551
CloseWindow (&Window) 551
CLS statement 41
CMOVE 495, 497
Collision mask 76
collision 440
collision control 456, 466
collision detection 496
collisions 456, 458, 484
CollMask 456, 466, 467
COLOR statement 33
color 382
color components 382
color cycling 44, 479
color depth 4
color map 318
color pens 339
color registers 4
ColorMap 153, 344
ColorMap structure 318
ColorPattern 134
colors 69
COMPLEMENT 125, 340
constants 503
ConvertFD 87
Copper 153, 180, 185, 452, 495
Copper list 1 53, 169, 174, 181, 317, 323, 381, 382, 455, 477, 495, 496, 542
Copper programming 435
CreateUpfrontLayer 190, 203
cursor 126
CWAIT 496, 497

damage list 195
DBufPacket 477
DECLARE FUNCTION LIBRARY 90
DEFINT 36
Delay (Time) 553
DeleteLayer() 190, 204
Depth 69
DIM instruction 36
Disk fonts 221
diskfont.bmap 87
diskfont.libraries 550
Diskfont.library 221
DiskFontBase functions 550
DisownBlitter () 534

DisplayBeep (&Screen) 551
DMA 451
DoCollision 457, 464, 534
dos.bmap 87
doScroll 205
Double Buffering 476, 477
double buffered displays 423
double buffering 175
DoubleBufferOff 180
DoubleBufferOn 179
Dragbar 99
Draw (&RastPort,x,y) 332, 534
Draw() 204
DrawEllipse 534
DrawGList 477, 535
Drawing color 124
Drawing Mode 125
drawing 69
drawing pens 341
Drawmode 340
Dual Playfield mode 422
DUALPF flag 422
DUALPF mode 155, 477
dualplayfield 476

Eddi II 56
ellipsoids 3
ERASE 50
exec.bmap 87
exec_lib 87
Exit (ReturnValue) 553
EXOR 340
Extra Halfbrite 155
EXTRA_HALFBRITE 418

fd 67
fill 30, 347
fill modes 357
fill patterns 354
filled rectangles 17
First ClipRect 192
First window 115
Flags 194
flags 72, 116
Flood (&RastPort, Mode, x, y) 535
Flood function 347
Font attribute 215
Font Data 216
Font kern 217
Font Style Flags 390
font 213
font reader 237
font style 392
fonts 219, 234, 391
Fore/background gadget 99
FreeColorMap (&ColorMap) 535
FreeColorMap() 170
FreeCopList (&CopList) 536
FreeMem (&Memory, Size) 554
FreeMem() 157, 204
FreeRaster() 170, 204, 536
FreeSprite (SprNumber) 536
FreeSprite (Status) 436
FreeVPortCopLists (&ViewPort) 537
FreeVPortCopLists() 170
function plotter 12

gadget structure 103
GEL 464
GEL list 464, 527
GELGONE 464
GelsInfo 124
GelsInfo structure 483
GelsInfo.lastcolor 452
Genlock Video 155
GET 46, 71
GetColorMap (NumberColors) 537
GetColorMap 170, 174
GetMsg (&Port) 554
GetRGB4 537
GetSprite 436, 452, 537
GfxBase 316
GfxBase routines 527
GfxBase structure 324
GimmeZeroZero parameters 105
GimmeZeroZero windows 99, 101
graphic cursor 9
graphic elements (GELS) 465
graphic library 141
graphic primitives 122, 129, 381
graphic.library 316, 345
graphics.bmap 87
graphics_lib.fd 87
Guru Mediation 93, 400

Halfbrite 156, 320, 418
HalfBriteOff 160
HalfBriteOn 160
HAM 155, 320
HAM mode 161, 419, 421
hardcopies 259
Hardcopy I 262
Hardcopy III 268, 271
Hardcopy II 265
Hardcopy V (Multitasking) 274
hardware registers 181
hardware sprites 435
hardware sprites 451
hexadecimal 35
Hi-Res 155
hi-res mode 415
hi-resolution graphics 1
Hitmask 457
hold-and-modify 161, 419

I/O Dump Rastport Request 259
IDCMP Flags 104
IFF 279
ILBM files 291
ILBM graphic 280
include files 503
InitArea 538
InitBitMap 538
InitBitMap() 170, 204
InitCop 181, 182
InitGels 538
InitMasks (&VSprite) 539
InitRastPort (&RastPort) 539
InitTmpRas 539
InitView (&View) 540
InitView 170, 173
InitVPort (&ViewPort) 540
InitVPort 170, 174
INPUT 50
Interchange File Format 279
Interlaced mode 155, 414
Intuition 93
Intuition screen 476
intuition.bmap 87
intuition_lib 87
IntuitionBase routines 551
INVERSVID 125, 341
Inverting graphics 52
IODRPReg 259
IODRPReg structure 261

JAM1 341
JAM2 341

Kickstart 316

LACE flag 415
Lattice V3.10 compiler 315
Layer 123
Layer data structure 191
Layerbackdrop 196
LayerInfo 117
Layers 188
layers libraries 190
layers.bmap 87
layers_lib 87
Layersimple 196
Layersmart 196
layersuper 194, 196
LIBRARY CLOSE 88
LIBRARY statement 88
library 87, 527
LINE 10, 19, 30
Line pattern 126
LoadRGB4 170, 174, 540
LoadView (&View) 323, 541
LoadView 170, 174
Lock-Fields 194
LockLayer 194
long field 95

MakeDoubleBuffer 179
MakeScreen (&Screen) 552
MakeVPort (&View, &ViewPort) 541
MakeVPort 174, 323
MakeVPort() 170
MeMask 457
menu header 100
Message 251
Message Port 104, 195, 260
message ports 195
Messages 195
Minterms 127, 401
ModifyIDCMP 552

