ACM - Amiga C Manual

Anders Bjerin

Book One

Part I: Introduction, Screens, Windows, Graphics, Gadgets
Part II: Requesters, Alerts, Menus, IDCMP, Miscellaneous
Part III: Sprites, AmigaDOS, Low Level Graphics, Virtual Sprites, Hints and Tips

http://aminet.net/package/dev/c/ACM

The complete boiled-down C manual for the Amiga which describes how to open and work with Screens, Windows, Graphics, Gadgets, Requesters, Alerts, Menus, IDCMP, Sprites, VSprites, AmigaDOS, Low Level Graphics Routines, Hints and Tips, etc. The manual also explains how to use your C Compiler and gives you important information about how the Amiga works and how your programs should be designed. The manual consists of 15 chapters together with more than 100 fully executable examples with source code.

0 INTRODUCTION 10
0.1 INTRODUCTION 10
0.2 HOW TO COMPILE AND LINK 10
0.2.1 SOURCE CODE 11
0.2.2 FIRST PHASE OF COMPILATION 11
0.2.3 SECOND PHASE OF COMPILATION 11
0.2.4 LINKING 12
0.3 C PROGRAMS 13
0.3.1 #INCLUDE 13
0.3.2 #DEFINE 14
0.3.3 OTHER PRE-PROCESSOR COMMANDS 16
0.3.4 FUNCTIONS 17
0.3.5 VARIABLES 18
0.3.6 STORAGE CLASSES FOR VARIABLES 19
0.3.6.1 AUTOMATIC 19
0.3.6.2 FORMAL 20
0.3.6.3 GLOBAL 21
0.3.6.4 EXTERNAL STATIC 22
0.3.6.5 INTERNAL STATIC 22
0.3.6.6 REGISTER 23
0.3.7 POINTERS 23
0.3.8 STRUCTURES 23
0.3.8.1 HOW TO DECLARE STRUCTURES 24
0.3.8.2 HOW TO CHANGE THE STRUCTURE'S FIELDS 24
0.3.8.3 POINTERS AND STRUCTURES 25
0.3.9 CASTING 25
0.4 LIBRARIES 26
0.4.1 ROM LIBRARIES 26
0.4.2 DISK LIBRARIES 27
0.4.3 OPEN AND CLOSE LIBRARIES 27
0.5 MEMORY 29
1 SCREENS 31
1.1 INTRODUCTION 31
1.2 DIFFERENT TYPES OF SCREENS 31
1.3 WORKBENCH SCREEN 31
1.4 CUSTOM SCREENS 32
1.4.1 RESOLUTION 32
1.4.2 DEPTH 32
1.4.3 INTERLACED 33
1.4.4 HAM AND EXTRA HALFBRIGHTE 33
1.4.5 DUAL PLAYFIELDS 33
1.4.6 FONTS 33
1.4.7 SIZE AND POSITION 34
1.4.8 TITLE 34
1.4.9 GADGETS 34
1.5 INITIALIZE A CUSTOM SCREEN 35

1.6 OPEN A CUSTOM SCREEN 36
1.7 SCREEN STRUCTURE 37
1.8 FUNCTIONS 38
1.9 EXAMPLES 41
2 WINDOWS 43
2.1 INTRODUCTION 43
2.2 SPECIAL WINDOWS 43
2.2.1 BACKDROP WINDOWS 43
2.2.2 BORDERLESS WINDOWS 44
2.2.3 GIMMEZEROZERO WINDOWS 44
2.2.4 SUPERBITMAP WINDOWS 44
2.3 SYSTEM GADGETS 45
2.4 REDRAWING THE WINDOW DISPLAY 46
2.5 INITIALIZE A WINDOW 46
2.6 OPEN A WINDOW 50
2.7 WINDOW STRUCTURE 51
2.8 OPEN A SUPERBITMAP WINDOW 52
2.9 MAKE YOUR OWN CUSTOM POINTER 53
2.10 FUNCTIONS 57
2.11 EXAMPLES 61
3 GRAPHICS 64
3.1 INTRODUCTION 64
3.2 LINES TEXT PICTURES 64
3.3 BORDERS 64
3.3.1 THE BORDER STRUCTURE 65
3.3.2 HOW TO USE THE BORDER STRUCTURE 66
3.4 TEXT 67
3.4.1 THE INTUITEXT STRUCTURE 67
3.4.2 FONTS 68
3.4.3 HOW TO USE THE INTUITEXT STRUCTURE 70
3.5 IMAGES 70
3.5.1 IMAGE DATA 71
3.5.2 THE IMAGE STRUCTURE 74
3.5.3 PLANEPICK 75
3.5.4 PLANEONOFF 76
3.5.5 HOW TO USE THE IMAGE STRUCTURE 76
3.6 FUNCTIONS 77
3.7 EXAMPLES 79
4 GADGETS 81
4.1 INTRODUCTION 81
4.2 DIFFERENT TYPES OF GADGETS 81
4.3 CUSTOM GADGETS 81
4.3.1 GRAPHICS FOR CUSTOM GADGETS 81
4.3.2 POSITION 82
4.3.3 SIZE 82
4.4 INITIALIZE A CUSTOM GADGET 83
4.5 BOOLEAN GADGET 88

4.6 STRING/INTEGER GADGET 90
4.6.1 STRINGINFO STRUCTURE 90
4.6.2 INITIALIZE A STRING/INTEGER GADGET 91
4.6.3 USING A STRING/INTEGER GADGET 92
4.7 PROPORTIONAL GADGET 93
4.7.1 PROPINFO STRUCTURE 93
4.7.2 INITIALIZE A PROPORTIONAL GADGET 95
4.8 MONITORING THE GADGETS 96
4.9 FUNCTIONS 100
4.10 EXAMPLES 104
5 REQUESTERS 108
5.1 INTRODUCTION 108
5.2 DIFFERENT TYPES OF REQUESTERS 108
5.2.1 SYSTEM REQUESTERS 108
5.2.2 APPLICATION REQUESTERS 109
5.2.3 DOUBLE-MENU REQUESTERS 109
5.3 GRAPHICS FOR REQUESTERS 109
5.4 POSITION 110
5.5 REQUESTERS AND GADGETS 110
5.6 SIMPLE REQUESTERS 111
5.7 OPEN A REQUESTERS 112
5.7.1 INITIALIZE A REQUESTER 112
5.7.2 HOW TO ACTIVATE AN APPLICATION REQUESTER 115
5.8 IDCMP FLAGS 116
5.9 FUNCTIONS 116
5.10 EXAMPLES 119
6 ALERTS 121
6.1 INTRODUCTION 121
6.2 DIFFERENT LEVELS OF WARNINGS 121
6.3 HOW TO USE THE DISPLAYALERT() FUNCTION 121
6.4 EXAMPLES OF STRINGS AND SUBSTRINGS 122
6.5 FUNCTIONS 124
6.6 EXAMPLES 125
7 MENUS 126
7.1 INTRODUCTION 126
7.2 MENU DESIGN 126
7.3 HOW TO ACCESS MENUS FROM THE KEYBOARD 127
7.4 MENU ITEMS 128
7.5 MUTUAL EXCLUDE 128
7.6 OPEN A MENU 129
7.6.1 INITIALIZE A MENU STRUCTURE 129
7.6.2 INITIALIZE A MENUITEM STRUCTURE 131
7.6.3 HOW TO SUBMIT AND REMOVE A MENU STRIP TO/FROM A WINDOW 134
7.7 SPECIAL IDCMP FLAGS 135
7.7.1 MENUPICK 135
7.7.2 MENUVERIFY 137

7.8 MENU NUMBERS 139
7.9 FUNCTIONS 140
7.10 MACROS 142
7.11 EXAMPLES 143
8 IDCMP 145
8.1 INTRODUCTION 145
8.2 IDCMP PORTS 145
8.3 HOW TO RECEIVE IDCMP MESSAGES 146
8.3.1 OPEN IDCMP PORTS 146
8.3.2 WAIT FOR MESSAGES 146
8.3.3 COLLECT MESSAGES 147
8.3.4 EXAMINE THE MESSAGE 147
8.3.5 REPLY 148
8.3.6 EXAMPLE 148
8.4 IDCMP FLAGS 150
8.5 FUNCTIONS 154
8.6 EXAMPLES 155
9 MISCELLANEOUS 157
9.1 INTRODUCTION 157
9.2 MEMORY 157
9.2.1 ALLOCATE MEMORY 157
9.2.2 DEALLOCATE MEMORY 159
9.2.3 REMEMBER MEMORY 159
9.3 PREFERENCES 162
9.4 WARNINGS 165
9.5 DOUBLE CLICK 165
9.6 TIME 167
9.7 STYLE 167
9.7.1 GADGETS 167
9.7.2 REQUESTERS 167
9.7.3 MENUS 168
9.7.4 MOUSE 169
9.8 FUNCTIONS 169
9.9 EXAMPLES 175
10 SPRITES 177
10.1 INTRODUCTION 177
10.2 LIMITATIONS 177
10.3 COLOURS 178
10.4 ACCESS HARDWARE SPRITES 178
10.4.1 SPRITE DATA 179
10.4.2 SIMPLESPRITE STRUCTURE 181
10.4.3 RESERVE A SPRITE 182
10.4.4 PLAY WITH THE SPRITE 182
10.4.5 FREE THE SPRITE 183
10.4.6 PROGRAM STRUCTURE 183
10.5 TECHNIQUES 185
10.5.1 WIDER SPRITES 185

10.5.2 MORE COLOURS 186
10.5.2.1 15 COLOURED SPRITE DATA 186
10.5.2.2 ATTACH SPRITES 189
10.5.2.3 MOVE ATTACHED SPRITES 189
10.5.3 LEVELS 190
10.6 FUNCTIONS 191
10.7 EXAMPLES 192
11 AMIGADOS 194
11.1 INTRODUCTION 194
11.1.1 PHYSICAL DEVICES 194
11.1.2 VOLUMES 195
11.1.3 DIRECTORIES/SUBDIRECTORIES/FILES 196
11.1.4 LOGICAL DEVICES 197
11.2 OPEN AND CLOSE FILES 198
11.3 READ AND WRITE FILES 199
11.3.1 READ() 200
11.3.2 WRITE() 200
11.4 MOVE INSIDE FILES 201
11.5 FILES AND MULTITASKING 201
11.6 OTHER USEFUL FUNCTIONS 203
11.6.1 CREATE DIRECTORIES 203
11.6.2 DELETE FILES AND DIRECTORIES 204
11.6.4 ATTACH COMMENTS TO FILES AND DIRECTORIES 205
11.6.5 PROTECT FILES AND DIRECTORIES 205
11.7 EXAMINE FILES AND DIRECTORIES 206
11.7.1 FILEINFOBLOCK AND DATESTAMP STRUCTURE 206
11.7.2 EXAMINE() 208
11.7.3 4 BYTE BOUNDARY 208
11.7.4 EXAMPLE 209
11.7.5 EXAMINE FILES/SUBDIRECTORIES IN A DIRECTORY/DEVICE 210
11.7.5.1 EXNEXT() 210
11.7.5.2 ERROR MESSAGES 211
11.7.5.3 EXAMPLE 212
11.8 FUNCTIONS 213
11.9 EXAMPLES 219
12 LOW LEVEL GRAPHICS ROUTINES 221
12.1 INTRODUCTION 221
12.2 CREATE A DISPLAY 221
12.2.1 GENERAL INFORMATION 221
12.2.1.1 HOW A MONITOR/TV WORK 221
12.2.1.2 INTERLACED 222
12.2.1.3 HIGH AND LOW RESOLUTION 223
12.2.1.4 PIXELS 223
12.2.1.5 COLOURS 224
12.2.2 DISPLAY ELEMENTS 224
12.2.2.1 RASTER 224
12.2.2.2 VIEW 225

12.2.2.3 VIEWPORT 226
12.2.2.4 BITMAP 227
12.2.3 CREATE A DISPLAY 228
12.2.3.1 VIEW 228
12.2.3.1.1 VIEW STRUCTURE 229
12.2.3.1.2 PREPARE A VIEW STRUCTURE 229
12.2.3.2 VIEWPORTS 230
12.2.3.2.1 VIEWPORT STRUCTURE 230
12.2.3.2.2 PREPARE A VIEWPORT STRUCTURE 232
12.2.3.3 COLORMAP 232
12.2.3.3.1 COLORMAP STRUCTURE 232
12.2.3.3.2 DECLARE AND INITIALIZE A COLORMAP STRUCTURE 233
12.2.3.3.3 SET THE RGB VALUES 233
12.2.3.3.4 DEALLOCATE THE COLOURMAP 234
12.2.3.4 BITMAP 234
12.2.3.4.1 BITMAP STRUCTURE 234
12.2.3.4.2 DECLARE AND INITIALIZE A BITMAP STRUCTURE 235
12.2.3.4.3 ALLOCATE RASTER 236
12.2.3.5 RASINFO 237
12.2.3.5.1 RASINFO STRUCTURE 237
12.2.3.5.2 DECLARE AND INITIALIZE A RASINFO STRUCTURE 237
12.2.3.6 MAKEVPORT() 238
12.2.3.7 MRGCOP() 238
12.2.3.8 LOADVIEW() 238
12.2.4 CLOSE A DISPLAY 239
12.2.5 EXAMPLE 240
12.3 DRAW 243
12.3.1 RASTPORT 244
12.3.1.1 RASTPORT STRUCTURE 244
12.3.1.2 PREPARE A RASTPORT 247
12.3.2 DRAWING PENS 248
12.3.3 DRAWING MODES 249
12.3.4 PATTERNS 249
12.3.4.1 LINE PATTERNS 250
12.3.4.2 AREA PATTERNS 250
12.3.4.3 MULTICOLOURED PATTERNS 251
12.3.5 BITPLANE MASK 252
12.3.6 DRAW SINGLE PIXELS 253
12.3.7 READ SINGLE PIXELS 253
12.3.8 POSITION THE CURSOR 253
12.3.9 TEXT 254
12.3.10 DRAW SINGLE LINES 254
12.3.11 DRAW MULTIPLE LINES 255
12.3.12 DRAW FILLED RECTANGLES 255
12.3.13 FLOOD FILL 256
12.3.14 DRAW FILLED AREAS 257
12.3.14.1 AREAINFO AND TMPRAS STRUCTURES 257

12.3.14.2 AREAMOVE(), AREADRAW() AND AREAEND() 258
12.3.14.3 TURN OFF THE OUTLINE FUNCTION 259
12.3.14.4 EXAMPLE 259
12.3.15 SET THE RASTER TO A SPECIFIC COLOUR 260
12.3.16 BLITTER 260
12.3.16.1 CLEAR RECTANGULAR MEMORY AREAS 260
12.3.16.2 SCROLL A RECTANGULAR AREA 261
12.3.16.3 COPY RECTANGULAR AREAS 261
12.4 FUNCTIONS 264
12.5 EXAMPLES 275
13 VSPRITES 277
13.1 INTRODUCTION 277
13.2 HOW VSPRITES WORK 277
13.2.1 LIMITATIONS 278
13.2.2 HOW TO AVOID THE LIMITATIONS 278
13.3 CREATE VSPRITES 279
13.3.1 VSPRITE DATA 279
13.3.2 VSPRITE STRUCTURE 279
13.3.3 COLOUR TABLE 281
13.3.4 GELSINFO STRUCTURE 282
13.3.5 INITIALIZE THE GELSINFO STRUCTURE 283
13.3.6 INITIALIZE THE VSPRITE STRUCTURE 284
13.3.7 ADD THE VSPRITE TO THE VSPRITE LIST 284
13.3.8 PREPARE THE GEL SYSTEM 284
13.3.9 CHANGE THE VSPRITE 286
13.3.10 REMOVE VSPRITES 286
13.4 A COMPLETE EXAMPLE 287
13.5 FUNCTIONS 293
13.6 EXAMPLES 295
14 HINTS AND TIPS 296
14.1 INTRODUCTION 296
14.2 NTSC VERSUS PAL 296
14.2.1 HOW TO WRITE PROGRAMS THAT WILL FIT BOTH SYSTEMS 296
14.2.2 NTSC OR PAL? 296
14.3 PROGRAMS RUNNING UNDER WORKBENCH 297
14.4 CHECK IF THE PROGRAM WAS STARTED FROM CLI OR WORKBENCH 299
14.5 EXAMPLES 300

ACM - Amiga C Manual
Anders Bjerin

Part I: Introduction, Screens, Windows, Graphics, Gadgets

http://aminet.net/package/dev/c/ACM

The complete boiled-down C manual for the Amiga which describes how to open and work with Screens, Windows, Graphics, Gadgets, Requesters, Alerts, Menus, IDCMP, Sprites, VSprites, AmigaDOS, Low Level Graphics Routines, Hints and Tips, etc. The manual also explains how to use your C Compiler and gives you important information about how the Amiga works and how your programs should be designed. The manual consists of 15 chapters together with more than 100 fully executable examples with source code.

0 INTRODUCTION

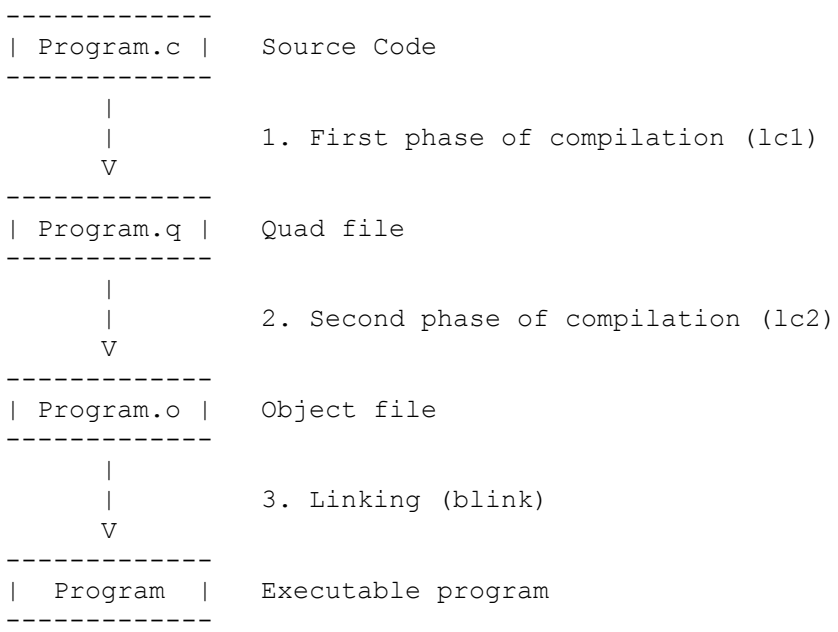
0.1 INTRODUCTION

In this chapter we will look at how C compilers work, how to use the Lattice C compiler, repeat some important things in C, and finish of by explaining how two use the Amigas's Libraries, and discuss the three different types of memory. As you have noticed, this chapter will give you a short introduction to C, compilers and the Amiga.