Modulo 216, 412
Moire patterns 10
MOUSE function 2
mouse coordinates 97, 116
mouse pointer 106
MOVE 180
Move (&RastPort, X, Y) 541
Move() 141, 204
MoveSprite 541
MoveWindow 109
MrgCop (&View) 542
MrgCop 170, 174
multi-colored patterns 140, 355

NewWindow structure 345
Next screen 115
Next ViewPort 153, 168
NextSeq 480
NTSC video 4

OBJECT statements 74
OBJECT-SHAPE 56, 71
OBJECT.AX/Y 74
OBJECT.CLIP 78, 79
OBJECT.HIT 76, 79
OBJECT.ON 74
OBJECT.PRIORITY 79
OBJECT.SHAPE 74
OBJECT.START 74
OBJECT.VX/Y 74
OBJECT.X/Y 74
OBJEDIT program 56
ON ERROR GOTO 16
OPEN 50
OpenDiskFont (&TextAttr) 551
OpenFont (&TextAttr) 542
OpenFont 391
OpenLibrary 554
OpenLibrary 316
OpenScreen (&NewScreen) 552
OPTION BASE 38
OR 53
Overlay flag 75
Overscan 414
OwnBlitter () 542

PAINT 27, 30
Paint-1024 program instructions 311
PAL systems 4
palette 41, 42, 43, 80, 318, 382
parameters 4
PATTERN 35, 129
Patterned areas 34, 35
PEEK 95
PEEKL 95
PEEKW 95
pentagram 32
PFBA 155
pixel 1
PlaneOnOff 82
PlanePick 81
POINT statement 7
Pointer to Window 206
POKE 95
POKEL 95
POKEW 95
PolyDraw 543
Preferences 390
PRESET 3, 6
PrevSeq 480
printer.device 259
proportional font 217
PSET 1, 6
pseudo menus 15
PUT 46, 71
PUT-PRESET 53
PUT-PSET 53

quix 11

RasInfo 167
RasInfo Block 154
RasInfo structure 322, 476
RastPort 100, 117, 122, 148, 192, 344, 412
RastPort border 101
RastPort structure 122, 123, 188, 323
ReadPixel (&RastPort, X, Y) 543
ReadPixel 90
rectangle 127, 349
RectFill 543
refresh mode 99
relative address 9, 30
ReMakeDisplay 157, 160,477, 552

RemFont (&TextFont) 544
RemIBob 544
RemVSprite (&VSprite) 544
ReplyMsg (&Port) 555
ReplyPort 278
Requester handling 100
requester 197
resolution 413
RethinkDisplay () 553
RINGTRIGGER 481
RingXTrans 483
RingYTrans 483
ROM routines 316
ROM versions 316

SADD 88
SaveBack flag 72
SaveBob 75
SaveBuffer 477
Scaling 16
Screen dimensions 116
Screen structure 115, 120
screen 68
screen colors 117
screen gadgets 117
screen title 105
Screen/Intuition 114
ScreenBye 118
ScreenHere 118
Scrolling 195
ScrollLayer() 203, 205
ScrollRaster 544
ScrollScreen 118
ScrollVPort (&ViewPort) 545
SetAPen (&RastPort, Colorregister) 545
SetBPen (&RastPort, Colorregister 545
SetCollision 546
SetCollision 484
SetDrMd (&RastPort, Mode) 546
SetDrMd 125, 141
SetFont (&RastPort, &Font) 546
SetPointer 106
SetRast (&RastPort, Colorregister) 547
SetRast 204
SetRG4CM 547
SetRGB4 547
SetSoftStyle 548
SetWindowTitles 204, 205
Shadow mask 75
Simple Refresh (Layersimple) 196
SimpleSprite structure 435
SizeWindow 111
sizing gadget 96, 99
SKIP 180
Smart Refresh (Layersmart) 196
Smart refresh 99
Software Failure 400
SortGList (&RastPort) 548
SortGList 455
Spr.Color 85, 454
Sprite collisions 440
sprite 55
sprite image 104
SPRITE_ATTACHED 439
SpriteData structure 439
Sprites 155, 435, 438, 439, 495, 496
STEP 9
struct AnimComp 505
struct AnimOb 506
struct AreaInfo 508
struct AvailFonts 508
struct AvailFontsHeader 508
struct BitMap 508
struct Bob 509
struct ColorMap 511
struct Custom 511
struct DBuffPacket 511
struct GelsInfo 512
struct GfxBase 511
struct IntuiMessage 513
struct IntuitionBase 513
struct NewScreen 515
struct NewWindow 513
struct RasInfo 519
struct RastPort 516
struct Screen 519
struct SimpleSprite 520
struct TextAttr 520
struct TextFont 521
struct View 521
struct ViewPort 522
struct VSprite 523
struct Window 525

StyleEnable 392
Superbit 99
Superbitmap (Layersuper) 196
Superbitmap 194, 204, 293
Superbitmap Paint Program 294
SuperClipRect 194
supergraphic 211
SWAP-assignments 40
switch elements 103
synchronous 274
system components 212
system crash 90
system data structure 212
system libraries 88

Text 548
Text height 127
text length 389
text styles 144
Text() 141
TextAtt 218
TextAttr structure 390, 393
TextFont 214
TextFont data structure 215
TextFont structure 220, 235, 250
TextLength 548
TIMER 29
Timer 481
TimeSet 483
Title text 100
TmpRas 124
TmpRas 207
topaz 8 213
topaz 9 213
topaz font 221
transmit 179

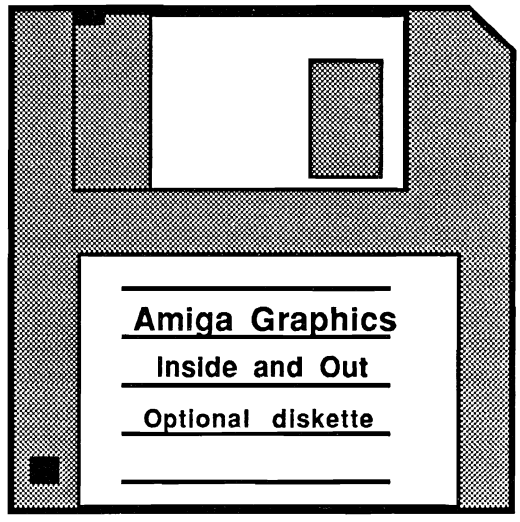
UCopList 181
Undefined function values 16
UnLockLayer 194
user Copper list 181
user data 117
UserCopperList 495