0.2 HOW TO COMPILE AND LINK

The idea with C compilers is that you write a document (usually referred as "source code"), which is read by the C compiler in two stages. The first time the C compiler will create a working file, referred as "quad file". The quad file is then compiled a second time into an "object file".

The object file contains mostly executable machine code, but it also contains references to external functions/variables in other object files and libraries. So before we can run the program we need to tie all object files and libraries together into one single program. It is done by the linker.



0.2.1 SOURCE CODE

When you are going to write C programs you need an editor/ wordprocessor to create the "source code". The source code is like a normal document except that it is written following some special rules. Almost any kind of editor/wordprocessor can be used. You can even use the MEMACS which you got when you bought your Amiga. (MEMACS exist on "The Extras Disk".)

Here is an example of a source code:

```
#include <stdio.h>

main()
{
  printf("Hello!\n");
}
```

Once you have written the source code you need to save it. It is standard that the file names should end with the extension ".c". So if you want to call your program "hello", the source code should be named "hello.c".

0.2.2 FIRST PHASE OF COMPILATION

Once the source code is saved you can start the first phase of compilation. If you use the Lattice C Compiler you simply use the command "lc1". To compile our example you only need to write "lc1 hello.c", and the compiler will then create a quad file named "hello.q".

0.2.3 SECOND PHASE OF COMPILATION

To start the second phase of compilation you use the command "lc2". To compile our example you then only need to write "lc2 hello.q", and the compiler will then create an object file named "hello.o".

If you want to invoke both the first and second compiler pass in one single call you can use the command "lc". So if you write "lc hello.c" both the first and the second phase of compilation would be executed. (lc calls both lc1 and lc2 automatically.)

0.2.4 LINKING

The quad file does not, as said before, contain only machine code instructions. It also contains calls to external functions in other object files and libraries. Before you may run the program you therefore need to tie everything together by a linker. Lattice C uses a linker called "blink".

Every program you create must normally be linked together with the startup routine "c.o", and the two libraries: "lc.lib" and "amiga.lib".

The object file and the two libraries are placed in the "lib" directory on the second Lattice C disk. That directory is normally already assigned the name "LIB:", so if you want to find the files, you simply write "LIB:c.o" for example.

For almost all programs you call the linker like this:
(This shows how to link our example "hello".)

```
blink LIB:c.o, hello.o  TO hello  LIB LIB:lc.lib, LIB:amiga.lib
        [A]      [B]       [C]    [D]     [E]          [F]
```

[A] The object file "c.o" must always be placed before all other object files. The file can be found in the logical device "LIB:".

[B] After the startup file "c.o" you may place all other object files you want to link together. (Remember to put a comma between them.) In this example we only have one object file, "hello.o".

[C] After the list of object files you write the word "TO" plus a name which will be the name of the executable program. In this example we want our program to be called "hello".

[D] Almost all programs need to be linked together with some libraries. The keyword "LIB" tells blink that the coming file-names are names on libraries. (Remember to put a comma between each library name.)

Do not mix up the keyword "LIB" with the logical device "LIB:". "LIB" tells the linker that a list of library-names will come, while the logical device "LIB:" tells AmigaDOS which device and directory to find some files in.

[E] The library "lc.lib" must be the second last library in the list. The library can be found in the logical device "LIB:".)

[F] The library "amiga.lib" must be the last library in the list. The library can be found in the logical device "LIB:".)

If you want to invoke both the first and second compiler pass, plus link the object file together with the standard libraries, in one single call you can use the command "lc" with the added parameter "-L". So if you want to compile and link the program in one single command you only need to write:

```
lc -L hello.c
```

The command "lc" will invoke both the first and the second compiler pass, and the added parameter "-L" means that the finished object code should be linked together with the standard libraries.

If you also want the linker to search the standard math library you simply put the letter "m" after the "-L":

```
lc -Lm <file-name.c>
```

Our simple example "hello" does not need the math library, but if you use any sort of calculations you normally need to include it. ALL examples in this manual will compile and link happily with the command:

```
lc -Lm ExampleX.c
```

0.3 C PROGRAMS

This manual is written for everyone who has a little knowledge of C and who wants to program the Amiga. If you know how to write a program that prints out "hello!" on the screen, and are not totally confused by pointers and structures, I do not think you will have any problems reading this manual. All examples have been written as clearly as possible, and I have avoided many shortcuts in order to make the examples easily understandable.

However, before we start with the real manual I think it is best to repeat some important features in C:

0.3.1 #INCLUDE

The first thing you find in a C program is the "#include" commands. They tell the compiler to read some files before compiling the code, and can therefore contain definitions of constants/structures/macros etc. The files are normally referred as "header files", and have because of that a ".h" extension.

There exist two types of header files; your own and standard system definitions. If you include your own header files you put a double-quotation around the file name:

```
#include "mydefinitions.h"
```

The compiler will then look in the "current directory" for your file. If you on the other hand include any standard system definitions you put two angle-brackets around the file name: (The compiler will then look in the logical device "INCLUDE:")

```
#include <intuition/intuition.h>
```

Note that there is not semicolons after the #include statements! That is because the #include files are scanned by a macro pre-processor (included in the first compiler phase, lc1), and have actually nothing to do with the C compiler.

Later in the manual we will often include the file intuition.h which contains definitions of many structures (such as Window, Border, Gadget etc) plus some important constants and macros. This and all other standard system header files are compressed and put on the second Lattice C disk. If you are interested to see what they actually define you can find the uncompressed files on the fourth Lattice C disk.

0.3.2 #DEFINE

After the #include statements the #define statements will come. With help of the #define command you can both declare constants and replacement strings, but you can even declare your own macros.

The important thing to remember is that the first defined string is replaced with the second string by the macro pre-processor: (The first space will separate the first string from the second string. Note the underline character between MAX and WIDTH, and the quotations between Anders Bjerin.)

```
#define MAX_WIDTH 150
#define AUTHOR    "Anders Bjerin"
```

The macro pre-processor will replace all MAX_WITH strings with 150, and all AUTHOR strings with "Anders Bjerin".

If there are any parentheses in the first string then you are declaring macros:

```
#define SQUARE(x) x*x
```

The first string is replaced with the second one as normal, but every element (separated by commas if more than one) inside the parentheses also moved into the formula. A line like this:

```
nr = SQUARE(4);

Will be replaced like this:

nr = 4*4;
```

It is however important to remember that the pre-processor consider everything to be strings, and a line like this:

```
nr = SQUARE(4+1);

Will be replaced like this:

nr = 4+1*4+1
```

Square of 4+1 is 25 (5*5), but nr is equal to 9. This shows that you have to be very careful when you define your macros. To solve this problem you only need to add some parentheses around the x:s:

```
#define SQUARE(x) (x)*(x)

nr = SQUARE(4+1);

Will then be replaced like this:

nr = (4+1)*(4+1);
```

However, the macro is still not perfectly defined. Consider this statement:

```
nr = 16 / SQUARE(2);

Will then be replaced like this:

nr = 16 / (2)*(2);
```

16 divided by square of 2 is 16/4, which is equal to 4, but nr is equal to 16 (16/2 * 2). The solution to all this problems is to put parentheses around every element, PLUS parentheses around the whole expression:

```
#define SQUARE(x) ((x)*(x))

nr = 16 / SQUARE(2);

Will then be replaced like this:

nr = 16 / ((2)*(2)); /* OK */
```

What you need to remember about macros is that:
1. Put parentheses around every element.
2. Put also parentheses around the whole expression.

Here is a list of some commonly used macros:

```
#define MAX(x,y)        ((x)>(y)?(x):(y))
#define ABS(x)          ((x)<0?-(x):(x))
#define ANSWER(c)       ((c)=='y'||(c)=='Y'?1:0)
```

Remember, if you have written some good constants/macros you can put them all in a header file and #include it every time you need them.

0.3.3 OTHER PRE-PROCESSOR COMMANDS

#include and #define are the two most commonly used pre-processor commands, but there exist five more which can be used together with the #define command:

1. You can check if something has been defined, and then define something else. (#ifdef)

2. You can check if something has not been defined, and then define something else. (#ifndef)

3. You can put an "else" statement which is connected to an #ifdef/ifndef. (#else)

4. You can put an end mark which is connected to an #ifdef/ #ifndef. (#endif)

5. You can undefine the something that has already been defined. (#undef)

Here some examples which shows how you can use the pre-processor commands:

```
#define BIG 1

#ifdef BIG
  #define HEIGHT 100  /* If BIG is defined, HEIGHT and WIDTH */
  #define WIDTH  300  /* is defined to 100 and 300.          */
#else
  #define HEIGHT  50  /* If BIG is undefined, HEIGHT and     */
  #define WIDTH   25  /* WIDTH is defined to 50 and 25.      */
#endif
```

These commands are very handy when you #include several files, and you want to check if something has been defined. If it is not defined you can include some other files and so on...

```
#ifndef MY_DEFINITIONS      /* If MY_DEFINITIONS is undefined */
  #include "mydefinitions"  /* we include "mydefinitions".    */
#endif
```

The last pre-processor command, #undef, undefines the last definition:

```
#define HEIGHT 200  /* HEIGHT is defined as 200.            */
#define HEIGHT 350  /* HEIGHT is redefined as 350.          */
#undef HEIGHT       /* HEIGHT last definition is undefined, */
                    /* and is redefined as previous to 200. */
#undef HEIGHT       /* HEIGHT is now undefined.             */
```

0.3.4 FUNCTIONS

C compilers expect in general to find function definitions (what type of data the function will return) before they are referenced. Since the main() function will come first, and will probably contain references to other functions further down, we need to declare the functions before the main() function. When a function is defined, before it is declared, we need to put a semicolon after it. The compiler will then understand that we only define a function, and that the function will be declared later on:

```
#include <stdio.h>

void main();              /* Will not return anything.     */
int get_radius();         /* Will return an integer value. */
float calculate_area();   /* Will return a float value.    */
void print_area();        /* Will not return anything.     */

void main()
{
  int radius;
  float area;

  radius = get_radius();
  area = calculate_area( radius );
  print_area( area );
}

int get_radius()
{
  int r;

  printf( "Please enter the radius: " );
  scanf( "%d", &r );

  return( r );
}
```

```
float calculate_area( r )
int r;
{
  float a;

  a = 3.14 * r * r;

  return( a );
}

void print_area( a )
float a;
{
  printf( "The area is %f square units.\n", a );
}
```

0.3.5 VARIABLES

Here are the standard Lattice C data types:

TYPE	BITS	Minimum value	Maximum value
char	8	-128	127
unsigned char	8	0	255
short	16	-32 768	32 767
unsigned short	16	0	65 535
int	32	-2 147 483 648	2 147 483 647
unsigned int	32	0	4 294 987 295
long	32	-2 147 483 648	2 147 483 647
unsigned long	32	0	4 294 987 295
float	32	±10E-37	±10E+38
double	64	±10E-307	±10E+308

Important, these data types can be different on other versions of C, so be careful when using them. To be sure that the size of the variables are the same on different systems there exist a special list of variable definitions. You only need to include the file "exec/types.h", and you have a list of safe variable types. The file defines, among many things, these commonly used data types:

Amiga Data Types	Lattice C Data Types	Description
LONG	long	Signed 32-bit
ULONG	unsigned long	Unsigned 32-bit
LONGBITS	unsigned long	32 bits manipulation
WORD	short	Signed 16-bit
UWORD	unsigned short	Unsigned 16-bit
WORDBITS	unsigned short	16 bits manipulation

BYTE	char	Signed 8-bit
UBYTE	unsigned char	Unsigned 8-bit
BYTEBITS	unsigned char	8 bits manipulation
VOID	void	Nothing
STRPTR	*unsigned char	String pointer
CPTR	ULONG	Absolute memory pointer
TEXT	unsigned char	Text
BOOL	short	Boolean (The file has also defined the two words TRUE = 1 and FALSE = 0)

Here is a list of some data types which should not be used any more:

APTR	*STRPTR	Absolute memory pointer (Misdefined, use CPTR!)
SHORT	short	Signed 16-bit (WORD)
USHORT	unsigned short	Unsigned 16-bit (UWORD)

See file "exec/types.h" on the fourth Lattice C disk for more information. This file is automatically included when you include the "intuition.h" header file.

0.3.6 STORAGE CLASSES FOR VARIABLES

There exist six main types of storage classes for variables:

1. Automatic
2. Formal
3. Global
4. External Static
5. Internal Static
6. Register

0.3.6.1 AUTOMATIC

Automatic variables are declared inside functions with the keyword "auto" in front of them. (Variables which are declared inside a function with no keyword in front of them are assumed to also be automatic variables.) Space for them are reserved from the stack, and they can only be reached inside the same function that declared them. When the function has finished the memory is automatically deallocated back to the stack.

Here is a small recursion program. The user can enter values until he enters 0 when all the numbers will be printed out backwards.

```
#include <stdio.h>

main()
{
  get_number();
}

get_number()
{
  auto int number; /* <== Here it is!             */
                   /* Keyword "auto" is optional. */

  printf( "Enter a number (0: Quit) =>" );
  scanf( "%d", &number );
  if( number )
    get_number();
  else
    printf( "%d\n", number );
}
```

0.3.6.2 FORMAL

Very similar to automatic variables. Memory is reserved from the stack, they can only be reached inside the function which declared them, the memory is automatically deallocated back to the stack when the function quits. The only difference is that they are the functions "parameters" and are declared after the parentheses, but before the curly brackets.

```
#include <stdio.h>

void main();
void print_number();

void main()
{
  print_number( 3.142 );
}

void print_number( number )
float number;                /* <== Here it is! */
{
  printf( "The number is %f!\n", number );
)
```

0.3.6.3 GLOBAL

Global variables are declared outside the functions and can therefore be reached from everywhere. The memory is reserved in the BSS or DATA hunks, and can not be deallocated. (All uninitialized data is put in the BSS hunks, while all initialized data is put in the DATA hunks.)

```
int number;      /* Put in the BSS hunks.  */
int width = 12;  /* Put in the DATA hunks. */
```

Information about global variables is put in the object file so other modules can reach the variables. If a variable is declared as a global variable (declared outside the functions) in module A, the variable should be declared as an external global variable (declared outside the functions with the keyword "extern" in front of it) in module B.

```
-----------------------------------------------------------
| Module A                                                |
-----------------------------------------------------------
#include <stdio.h>

void main();

char name = "Andirs Bjerin"; /* Declare this variable as a  */
                             /* global variable. (DATA)    */
void main()
{
  print_name();
}


-----------------------------------------------------------
| Module B                                                |
-----------------------------------------------------------
#include <stdio.h>

void print_name();

extern char name;   /* Tell the compiler that this variable */
                    /* actually exist, but is declared in   */
                    /* another module.                      */
void print_name()
{
  name[ 3 ] = 'e';
  printf("Programmer: %s\n", name );
}
```

To compile these two modules and link them you only need to:

1. Compile (lc1 and lc2) Module A: lc ModuleA.c
2. Compile (lc1 and lc2) Module B: lc ModuleB.c
3. Link the two object modules together with some other libraries etc:
   blink LIB:c.o, ModuleA.o, ModuleB.o TO program
         LIB LIB:lc.lib, LIB:amiga.lib

0.3.6.4 EXTERNAL STATIC

Same as global variables but can ONLY be reached inside the same module. All functions in that module will know about the variable, but no functions in other modules can reach it. The memory is taken from the BSS or DATA hunks, but no information about these variables is put in the object files.

To declare an external static variable you simply put the keyword "static" in front of a global variable.

```
static numbers[ 20 ];
```

0.3.6.5 INTERNAL STATIC

Same as external static variables except that they are declared inside the functions, and can only be reached by that function. The special thing about internal static variables is that the memory is not deallocated when the the function has finished, and the values which were stored in them is untouched. (The memory was reserved from the BSS or DATA hunks.) An internal static variable will only be initialized once, but can later be changed, but not deallocated.

```
#include <stdio.h>

void main();
void count_up();

void main()
{
  int loop;

  for( loop = 0; loop < 20; loop++ )
    count_up();
}

void count_up()
{
  static number = 1; /* We declare and initialize an internal */
                     /* static variable. Since the number is  */
                     /* a static variable it will only be     */
                     /* initialized the first time we call    */
                     /* the function. (DATA)                  */

  number++;

  printf("Number: %d\n", number );
}
```

0.3.6.6 REGISTER

Register variables are like automatic variables except that you ask the compiler if the variable can be allowed to use a processor register for it self. This is very handy if you use a variable a lot, for example in a loop. However, it is not sure that you can reserve a register. If you are lucky the compiler managed to reserve a register for the variable, but it is not always possible. The register variable will then be a normal automatic variable.

```
#include <stdio.h>

void main();

void main()
{
  register int loop; /* <== Here it is! */

  for( loop = 0; loop < 10000; loop++)
    ;
}
```

0.3.7 POINTERS

Pointers are declared in the same way as normal variables with only one important difference. We put a "*" sign in front of the name:

```
int a;  /* "a" is an integer variable. */
int *b  /* "b" is an integer pointer variable. */
```

If we want to get the address of a variable we put a "&" sign in front of the variable. "a" contains an integer value while "&a" is the memory address of that variable. If we want the pointer "b" to point to the variable "a", we simply write:

```
b = &a; /* "b" will contain the address of variable "a". */
```

It is important to remember that "b" is a pointer variable ("b" can point to any integer variable), while "&a" is a pointer constant ("&a" will always be the address of "a").

0.3.8 STRUCTURES

I will not try to explain everything about structures, but I will however give a short summary of how to use them.

0.3.8.1 HOW TO DECLARE STRUCTURES

You declare a structure type like this:
1. Write the keyword "struct" plus a name for the structure type.
2. Put all variable-types with their names inside two curly brackets.

If you want to declare a structure type called "person" which should consist of an integer field, named "age", and two float fields, named "height" and "weight", you write:

```
struct person
{
  int age;
  float height, weight;
}
```

You can put often used structures declarations in a header file and then include it when necessary. When we are going to program the Amiga we will use a lot of declared structures which mainly exist in the header file "intuition.h".