VBeamPos () 549
View 167
View 317, 413
View structure 173
ViewAddress 168
ViewPort 117, 151, 166, 188, 317, 323, 344, 381, 413
ViewPort Modes 153
ViewPort structure 153, 318
ViewPorts 345
VP_HIDE 155, 435
VSprite 459
VSprite flag 464, 466
VSprite structure 456, 466, 453
VSprite variables 467
VSprite.OldX 477
VSprite.OldY 477
VSprites 435, 451, 453, 456, 458, 465, 467 495

WAIT 180
WaitBlit () 549
WaitBOVP (&ViewPort) 549
WaitTOF 204, 205, 549
wd 67
width table 217
WINDOW OUTPUT 95
Window 93
Window borders 101
Window colors 105
Window limits 98
Window modes 99
window data structure 96, 120
window size 96
WINDOW(7) 93, 103
WINDOW(8) 122, 214
WindowLimits 110
windows 103, 345
WindowToBack 113
WindowToFront 113
word field 95
WRITE 50
Writemask 124
WritePixel (&RastPort, X, Y) 550
WritePixel (&RastPort, x, y) 327

XOR 51

Optional Diskette



For your convenience, the program listings contained in this book are available on an Amiga 3 1/2" formatted floppy disk. You should order the diskette if you want to use the programs, but don't want to type them in from the listings in the book.

All programs on the diskette have been fully tested. You can change the programs for your particular needs. The diskette is available for $14.95 plus $2.00 ($5.00 foreign) for postage and handling.

When ordering, please give your name and shipping address. Enclose a check, money order or credit card information. Mail your order to:

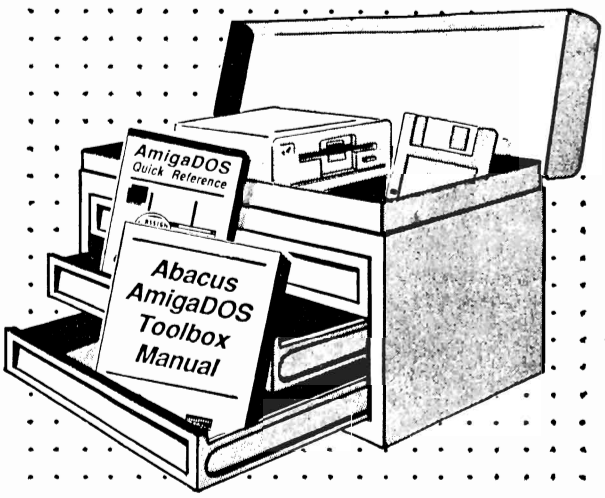
Abacus Software
5370 52nd Street SE
Grand Rapids, MI 49512

Or for fast service,
Call Toll Free **1-800-451-4319**

Abacus AmigaDOS® Toolbox

Essential software tools for all Amiga users.

NEW FOR ALL AMIGAS!



A collection of essential, powerful and easy-to-use tools.

The Abacus AmigaDos® ToolBox includes 11 new fonts, new handy CLI commands, a disk copier, floppy disk speeder, screen capture utility and more.

Suggested Retail Price: $59.95

from a company you can count on

Abacus

5370 52nd Street SE • Grand Rapids, MI 49508 • Phone (616) 698-0330 • Telex 709-101 • Facsimile (616) 698-0325

New Software

All Abacus software runs on the Amiga 500, Amiga 1000 or Amiga 2000. Each package is fully compatible with our other products in the Amiga line

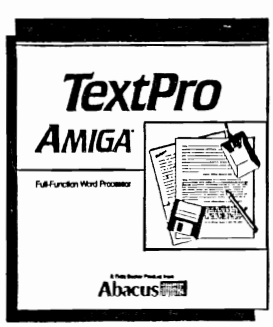
The *Ideal* AMIGA wordprocessor

TextPro *AMIGA*

TextPro AMIGA upholds the true spirit of the AMIGA: it's powerful, it has a surprising number of "extra" features, but it's also very easy to use. **TextPro AMIGA**—the Ideal AMIGA word processor that proves just how easy word processing can be. You can write your first documents immediately, with a minimum of learning—without even reading the manual. But **TextPro AMIGA** is much more than a beginner's package. Ultra-fast onscreen formatting, graphic merge capabilities, automatic hyphenation and many more features make **TextPro AMIGA** ideal for the professional user as well. **TextPro AMIGA** features:

- High-speed text input and editing
- Functions accessible through menus or shortcut keys
- Fast onscreen formatting
- Automatic hyphenation
- Versatile function key assignment
- Save any section of an AMIGA screen & print as text
- Loading and saving through the RS-232 interface
- Multiple tab settings
- Accepts IFF format graphics in texts
- Extremely flexible printer adaptations. Printer drivers for most popular dot-matrix printers included
- Includes thorough manual
- Not copy protected

TextPro AMIGA sets a new standard for word processing packages in its price range. So easy to use and modestly priced that any AMIGA owner can use it—so packed with advanced features, you can't pass it up.



Suggested retail price: **$79.95**

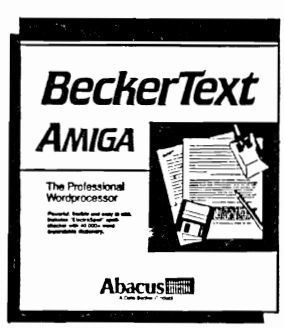
More than word processing...