0.3.8.2 HOW TO CHANGE THE STRUCTURE'S FIELDS

Once you have declared a structure type, you can declare structure variables and pointers. To declare a structure variable named "anders" and a structure pointer named "programmer" you write:

```
struct person anders;

struct person *programmer;
```

You can now initialize the structure variable "anders" with some data. You do it by writing the name of the structure variable, put a dot, and then write the field you want to access. So if you want to change the age in the structure variable "anders" to 20, the height to 1.92, and the weight to 74.5 you write:

```
anders.age = 20;
anders.height = 1.92;
anders.weight = 74.5;
```

0.3.8.3 POINTERS AND STRUCTURES

If we want to initialize the pointer "programmer" so it points to the "anders" structure, we write:

```
programmer = &anders;
```

Remember: "anders" is the structure variable.
"&anders" is the address of the structure variable.

Once the structure pointer points to a structure we can start to check and change values of that structure. The important thing to remember is that we do not put a dot between the pointer name and the field name, we put a "->" sign. (The pointer "programmer" points to the field "height": programmer -> height etc.)

```
if( programmer -> age == 20 )
  /* The programmer is 20 years old. */

programmer -> weight -= 5; /* The programmer is on a diet. */
```

0.3.9 CASTING

If you want to convert one type of data into another type of data you need to use a method which is usually referred as "casting". If you want to convert a WORD variable into a LONG variable type you would write something like this:

```
WORD number;
LONG data;

data = (LONG) number;
```

The command (LONG) tells the C compiler that the next data type should be converted into a LONG variable type.

When you convert different variable types you normally do not need to do any casting. In the previous you could therefore have left out the (LONG) command, but to make it clear for anyone who reads the code that it is no mistake, it is best to keep it.

However, when you convert different pointer types you should use the casting method. Since the most common errors in C have something to do with pointers, C compiler are usually very fussy about which pointers may point to which variables/structures etc.

If we have declared a pointer to be a normal memory CPTR pointer (programmer), and want it to point to a structure (person), the compiler would be very confused.

```
struct person anders;
CPTR programmer;

programmer = &anders;
```

A memory pointer, and a pointer to a structure, is actually the same thing, but the paranoid compiler does not believe it, and will give us a warning which look something like this:

```
Example.c 27 Warning 30: pointers do not point to same type of object
```

To tell the compiler that the two different types of memory addresses are actually the same thing you need to do some casting. We need to tell the compiler that a pointer to the structure anders (&anders) is actually a memory pointer (CPTR).

```
programmer = (CPTR) &anders;
```

0.4 LIBRARIES

Many special functions have already been written for you, and are placed in different "libraries". If you want to access any of the functions, you simply need to "open" the right library. You open a library by calling the function OpenLibrary(), and it will return a pointer to a library structure.

There exist roughly two different types of libraries, ROM libraries which are placed in ROM (surprise, surprise), and Disk libraries which are placed on the boot disk (system disk).

0.4.1 ROM LIBRARIES

ROM libraries are always available (they are placed in the ROM) but they still need to be opened before you may use their functions. Here is a list of the ROM Libraries:

GRAPHICS — This library contains all functions which has anything to do with graphics: Display (View, ViewPorts and RastPorts), sprites (Hardware/ software sprites and BOBs), text and fonts etc. Chapter 10 and some examples in the chapters 1 - 9 uses this library.

LAYERS — This library contains routines that handles the screen in such a way that different "layers" (planes) can be used and overlap each other.

INTUITION — This library takes care of the "user interface" and works with the Graphics and Layers libraries. Takes care of screens/windows/gadgets/requesters/menus etc. Chapter 1 - 9 explains how to use this library.

EXEC	Takes care of the memory/tasks/libraries/devices/ resources, i/o ports etc.
DOS	AmigaDOS. Contains routines which open/close/read/ write files etc.
MATHFFP	Motorola Fast Floating Point Routines. Handles addition, subtraction, division, multiplication, comparison, conversion etc. (Single Precision)
RAM-LIB	Takes care of the RAM disk.
EXPANSION	Handles the expansion slots.

0.4.2 DISK LIBRARIES

If a programs opens a disk library, AmigaDOS will look in the logical device "LIBS:", which has been assigned to system disk's "libs" directory, and will then load the whole library into RAM. If the library has already been loaded into RAM by another task, Exec will then use the already existing library. (This shows how memory efficient libraries are, since several program can use the same code.)

Here is a list of the Disk Libraries:

TRANSLATOR	This library translates English strings into phonetic strings.
DISKFONT	Handles the Fonts on the disk.
ICON	Takes care of the "icons" used by the Workbench.
MATHTRANS	Contains functions like sin, cos, log, ln, square, power etc.
MATHIEEEDOUBBAS	Same as the ROM Library "MATNFFP" but uses Double Precision.

0.4.3 OPEN AND CLOSE LIBRARIES

You open, as said above, a library by calling the function OpenLibrary(). It will return a pointer to a library structure, and the pointer MUST be stored in a pointer variable with a special name. For example, if you open the Intuition library, the pointer should be called "IntuitionBase", and if you open the Graphics library, the pointer should be called "GfxBase".)

```
Synopsis: pointer = OpenLibrary( name, version );

pointer:  (struct Library *) Pointer to a library structure.
          If you open the Intuition library it is a pointer
          to an IntuitionBase structure, and should be called
          "IntuitionBase". If you open the Graphics library
          it is a pointer to a GfxBase structure, and should
          be called "GfxBase". (You will need to do some
          casting here.)

name:     (char *) Pointer to a string containing the library
          name. The Intuition library is called
          "intuition.library", and the Graphics library is
          called "graphics.library".

version:  (long) You can here specify that the user need a
          special version (or higher) of the library to run
          the program. If you want any version of the
          library, simply write 0. (The highest library
          version is for the moment 34.)
```

Every library which has been opened, must be closed before the program terminates. You close a library by calling the function CloseLibrary(), with the library pointer as the only argument. The ROM libraries do not actually need to be closed since they exist in ROM, and will not occupy any valuable memory, but close it anyway. (Always follow the rule: Close everything that you have opened!)

```
Synopsis: CloseLibrary( pointer );

pointer:  (struct Library *) Pointer to a library structure
          which has been previously initialized by an
          OpenLibrary() call.
```

In this manual we will only discuss functions in the Intuition and Graphics libraries. Here is a fragment of a program that opens and closes the Intuition library:

```
struct IntuitionBase *IntuitionBase;

main()
{
  /* Open the Intuition Library (any version): */
  IntuitionBase = (struct IntuitionBase *)
    OpenLibrary( "intuition.library", 0 );

  if( IntuitionBase == NULL )
    exit(); /* Could NOT open the Intuition Library! */

  ... ...

  /* Close the Intuition Library: */
  CloseLibrary( IntuitionBase );
}
```

Here is another fragment of a program which this time opens and closes the Graphics library:

```
struct GfxBase *GfxBase;

main()
{
  /* Open the Graphics Library (any version): */
  GfxBase = (struct GfxBase *)
    OpenLibrary( "graphics.library", 0 );

  if( GfxBase == NULL )
    exit(); /* Could NOT open the Graphics Library! */


  ... ...


  /* Close the Graphics Library: */
  CloseLibrary( GfxBase );
}
```

0.5 MEMORY

The Amiga can be expanded to a total of 9MB of memory. This memory can be divided into three different types:

Chip Memory — The first 512KB is normally called "Chip Memory", and can be accessed both by the processor as well as the co-processors and the other special chips in the Amiga. (All graphics/Sound etc which is used by the special Chips (Blitter etc) must be located here!)

Slow Memory — The following 512KB (extra memory which is put inside the Amiga) can only be accessed by the main CPU, and can therefore not be used to store graphics and sound data. It is called "Slow" since it is not accessible by the Chips, but can be interrupted by them. If you have a screen with a high resolution and/or many colours this memory will be slowed down since the Chips will steal some memory cycles from it.

There exist a new Chip set which enables the co-processors and Chips to use this memory also, which means that this memory could be used to store graphics/sound data etc. However, most of the Amigas (99.9%) have still the old Chips, and this memory can NOT be used for graphics/sound data.

Fast Memory: The third type of memory is the following 8MB, which can ONLY be accessed by the main processor, and is NOT slowed down by the Chips. This memory is therefore called "Fast Memory".

The important thing to remember is that all Graphics and Sound data (all data handled by the Chips) MUST be stored in the "Chip Memory"! There exist three different ways of putting the data into the Chip Memory:

1. If you use the Lattice C Compiler V5.00 or higher, you can put the keyword "chip" in front of the name of the data, and the Compiler will automatically put it in the Chip Memory.
   Eg. UWORD chip graphics_data[] = { ... };

2. If you use the Lattice C Compiler V4.00 you can compile the source code with the added command "-acdb" which will load all initialized data (DATA) into Chip Memory when the program is loaded.
   Eg. lc -acdb -Lm Example.c

3. Allocate some Chip Memory by calling the function AllocMem(), and then move all data into the newly allocated memory.

There exist other methods, such as running the program "ATOM" etc, but I will not explain it here. (See your own manuals for more information.)

1 SCREENS

1.1 INTRODUCTION

The screen is the foundation of Intuition's display. It determines how many colours you can use, what kind of colours, what resolution etc. Every window, gadget, icon, drawing is connected to a screen. Moving that screen, and you are also moving the objects on it. You may have several screens running at the same time, and you may combine different screens (with their own individual resolutions, colours, etc) on the same display.

1.2 DIFFERENT TYPES OF SCREENS

When you are going to use a screen you first need to decide if you want to use a Standard (Workbench) Screen or if you want to use a Screen which you yourself has customized (Custom Screens).

Workbench Screen:
  This is Intuition's standard screen. It is a four-colour, high-resolution screen.

Custom Screens:
  When you want a screen with your own display mode you should use a Custom Screen. You can then decide yourself how many colours you want, what colours, resolution, size etc.

If your program should use a high-resolution screen, and four colours is enough, you are advised to use the Workbench Screen. This will save a lot of memory (no memory allocated for a Custom Screen), and allows the user to have several programs running on the same display. On the other hand, if you want more colours or want to use a different resolution/display-mode you should use a Custom Screen.

1.3 WORKBENCH SCREEN

Workbench Screen is a high-resolution (640 pixels wide), four colour screen. It is either 200 or 256 lines high depending on if you have a NTSC (American) or a PAL (European) computer. (400/512 lines if you have an Interlaced display.)

The Workbench Screen will automatically open if there does not exist any other screens. If you do not want the Workbench Screen, and need a lot of memory, you can try to close it, CloseWorkBench(). Remember to reopen it when your program has finished, OpenWorkBench().

You are advised to always call the function OpenWorkBench() when your program exits, even if you have not closed it yourself. Another program may have closed it because of shortage of memory, and when your program is finished there may exist enough memory. This will make the Workbench Screen available as much as possible.

1.4 CUSTOM SCREENS

If you are going to use a Custom Screen you need to initialize a structure (NewScreen) with your requirements. You also need to open the screen yourself by calling the function OpenScreen().

Here is a list of some important decisions you need to make before you can open the screen:

1.4.1 RESOLUTION

You can either have a high-resolution or a low-resolution screen. A high resolution screen is 640 pixels wide, while a low-resolution screen is only 320 pixels wide. However, a high-resolution screen may only have a depth of up to 4 (maximum 16 colours), while a low-resolution screen may have a depth of up to 5 (32 colours), or even 6 if using some special display modes.

1.4.2 DEPTH

A screen's depth means how many bits are used for every pixel. The more bits the more combinations/colours:

Depth	Number of colours	Colour register number	
1	2	0 - 1	
2	4	0 - 3	
3	8	0 - 7	
4	16	0 - 15	
5	32	0 - 31	(Only low-res)

A low resolution screen may even have a depth of 6 if using some special display modes (HAM / EXTRA HALFBRIGHTE).

Each colour may be picked out of a 4096 colour palette. Once you have opened your Custom Screen you may change the colours by calling the graphics function SetRGB4().

1.4.3 INTERLACED

An Interlaced screen can be 400/512 lines while a Non-Interlaced screen can only be 200/256 lines. You are recommended to use a Non-Interlaced screen if possible, since an Interlaced display appear to "flicker" if the user does not have a special high-phosphor-persistence monitor. (99% of the users do not have it.)

The monitor normally draws the display 50 times per second. If you are using an Interlaced screen the Amiga will only draw every second line first, and will the next time draw the remaining lines. This is why an Interlaced display appear to "flicker". You can eliminate much of the disturbing effects if you are using colours with low contrasts, black and grey instead of black and white for example.

1.4.4 HAM AND EXTRA HALFBRIGHTE

If you have a low-resolution, six bitplanes (Depth 6) screen you can either use the special HAM mode, or the Extra Halfbrighte mode. HAM allows you to display all 4096 colours at the same time, but has some restrictions, and is complicated to use. Extra Halfbrighte gives you 32 extra colours, which are the same as the first 32 colours, but are a bit brighter.

Both HAM and Extra Halfbrighte is using a lot of memory/ processing time, and is therefore not commonly used.

1.4.5 DUAL PLAYFIELDS

Dual Playfields allows you to have two screens on top of each other. The top screen can then be transparent in some places (colour register 0) to show the bottom screen.

1.4.6 FONTS

You can specify a default font which will be used, if nothing else is specified, to draw the text on the screen. The system's default font is called Topaz and exist in two sizes:

TOPAZ_SIXTY : 9 lines tall, 64/32 characters per line.
TOPAZ_EIGHTY : 8 lines tall, 80/40 characters per line.

This font is built in the Amiga, and is therefore always available. (See chapter 3 GRAPHICS for more information.)

1.4.7 SIZE AND POSITION

The top of your screen does not need to be at the top of the display. You may open several screens and position them under each other. If you are, for example, designing an adventure game, you can have a low-resolution 32 colour screen at the top of the display (showing some nice pictures), and have a high-resolution 2 colour screen at the bottom of the display (showing the text).

(Remember that the user later can drag the screens up or down himself.)

The width of the screen should be either 320 (low-resolution) or 640 pixels (high-resolution).

The height may be anything between 1 - 200/256 lines on an Non-Interlaced screen, and between 1 - 400/512 lines on an Interlaced display.

1.4.8 TITLE

On top of each screen (on the "drag bar") there exist a screen title. There exist two kinds of titles:

- The default title, which is specified in the NewScreen structure.
- A "current" title which is the same as the title of the current active window. (See chapter 2 WINDOWS for more information.)

1.4.9 GADGETS

For the moment you are not allowed to attach any custom gadgets to a screen. (See chapter 4 GADGETS for more information about gadgets.)

1.5 INITIALIZE A CUSTOM SCREEN

Before you can open a custom screen you need to initialize a NewScreen structure which look like this:

```
struct NewScreen
{
  SHORT LeftEdge, TopEdge, Width, Height, Depth;
  UBYTE DetailPen, BlockPen;
  USHORT ViewModes;
  USHORT Type;
  struct TextAttr *Font;
  UBYTE *DefaultTitle;
  struct Gadget *Gadgets;
  struct BitMap *CustomBitMap;
};
```

LeftEdge: Initial x position of the screen. Should always be 0.

TopEdge: Initial y position of the screen.

Width: If it is a low-resolution screen it should be 320, otherwise 640.

Height: If it is a Non-Interlaced screen it can be anything between 1 - 200/256, otherwise 1 - 400/512.

Depth: 1-4 if high-resolution, otherwise 1-6.

DetailPen: The colour register used to draw the text with.

BlockPen: The colour register used for block fills etc.

ViewModes: You can use none or some of the flags. If you set more than one flag you put a "|" between them. (You are not allowed to set the HIRES flag together with the DUALPF or HAM flag, and you can not have DUALPF at the same time as HAM.)

- HIRES — Set this flag if you want a high-resolution screen. (The default is low-resolution.)
- INTERLACE — Set this flag if you want an Interlaced screen. (The default is Non-Interlaced)
- SPRITES — Set this flag if you are going to use sprites in your display.
- DUALPF — Set this flag if you want to use the Dual Playfields Mode.

HAM — Set this flag if you want to use the Hold And Modify Mode.

Type: Should be CUSTOMSCREEN. You can also set the CUSTOMBITMAP flag if you want the screen to use your own declared and initialized BitMap. (See chapter 2 WINDOWS for more information about Custom BitMaps.)

Font: A pointer to an already initialized TextAttr structure. The screen will use it as the default font. (See chapter 3 GRAPHICS for more information about fonts.) Set to NULL if you want to use the current default font.

DefaultTitle: A pointer to a text string which will be used as "default" title.

Gadgets: Not used for the moment. Set to NULL.

CustomBitMap: If you want to use your own BitMap you should give this field a pointer to the BitMap structure which you have declared, and initialized yourself. Remember to set the CUSTOMBITMAP flag in the Type variable. However, if you want Intuition to take care of the BitMap, set this field to NULL. (See chapter 2 WINDOWS for more information about Custom BitMaps.)

1.6 OPEN A CUSTOM SCREEN

Once you have declared and initialized the NewScreen structure, you can call the function OpenScreen() which will open your Custom Screen. When your screen has been opened you can, if you have allocated memory for the NewScreen structure, deallocate it since you do not need the structure any more.

This is how you call the OpenScreen() function:

```
my_screen = OpenScreen( &my_new_screen );
```

my_screen has been declared as:

```
  struct Screen *my_screen;
```

my_new_screen has been declared as:

```
  struct NewScreen my_new_screen;
```

and has been initialized with your requirements.

OpenScreen() will return a pointer to a Screen structure, else NULL if it could not open the screen (for example, not enough memory). Remember to check what OpenScreen() returned!

```
if( my_screen==NULL )
{
  /* PANIC! Could not open the screen */
}
```

1.7 SCREEN STRUCTURE

Once you have opened the screen you will receive a pointer to a Screen structure which look like this:

```
struct Screen
{
  struct Screen *NextScreen;    /* Pointer to the next */
                                /* screen, or NULL. */
  struct Window *FirstWindow;   /* Pointer to the first */
                                /* Window on this screen. */
  SHORT LeftEdge, TopEdge;      /* Position of the screen. */
  SHORT Width, Height;          /* Size of the screen. */
  SHORT MouseY, MouseX;         /* Mouse position relative */
                                /* to the top left corner */
                                /* of the screen. */
  USHORT Flags;                 /* The selected flags. */
  UBYTE *Title;                 /* The screen's Current title. */
  UBYTE *DefaultTitle;          /* The screen's Default title. */

  BYTE BarHeight, BarVBorder,
       BarHBorder, MenuVBorder,
       MenuHBorder;
  BYTE WBorTop, WBorLeft,
       WBorRight, WBorBottom;

  struct TextAttr *Font;        /* The screens default font. */
  struct ViewPort ViewPort;     /* The screen's ViewPort etc: */
  struct RastPort RastPort;
  struct BitMap BitMap;
  struct Layer_Info LayerInfo;
  struct Gadget *FirstGadget;
  UBYTE DetailPen, BlockPen;
  USHORT SaveColor0;
  struct Layer *BarLayer;
  UBYTE *ExtData;
  UBYTE *UserData;
};
```

You will probably not use this structure a lot, but some variables/structures can be useful later on. When you are, for example, using the function SetRGB4(). More about this later.