BeckerText *AMIGA*

This is one program for serious AMIGA owners. **BeckerText Amiga** is more than a word processor. It has all the features of **TextPro AMIGA**, but it also has features that you might not expect:

- Fast WYSIWYG formatting
- Calculations within a text—like having a spreadsheet program anytime you want it
- Templates for calculations in columns
- Line spacing options
- Auto-hyphenation and Auto-indexing
- Multiple-column printing, up to 5 columns on a single page
- Online dictionary checks spelling in text as it's written
- Spell checker for interactive proofing of documents
- Up to 999 characters per line (with scrolling)
- Many more features for the professional

BeckerText AMIGA is a vital addition for C programmers—it's an extremely flexible C editor. Whether you're deleting, adding or duplicating a block of C source-code, **BeckerText AMIGA** does it all, automatically. And the online dictionary acts as a C syntax checker and finds syntax errors in a flash.



BeckerText AMIGA. When you need more from your word processor than just word processing.

Suggested retail price: **$150.00**

Imagine the perfect database

DataRetrieve
AMIGA

Imagine, for a moment, what the perfect database for your AMIGA would have. You'd want power and speed, for quick access to your information. An unlimited amount of storage space. And you'd want it easy to use—no baffling commands or file structures—with a graphic interface that does your AMIGA justice.

Enter **DataRetrieve AMIGA**. It's unlike any other database you can buy. Powerful, feature-packed, with the capacity for any business or personal application—mailing lists, inventory, billing, etc. Yet it's so simple to use, it's startling. **DataRetrieve AMIGA**'s drop-down menus help you to define files quickly. Then you conveniently enter information using on-screen templates. **DataRetrieve AMIGA** takes advantage of the Amiga's multi-tasking capability for *optimum* processing speed.

DataRetrieve AMIGA features:

- Open eight files simultaneously
- Password protection
- Edit files in memory
- Maximum of 80 index fields with variable precision (1-999 characters)
- Convenient search/select criteria (range, AND/OR comparisons)
- Text, date, time, numeric and selection fields, IFF file reading capability
- Exchange data with other software packages (for form letters, mailing lists, etc.)
- Control operations with keyboard or mouse
- Adjustable screen masks, up to 5000 x 5000 pixels
- Insert graphic elements into screen masks (e.g., rectangles, circles, lines, patterns, etc.)
- Screen masks support different text styles and sizes
- Multiple text fields with word make-up and formatting capabilities
- Integrated printer masks and list editor.
- Maximum filesize 2 billion characters
- Maximum data record size 64,000 characters
- Maximum data set 2 billion characters
- Unlimited number of data fields
- Maximum field size 32,000 characters

DataRetrieve AMIGA —it'll handle your data with the speed and easy operation that you've come to expect from Abacus products for the AMIGA.

Suggested retail price: **$79.95**

Not just for the experts

AssemPro
AMIGA

AssemPro AMIGA lets every Amiga owner enjoy the benefits of fast machine language programming.

Because machine language programming isn't just for 68000 experts. **AssemPro AMIGA** is easily learned and user-friendly—it uses Amiga menus for simplicity. But **AssemPro AMIGA** boasts a long list of professional features that eliminate the tedium and repetition of M/L programming. **AssemPro AMIGA** is the complete developer's package for writing of 68000 machine language on the Amiga, complete with editor, debugger, disassembler and reassembler. **AssemPro AMIGA** is the perfect introduction to machine langage development and programming. And it's even got what you 68000 experts need.

AssemPro AMIGA features:

- Written completely in machine language, for ultra-fast operation
- Integrated editor, debugger, disassembler, reassembler
- Large operating system library
- Runs under CLI and Workbench
- Produces either PC-relocatable or absolute code
- Macros possible for nearly any parameter (of different types)
- Error search function
- Cross-reference list
- Menu-controlled conditional and repeated assembly
- Full 32-bit arithmetic
- Debugger with 68020 single-step emulation
- Runs on any AMIGA with 512K and Kickstart 1.2.

Suggested retail price: **$99.95**

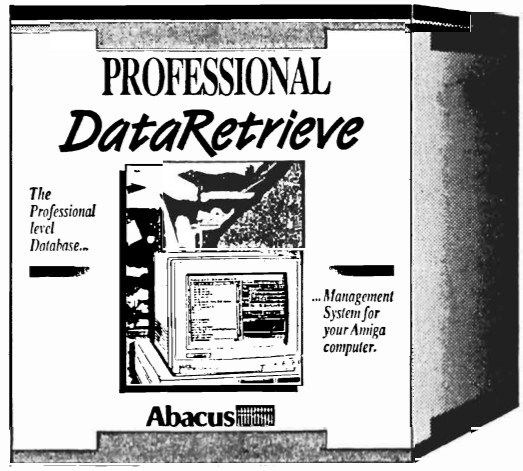
Abacus Products for *Amiga* computers

Professional DataRetrieve

The Professional Level Database Management System

Professional DataRetrieve, for the Amiga 500/1000/2000, is a friendly easy-to-operate professional level data management package with the features most wanted in a relational data base system.
Professional DataRetrieve has complete relational data mangagement capabilities. Define relationships between different files (one to one, one to many, many to many). Change relations without file reorganization.

Professional DataRetrieve includes an extensive programming laguage which includes more than 200 BASIC-like commands and functions and integrated program editor. Design custom user interfaces with pulldown menus, icon selection, window activation and more.
Professional DataRetrieve can perform calculations and searches using complex mathematical comparisons using over 80 functions and constants.



Professional DataRetrieve is a friendly, easy to operate programmable RELATIONAL data base system. **PDR** includes PROFIL, a programming language similar to BASIC. You can open and edit up to 8 files simultaneously and the size of your data fields, records and files are limited only by your memory and disk storage. You have complete interrelation between files which can include IFF graphics. NOT COPY PROTECTED. ISBN 1-55755-048-4

MORE features of **Professional DataRetrieve**

Easily import data from other databases....file compatible with standard **DataRetrieve**....supports multitasking...design your own custom forms with the completely integrated printer mask editor....includes PROFIL programming language that allows the programmer to custom tailor his database requirements...

MORE features of PROFIL include:

Open Amiga devices including the console, printer, serial and the CLI.
Create your own programmable requestors
Complete error trapping.
Built-in compiler and much, much more.

Suggested retail price: **$295.00**

Features

- Up to 8 files can be edited simultaneously
- Maximum size of a data field 32,000 characters (text fields only)
- Maximum number of data fields limited by RAM
- Maximum record size of 64,000 characters
- Maximum number of records disk dependent (2,000,000,000 maximum)
- Up to 80 index fields per file
- Up to 6 field types - Text, Date, Time, Numeric, IFF, Choice
- Unlimited number of searches and subrange criteria
- Integrated list editor and full-page printer mask editor
- Index accuracy selectable from 1-999 characters
- Multiple file masks on-screen
- Easily create/edit on-screen masks for one or many files
- User-programmable pulldown menus
- Operate the program from the mouse or the key board
- Calculation fields, Data Fields
  IFF Graphics supported
- Mass-storage-oriented file organization
- Not Copy Protected, NO DONGLE; can be in stalled on your hard drive

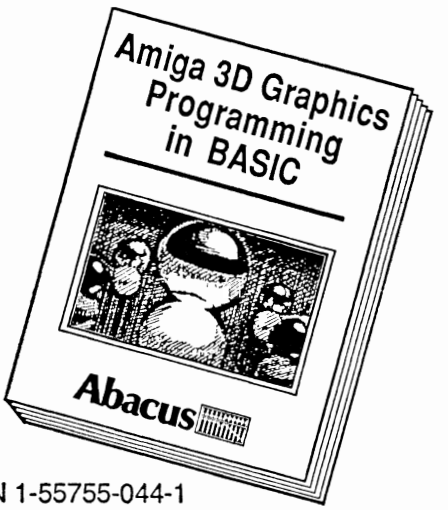
Books for the AMIGA

Amiga 3-D Graphics Programming in BASIC

Shows you how to use the powerful graphics capabilities of the Amiga. Details the techniques and algorithm for writing three-dimensional graphics programs: ray tracing in all resolutions, light sources and shading, saving graphics in IFF format and more.

Topics include:

- Basics of ray tracing
- Using an object editor to enter three-dimensional objects
- Material editor for setting up materials
- Automatic computation in different resolutions
- Using any Amiga resolution (low-res, high-res, interface, HAM)
- Different light sources and any active pixel
- Save graphics in IFF format for later recall into any IFF compatible drawing program
- Mathematical basics for the non-mathematition

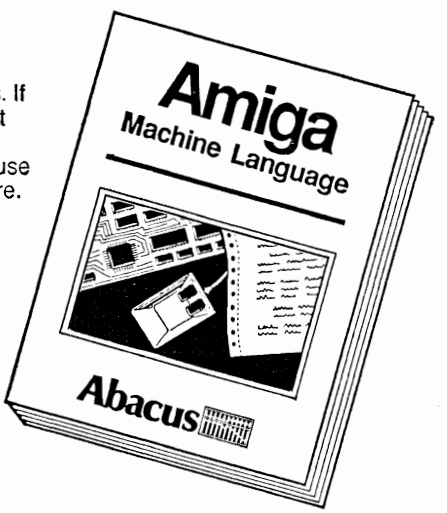


Volume 3 Suggested retail price $19.95 ISBN 1-55755-044-1

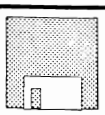
Amiga Machine Language

Amiga Machine Language introduces you to 68000 machine language programming presented in clear, easy to understand terms. If you're a beginner, the introduction eases you into programming right away. If you're an advance programmer, you'll discover the hidden powers of your Amiga. Learn how to access the hardware registers, use the Amiga libraries, create gadgets, work with Intuition and much more.

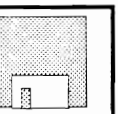
- 68000 address modes and instruction set
- Accessing RAM, operating system and multitasking capabilities
- Details the powerful Amiga libraries for using AmigaDOS
- Speech and sound facilities from machine language
- Simple number base conversions
- Text input and output
- Checking for special keys
- Opening CON: RAW: SER: and PRT: devices
- New directory program that doesn't access the CLI
- Menu programming explained
- Complete Intuition demonstration program including Proportional, Boolean and String gadgets.



Volume 4 Suggested retail price $19.95 ISBN 1-55755-025-5

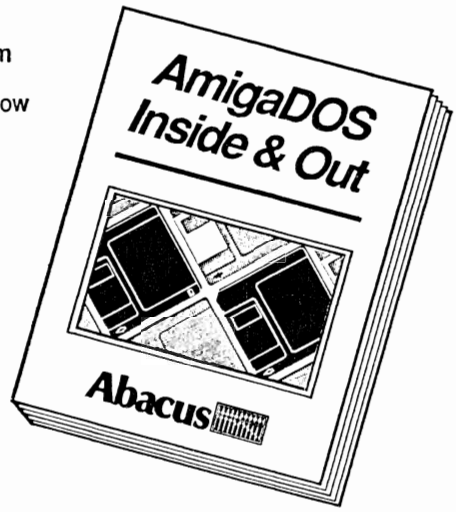


Save Time and Money!-Optional program disks are available for all our Amiga reference books (except Amiga for Beginners). All programs listed in the book are on each respective disk and will save you countless hours of typing! $14.95



Books for the AMIGA

AmigaDOS: Inside & Out



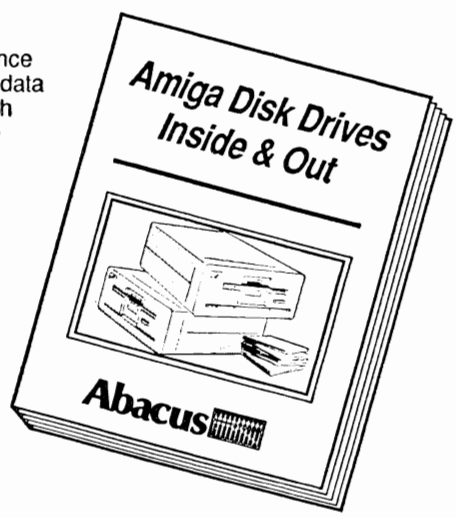
AmigaDOS: Inside & Out-covers the insides of AmigaDOS from the internal design up to practical applications. Includes detailed reference section, tasks and handling, DOS editors ED and EDIT, how to create and use batch files, multitasking, and much more.