1.8 FUNCTIONS

Here are some common functions which will effect screens:

OpenScreen()

This function will open a Custom Screen with your requirements.

Synopsis: my_screen = OpenScreen(my_new_screen);

my_screen: (struct Screen *) Pointer to a Screen structure. It will point to your newly opened screen or be equal to NULL if the screen could not be opened.

my_new_screen: (struct NewScreen *) Pointer to a NewScreen structure which contains your preferences.

CloseScreen()

This function will close a Custom Screen which you have previously opened.

Synopsis: CloseScreen(my_screen);

my_screen: (struct Screen *) Pointer to an already opened screen.

All windows (See chapter 2 WINDOWS for more information) on your Screen MUST have been closed before you may close the screen. If you close a window after the screen has been closed, the system will crash. (Not recommended.)

If there does not exist any more screens when you close yours, Intuition will automatically reopen the Workbench Screen.

MoveScreen()

This function will move the screen. For the moment you may only move it vertically.

Synopsis: MoveScreen(my_screen, delta_x, delta_y);

my_screen: (struct Screen *) Pointer to the screen which you want to move.

delta_x: (long) Number of pixels which the screen should move horizontally. For the moment you may not move a screen horizontally, set it therefore to 0.

delta_y: (long) Number of lines which the screen should move vertically.

ScreenToBack()

This will move the screen behind all other screens.

Synopsis: ScreenToBack(my_screen);

my_screen: (struct Screen *) Pointer to the screen which you want to move.

ScreenToFront()

This will move the screen in front of all other screens.

Synopsis: ScreenToFront(my_screen);

my_screen: (struct Screen *) Pointer to the screen which you want to move.

ShowTitle()

This function will make the screen's Title appear above or behind any Backdrop Windows (See chapter 2 WINDOWS for more information about Backdrop Windows). (The screen's title appear always behind normal windows.)

Synopsis: ShowTitle(my_screen, show_it);

my_screen: (struct Screen *) Pointer to the screen.

show_it: (long) A boolean value which can be:
TRUE: The title will be in front of any Backdrop Windows, but behind any other windows.
FALSE: The Title will be behind any windows.

OpenWorkBench()

This function will try to open the Workbench Screen if there exist enough memory.

Synopsis: result = OpenWorkBench();

result: (long) A boolean value which tell us if the Workbench Screen has been (or already was) opened (TRUE), or not (FALSE).

CloseWorkBench()

This function will try to close the Workbench Screen if possible. If any other programs is using the Workbench Screen, the function can not close it. Closing the Workbench will free some memory, and can therefore be used if your program needs more memory.

(Remember to reopen the Workbench Screen when your program terminates.)

Synopsis: result = CloseWorkBench();

result: (long) A boolean value which tell us if the Workbench screen has been (or already was) closed (TRUE), or not (FALSE).

WBenchToBack()

This will move the Workbench Screen behind all other screens.

Synopsis: result = WBenchToBack();

result: (long) A boolean value which is TRUE if the Workbench screen was open, or FALSE it it was not.

WBenchToFront()

This will move the Workbench Screen in front of all other screens.

Synopsis: result = WBenchToFront();

result: (long) A boolean value which is TRUE if the Workbench screen was open, or FALSE it it was not.

SetRGB4()

This function allows you to change your screen's colours. Each colour may be picked out of a 4096 colour palette. (16 levels of red, 16 levels of green and 16 levels of blue; 16*16*16 = 4096.)

IMPORTANT! Before you may use this function you must have opened the Graphics Library. (All other functions are in the Intuition Library.) (See chapter 0 INTRODUCTION for more information.)

Synopsis: SetRGB4(viewport, register, red, green, blue);

viewport: (struct ViewPort *) Pointer to a ViewPort which colour registers we are going to change. We can find the screen's ViewPort in the Screen structure. (If my_screen is a pointer to a Screen structure, this will get us a pointer to that screen's ViewPort: &my_screen->ViewPort)

register: (long) The colour register you want to change. The screen's Depth decides how many colour registers the screen have:

Depth	Colour Registers
1	0 - 1
2	0 - 3
3	0 - 7
4	0 - 15
5	0 - 31
6	0 - 63

red: Amount of red. (0 - 15)

green: Amount of green. (0 - 15)

blue: Amount of blue. (0 - 15)

Eg: SetRGB4(&my_screen->ViewPort, 2, 15, 15, 0); will change colour register 2 to be light yellow. (Red and green together will be yellow.)

1.9 EXAMPLES

We have now talked about different screens, Workbench Screen and your own Custom Screens. We have looked at how you can change the Custom Screen's display mode, resolution, depth etc, and we have described some important functions. It is now time for us to have some examples to clear up any confusions.

All Examples, both the programs as well as the source codes, are in the same directory as this document.

The printed version of these examples can be found in Appendix A (Part IV: Appendices).

Example1

This program will open a low-resolution, non-Interlaced, eight colour Custom Screen. It will display it for 30 seconds, and then close it.

Example2

Same as Example1 except that the screen will be a high-resolution, Interlaced, 4 colour Custom Screen.

Example3

Same as Example1 except that we will use the TOPAZ_SIXTY Italic style as default font. (See chapter 3 GRAPHICS for more information about text styles.)

Example4

This program will open two screens, one (low-resolution 32 colours) at the top of the display, and one (high-resolution 16 colours) a bit down.

Example5

Same as Example4 except that after 10 seconds the low-resolution screen will move down 75 lines. After another 10 seconds it will be put in front of all other screens. 10 seconds later it will move down another 75 lines. The program will wait 10 seconds before the screens are closed and the program exits.

Example6

This program will open a low-resolution, non-Interlaced, 4 colour Custom Screen. It will after 5 seconds start to change the screens colours, and will after a while close the screen and exit.

2 WINDOWS

2.1 INTRODUCTION

Windows are rectangular boxes which can be as big as the screen or only 1 pixel wide. They can be resized, moved around on the screen, put in front or behind other windows and so on. All this is taken care of Intuition, and your program does not even need to know about it if you do not want.

The aim with Windows is to make the communication between the user and the computer as easy as possible. Windows gives the user a structured display which is easy to understand. It also allows the user to interact with the programs through graphics symbols (gadgets) which can be connected to the windows. (See chapter 4 GADGETS for more information about gadgets.)

Windows are working close together with screens, since the screen's resolution etc affects the way the windows are drawn. Moving a screen will also result in moving all the windows connected to that screen.

2.2 SPECIAL WINDOWS

The normal windows are rectangular boxes with a border around. However, there exist some special types of windows:

2.2.1 BACKDROP WINDOWS

Backdrop Windows will always be behind any other windows. This allows you to have a window in the bottom of the display, and it will stay there, even if the user is using the "depth-arrange gadgets".

Backdrop Windows differ from other windows since:

- It will always open behind all other windows, including other already opened Backdrop Windows.

- The "close-window gadget" is the only System Gadget you can attach to a Backdrop Window. (You can of course attach your own gadgets as usual.)

- It will always be behind of any other windows.

The Backdrop Window is very handy when you want to have a bottom drawing (for example your main display), with some tools etc on top of the bottom window. This allows the user to move around your tools as he/she wants without needing to worry about hiding the windows behind the main display.

Drawing into a Backdrop Window instead of the screen itself is very common since you can then not draw anything at the wrong moment. If a menu (See chapter 5 MENUS for more information) is displayed, the drawing routines will wait automatically, and will therefore not destroy the menu, as it would have happened if you were drawing directly into the screen.

2.2.2 BORDERLESS WINDOWS

Borderless windows are as normal windows except that they (surprise, surprise) do not have any borders. It is very common to combine a Borderless window with a Backdrop window, to cover a screen.

It can be confusing for the user if a Borderless window does not cover the entire display, since he/she will not be able to see where the window's edges are. It is therefore normally best to make a Borderless window as big as the screen.

2.2.3 GIMMEZEROZERO WINDOWS

A Gimmezerozero Window is as a normal window, except that it consist of two drawing areas: one Outer and one Inner window. The Outer window is used for displaying the borders, system gadgets etc, while the Inner window is only used by yourself.

If you are drawing into a normal window you need to start some pixels down/out (11, 1), so you don not draw into the borders etc. However, if you are drawing into a Gimmezerozero Window you do not need to make any adjustments. The top left corner of your inner window is always at the position (0,0).

Gimmezerozero Windows are therefore very handy if you want to make a lot of drawings, without wanting to worry about the borders. The disadvantages is that it takes more memory and processing time than a normal window.

2.2.4 SUPERBITMAP WINDOWS

If you are using a SuperBitMap Window you allocate display memory yourself for the window instead of letting Intuition

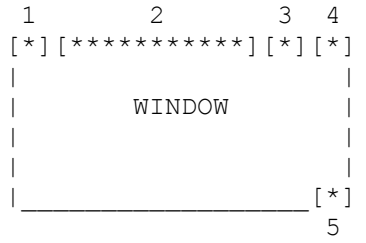
take care of it. The advantage is that you can define a larger drawing area then the size of the window. You can then scroll the drawing area inside the window. A good example of this are the demo programs that followed with you Workbench disk when you bought your computer.

It normally best to combine a SuperBitMap Window together with a Gimmezerozero Window. The borders etc will then NOT be drawn into your SuperBitMap, and will instead be drawn in the Outer window. This allows you to make whatever you want with the inner window without destroying any borders etc.