Topics include:

- 68000 microprocessor architecture
- AmigaDOS - Tasks and handling
- Detailed explanations of CLI commands and their functions
- DOS editors ED and EDIT
- Operating notes about the CLI (Wildcards, shortening input and output)
- Amiga devices and how the CLI uses them
- Batch files - what they are and how to write them
- Changing the startup sequence
- AmigaDOS and multitasking
- Writing your own CLI commands
- Reference to the CLI, ED and EDIT commands
- Resetting priorities - the TaskPri command
- Protecting your Amiga from unauthorized use

Volume 8 Suggested retail price $19.95 ISBN 1-55755-041-7

Amiga Disk Drives: Inside & Out



Amiga Disk Drives: Inside & Out is the most in-depth reference available covering the Amiga's disk drives. Learn how to speed up data transfer, how copy protection works, computer viruses, Workbench and the CLI DOS functions, loading, saving sequential, relative file organization, more.

Topics include:

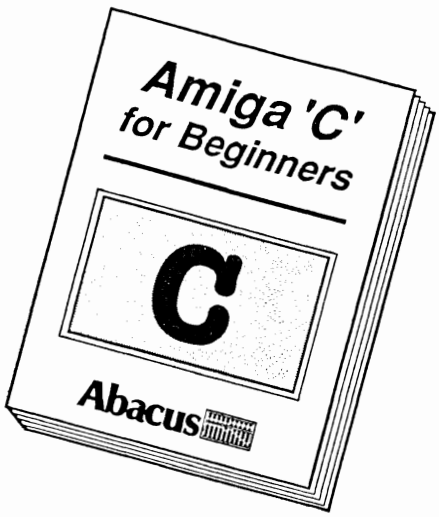
- Floppy disk operation from the Workbench and CLI
- BASIC: Loading, saving, sequential and relative files DOS functions
- File management: Block types, boot blocks, checksums, file headers, hashmarks and protection methods
- Viruses: Protecting your boot block
- Trackdisk.device: Commands, structures
- Trackdisk-task: Function and design
- Diskette access with DOS
- MFM, GCR, Track design, blockheader, data blocks, checksums, coding and decoding data , hardware registers, SYNC, and interrupts

Volume 9 Suggested retail price $29.95 ISBN 1-55755-042-5

Save Time and Money!-Optional program disks are available for all our Amiga reference books (except Amiga for Beginners and DOS Quick Reference). All programs listed in the book are on each respective disk and will save you countless hours of typing! $14.95

Books for the AMIGA

Amiga C for Beginners



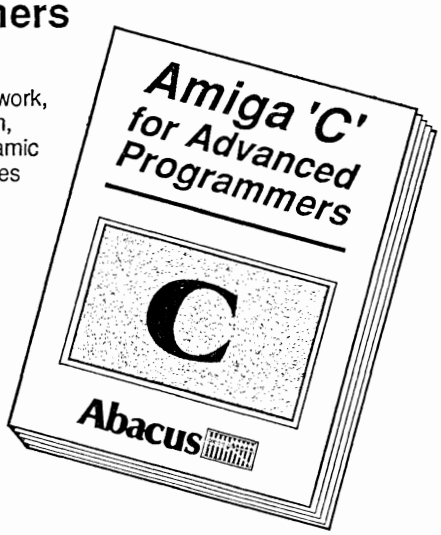
An introduction to learning the popular C language. Explains the language elements using examples specifically geared to the Amiga. Describes C library routines, how the compiler works and more.

Topics include:

- Particulars of C
- How a compiler works
- Writing your first program
- The scope of the language (loops, conditions, functions, structures)
- Special features of C
- Important routines in the C libraries
- Input/Output
- Tricks and Tips for finding errors
- Introduction to direct programming of the operating system (windows, screens, direct text output, DOS functions)
- Using the LATTICE and AZTEC C compilers

Volume 10 Suggested retail price $19.95 ISBN 1-55755-045-X

Amiga C for Advanced Programmers

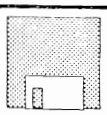


Amiga C for Advanced Programmers- contains a wealth of information from the pros: how compilers, assemblers and linkers work, designing and programming user friendly interfaces using Intuition, managing large programming projects, using jump tables and dynamic arrays, coming assembly language and C codes, and more. Includes complete source code for text editor.

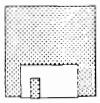
Topics include:

- Using INCLUDE, DEFINE and CASTS
- Debugging and optimizing assembler sources
- All about Intuition programming (windows, screens, pulldown menus, requesters, gadgets)
- Programming the console devices
- A professional editor's view of problems with developing larger programs
- Using MAKE correctly
- Debugging C programs with different utilities
- Folding (formatting text lines and functions for readability)

Volume 11 Suggested retail price $34.95 ISBN 1-55755-046-8



Save Time and Money!-Optional program disks are available for all our Amiga reference books (except Amiga for Beginners). All programs listed in the book are on each respective disk and will save you countless hours of typing! $14.95



One Good Book deserves another and another, and a...

New!

Amiga C for Advanced Programmers

-contains a wealth of information from the pros: how compilers, assemblers and linkers work, designing and programming user friendly interfaces using Intuition, combining assembly language and C codes, and more. Includes complete source code for text editor.

ISBN 1-55755-046-8 **$34.95**

New!

Amiga C for Beginners

-an introduction to learning the popular C language. Explains the language elements using examples specifically geared to the Amiga. Describes C library routines, how the compiler works and more.

ISBN 1-55755-045-X **$19.95**

New!

Amiga 3-D Graphic Programming in BASIC

-shows you how to use the powerful graphic capabilities of the Amiga. Details the techniques and algorithms for writing three dimensional graphic programs: ray tracing in all resolutions, light sources and shading, saving graphics in IFF format and more.

ISBN 1-55755-044-1 **$19.95**

Amiga Disk Drives Inside & Out

-is the most in-depth reference available covering the Amiga's disk drives. Learn how to speed up data transfer, how copy protection works, computer viruses, Workbench and the CLI DOS functions, loading, saving, sequential and random file organization, more.

ISBN1-55755-042-5 **$29.95**

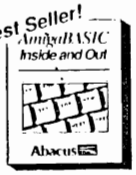
Amiga For Beginners *

-the first volume in our Amiga series, introduces you to Intuition (Amiga's graphic interface), the mouse, windows, the CLI and Amiga BASIC and explains every practical aspect of the Amiga in plain English.

ISBN 1-55755-021-2 **$16.95**

Includes Workbench 1.3

Best Seller!

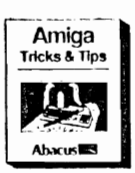


AmigaBASIC Inside and Out

-THE definitive step by-step guide to programming the Amiga in BASIC. Every AmigaBASIC command is fully described and detailed. Topics include charts, windows, pulldown menus, files, mouse and speech commands.

ISBN 0-916439-87-9 **$24.95**

Includes Workbench 1.3



Amiga Tricks and Tips

-follows our tradition of other Tricks and Tips books for CBM users. Presents dozens of tips on accessing libraries from BASIC, custom character sets, AmigaDOS, sound, important 68000 memory locations, and much more!

ISBN 0-916439-88-7 **$19.95**

AmigaDOS Inside and Out

-covers the insides of AmigaDOS from the internal design up to practical applications. Includes detailed reference section, tasks and handling, DOS editors ED and EDIT, how to create and use script files, multitasking, and much more.

ISBN 1-55755-041-7 **$19.95**

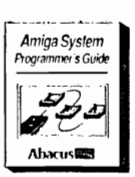
Includes Workbench 1.3

Amiga Machine Language

is a comprehensive introduction to 68000 assembler machine language programming and is THE practical guide for learning to program the Amiga in ultra-fast ML. Also covers 68000 microprocessor address modes and architecture, speech and sound from ML and much more.

ISBN 1-55755-025-5 **$19.95**

Amiga System Programmer's Guide



-comprehensive guide to what goes on inside the Amiga in a single volume. Only a few of the many subjects covered include the EXEC structure, I/O requests, interrupts and resource management, multitasking functions and much, much more.

ISBN 1-55755-034-4 **$34.95**

AmigaDOS Quick Reference *

New!

-an easy-to-use reference tool for beginners and advanced programmers alike. You can quickly find commands for your Amiga by using the three handy indexes designed with the user in mind. All commands are in alphabetical order for easy reference. *Includes Workbench 1.3*

ISBN 1-55755-049-2 **$9.95**

Computer Viruses: a high-tech disease *

-describes what a computer virus is, how viruses work, viruses and batch files, protecting your computer, designing virus proof systems and more.

ISBN 1-55755-043-3 **$18.95**

"Probably the best and most current book about preventive measures"
PC Week 11-21-88

Save Time and Money!-Optional program disks are available for many of our Amiga reference books. All programs listed in the books are on each respective disk and will save you countless hours of typing! **$14.95**

(* Optional Diskette Not Available for these Titles)

Abacus

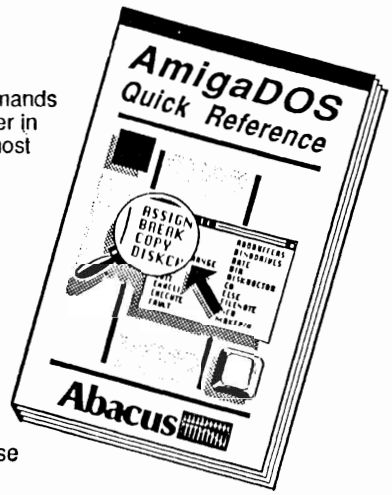
5370 52nd Street SE, Grand Rapids MI 49512

See your local Dealer or Call Toll Free 1-800-451-4319

Add $4.00 Shipping per Order
Foreign add $12.00 per item

Books for the AMIGA

AmigaDOS Quick Reference Guide



AmigaDos Quick Reference Guide is an easy-to-use reference tool for beginners and advanced programmers alike. You can quickly find commands for your Amiga by using the three handy indexes designed with the user in mind. All commands are in alphabetical order for easy reference. The most useful information you need fast can be found- including:

- All AmigaDOS commands described, including Workbench 1.3
- Command syntax and arguments described with examples
- CLI shortcuts
- CTRL sequences
- ESCape sequences
- Amiga ASCII table
- Guru Meditation Codes
- Error messages with their corresponding numbers

Three indexes for quick information at your fingertips! The AmigaDOS Quick Reference Guide is an indispensable tool you'll want to keep close to your Amiga.

Suggested retail price $9.95 ISBN 1-55755-049-2

Abacus Amiga Books

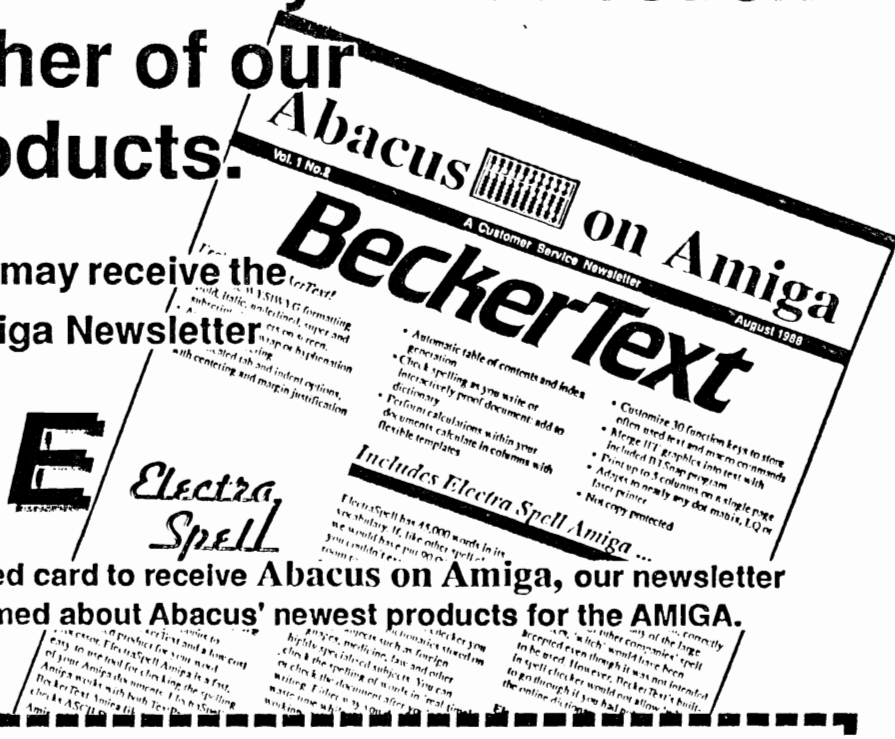
Book Title	ISBN No.	PRICE
Vol. 1 Amiga for Beginners	1-55755-021-2	$16.95
Vol. 2 AmigaBASIC Inside & Out	0-916439-87-9	24.95
Vol. 3 Amiga 3D Graphic Prg'ing in BASIC	1-55755-044-1	19.95
Vol. 4 Amiga Machine Language	1-55755-025-5	19.95
Vol. 5 Amiga Tricks & Tips	0-916439-88-7	19.95
Vol. 6 Amiga System Prg'ers Guide	1-55755-034-4	34.95
Vol. 7 Adv'd System Prg'ers Guide	1-55755-047-6	34.95
Vol. 8 AmigaDOS Inside & Out	1-55755-041-7	19.95
Vol. 9 Amiga Disk Drives Inside & Out	1-55755-042-5	29.95
Vol. 10 Amiga C for Beginners	1-55755-045-X	19.95
Vol. 11 Amiga C for Advanced Prg'ers	1-55755-046-8	34.95
Vol. 12 More Tricks & Tips	1-55755-051-4	19.95
Vol. 13 Amiga Graphics Inside & Out	1-55755-052-2	34.95
AmigaDOS Quick Reference Guide	1-55755-049-2	9.95

Save Time and Money!-Optional program disks are available for all our Amiga reference books (except Amiga for Beginners and DOS Quick Reference). All programs listed in the book are on each respective disk and will save you countless hours of typing! $14.95

We appreciate your selection of another of our fine products.

In addition you may receive the Abacus on Amiga Newsletter

FREE



Return this completed card to receive Abacus on Amiga, our newsletter that keeps you informed about Abacus' newest products for the AMIGA.

Reserve your copy now!

Abacus on Amiga

Abacus
5370 52nd Street S.E., Grand Rapids, MI 49512
(616) 698-0330

Name _____

Address _____

City _____ State _____ Zip _____

Phone ( ) _____

Where did you purchase your Abacus Amiga Product ? _____

What other Abacus Products would you be interested in? _____

☐ Please send me additional information on other Amiga products.

Amiga® Graphics Inside & Out

Amiga users are well aware of the outstanding graphic capabilities of their computer. What isn't well known is how to harness all these amazing and powerful graphic capabilities. ***Amiga Graphics Inside & Out*** shows you simply and in plain English how to use the super graphic features and functions of the Amiga. You'll learn how to access the graphic features from both AmigaBASIC and C.

For the beginner, we present examples demonstrating AmigaBASIC graphic commands to draw points, lines, circles, rectangles, patterns, flood fill and more.

Amiga Graphics Inside & Out contains many sample programs to demonstrate how you can create your own Amiga graphic code that you can use in your own programs. You'll learn new ways to access the Amiga libraries from BASIC, how to display 4096 colors at once, color patterns, screen and window dumps to your printer and more.

If you're an advanced user you'll see how to use the graphic routines from the Amiga's built-in graphic libraries. You'll also learn graphic programming in C with examples of points, lines, rectangles, polygons, colors and more. ***Amiga Graphics Inside & Out*** contains a complete description of the Amiga graphic system - View, ViewPort, RastPort, bitmap mapping, screens, and windows. All these and more are presented in an easy to understand format.

Amiga Graphics Inside & Out–
the most thorough guide to your Amiga's graphic powers.

A comprehensive book for understanding and using Amiga graphics

Topics include:

- Complete description of the Amiga graphic system - View, ViewPort, Rast Port, bitmap mapping, screens, windows, and much more
- AmigaBASIC graphic commands - points, lines, circles, rectangles, patterns, flood fill
- Accessing fonts and type styles in AmigaBASIC, Loading and saving IFF graphics
- CAD on a 1024 x 1024 super bitmap, Using graphic library routines
- New ways to access libraries and chips from BASIC - 4096 colors at once, color patterns, screen and window dumps to printer
- Graphic programming in C - points, lines, rectangles, polygons, colors
- Amiga animation explained including sprites, bobs and AnimObs, Copper and blitter programming
- Gridline interrupts and fast copying

Optional Program Diskette Available:
Contains every program listed in the book– complete, error-free and ready to run! Saves you hours of typing in the program listings.

ISBN 1-55755-052-2
90000
9 781557 550521

Abacus
5370 52nd Street SE • Grand Rapids, MI 49512