2.3 SYSTEM GADGETS

The System Gadgets enables the user to move the window, resize it, push it behind other windows etc, without your program even knowing about it. The System Gadgets are controlled by Intuition, and are always looking the same. The only thing you need to do is to tell Intuition which gadgets you want, and the rest is done for you.

The five System Gadgets are placed like this:



1 Close Gadget. Press this to close the window. (This is the only System Gadget which your program have to respond to. Your program will be told that the user has clicked on the Close Gadget, but you need to close the window yourself by calling the function CloseWindow().)

2 Drag Gadget. Pressing the left mouse button somewhere on it and move the mouse while the button is still pressed, will move the window. If your window has a title it will be displayed over the Drag Gadget, but will not interfere with it.

3 Depth-Arrangement Gadget BACK. Clicking on this gadget will push the window behind all other windows.

4 Depth-Arrangement Gadget UP. Clicking on this gadget will put the window in front of all other windows.

5 Sizing Gadget. Pressing the left mouse button somewhere on it and move the mouse while the button is still pressed, will change the size of the window.

2.4 REDRAWING THE WINDOW DISPLAY

Since windows can be dragged around the screen it is very common that they sometimes overlap each other. If the user is then moving away the top window, the bottom window has to be refreshed.

There exist two methods of redrawing the window:
- Simple Refresh. Intuition will only tell you that you need to refresh it, and your program has to redraw everything itself.
- Smart Refresh. Intuition saves the oscured pieces, and replaces them automatically. Needs more memory than Simple Refresh, but redraws the display much faster.

If you have a SuperBitMap Window, you have allocated the display memory yourself, and your window will therefore not be destroyed by other windows.

If you change the size of a Simple Refresh Window, or it is reveled after having been overlapped, Intuition will tell you that you need to refresh the window. A Smart Refresh Window will only ask you to redraw its display if you enlarged it.

The IDCMP message "REFRESHWINDOW" tells you that the window needs to be refreshed. (See chapter 8 IDCMP for more about IDCMP flags.)

IMPORTANT! If your program receive a REFRESHWINDOW message you must call the functions BeginRefresh() - EndRefresh():

- BeginRefresh() will make the redrawing as fast as possible. Only the destroyed pieces of the window will be refreshed.

- EndRefresh() tells Intuition that you are finished with the redrawing.

If you receive a REFRESHWINDOW message and you do not want to redraw the display, you should still call the two functions. This will clear up the system and reorganize the Layer Library.

2.5 INITIALIZE A WINDOW

Before you can open a window you need to initialize a NewWindow structure which look like this:

```
struct NewWindow
{
  SHORT LeftEdge, TopEdge;
```

```
  SHORT Width, Height;
  UBYTE DetailPen, BlockPen;
  ULONG IDCMPFlags;
  ULONG Flags;
  struct Gadget *FirstGadget;
  struct Image *CheckMark;
  UBYTE *Title;
  struct Screen *Screen;
  struct BitMap *BitMap;
  SHORT MinWidth, MinHeight;
  SHORT MaxWidth, MaxHeight;
  USHORT Type;
};
```

LeftEdge: Initial x position of the window.

TopEdge: Initial y position of the window.

Width: Initial width of the window. If the window is connected to a high-resolution screen, it can be anything between 1 and 640. Otherwise (low-resolution screen) it can be between 1 and 320.

Height: Initial height of the window. Can be anything between 1 and the height of the screen.

DetailPen: The colour register used to draw the text with.

BlockPen: The colour register used for block fills etc.

IDCMPFlags: See chapter 8 IDCMP for a list and explanations of the flags.

Flags: If you want any System Gadgets, a special window (Borderless, Backdrop etc), you set the desired flags:

System Gadgets:

WINDOWCLOSE
  This will put the Close Gadget into the top left corner of your window. (Remember that this is the only System Gadget your program need to response to. Intuition will simply inform you that the user wants to close the window, but you need to close the window yourself by calling the CloseWindow() function.

WINDOWDRAG
  This makes the whole title bar into a gadget, which allows the user to move around the Window.

WINDOWDEPTH
  If you want that the user should be able to push the window in front, or behind all other windows you set this flag. The gadgets (to the

back and to the front) will be placed in the top right corner of the window.

WINDOWSIZING
This enables the user to resize the window as desired. (You can specify the maximum and minimum size of the window by setting the MinWidth/MinHeight/MaxWidth/MaxHeight variables)

The Size Gadget will be placed in the right border as default (SIZEBRIGHT), but can also be put in the bottom border (set the flag SIZEBOTTOM) if you want the window display to be as wide as possible.

Special Windows:

BACKDROP
Set this flag if you want a Backdrop Window.

BORDERLESS
Set this flag if you want a Borderless Window.

GIMMEZEROZERO
Set this flag if you want a Gimmezerozero Window.

SUPER_BITMAP
Set this flag if you want a SuperBitMap Window. (If you are going to use your own allocated BitMap you also need to set the BitMap variable to point to your BitMap structure. Explained later in this chapter.)

Refreshing Flags. If you do not use a SuperBitMap Window you need to set one of these two flags:

SIMPLE_REFRESH
Your program must redraw the display itself.

SMART_REFRESH
Intuition will automatically redraw the display if necessary. It is only when the user make the window bigger you need to redraw the display yourself.

Other flags:

REPORTMOUSE
Set this flag if you want to receive the pointer movements as x,y coordinates. (See chapter 8 IDCMP for more information.)

NOCAREREFRESH
Set this flag if you do not want to receive any messages telling you to redraw your window.

RMBTRAP
Set this flag if you do not want that the user to be able to access any menus while this window is active. (If you want that the right mouse button on the mouse should be used for something else than menu operations, you should set this flag. See chapter 8 IDCMP for more information.)

ACTIVATE
Set this flag if you want the window to become active when it is opened. The user can of course activate some other window later if he/she wants. (Clicking inside a window will activate it, and all input will go to the new active window. All other widows will become inactive, and their title bars will become "ghosted".)

FirstGadget: A pointer to the first Gadget in your list, or NULL if you do not have any own gadgets connected to the window. (See chapter 4 GADGETS for more information about gadgets.)

CheckMark: A pointer to an Image structure (See chapter 3 GRAPHICS for more information about Image structures) which will be used for selected menus items. (See chapter 7 MENUS for more information about menus.) If no pointer is given (NULL) Intuition will use the default checkmark.

Title: A pointer to a NULL-terminated string that will be used as window's title. The string will appear on the title bar, at the top of the window.

Screen: A pointer to your Custom Screen, or NULL if the window should be connected to the Workbench Screen. (If the window is connected to the Workbench Screen set the Type variable to WBENCHSCREEN, otherwise set it to CUSTOMSCREEN.)

BitMap: If you are going to use a SuperBitMap Window, then you need to give Intuition a pointer to your own initialized BitMap structure, otherwise write NULL. (Remember to set the Flag SUPER_BITMAP, and you should probably also make the window into a Gimmezerozero Window.)

If you have asked for the System Gadget WINDOWSIZING you need to decide the minimum/maximum size of your window, otherwise you can ignore these variables. If any variable is set to zero the initial size (Width/Height) will be used as max/min size:

MinWidth: Minimum width of the window.

MinHeight: Minimum height of the window.

MaxWidth: Maximum width of the window.

MaxHeight: Maximum height of the window.

Type: If your window should be connected to the Workbench Screen you write WBENCHSCREEN, otherwise you write CUSTOMSCREEN. (If you set the CUSTOMSCREEN flag you need to give the Screen variable a pointer to your already opened Custom Screen.

It is common that you declare and initialize the NewWindow structure with your requirements, except that you ignore (NULL) the Screen pointer. You then open your Custom Screen, and first now initialize the Screen pointer with the pointer returned from the OpenScreen() function. (See Example 2 for more information.)

2.6 OPEN A WINDOW

The procedure to open windows is very similar to open Custom Screens. The idea is that you declare a NewWindow structure (similar to NewScreen) and initialize it with the requirements. You then simply call the function OpenWindow() (similar to OpenScreen()) with your NewWindow structure, and the function returns a pointer to your Window structure. You will now not need the NewWindow structure any more (if you do not want to open more windows using the same structure of course).

This is how you call the OpenWindow() function:

```
my_window = OpenWindow( &my_new_window );
```

my_window has been declared as:

```
  struct Window *my_window;
```

my_new_window has been declared as:

```
  struct NewWindow *my_new_window;
```

and has been initialized with your requirements.

OpenWindow() will return a pointer to your Window structure. If it could not open the Window (for example, not enough memory) it will return NULL. Remember therefore to check what OpenWindow() returned:

```
if( my_window == NULL )
{
```

```
  /* PANIC! Could not open the window */
}
```

2.7 WINDOW STRUCTURE

Once you have opened the window you will receive a pointer (my_window) to a Window structure:

```
struct Window
{
    struct Window *NextWindow;  /* Pointer to next window. */
    SHORT LeftEdge, TopEdge;    /* Position of the window. */
    SHORT Width, Height;        /* Size of the window. */
    SHORT MouseY, MouseX;       /* Position of the pointer */
                                /* relative to the top left */
                                /* corner of the window. */
    SHORT MinWidth, MinHeight;  /* Minimum/Maximum size of */
    USHORT MaxWidth, MaxHeight; /* the window. */

    ULONG Flags;                /* The window's flags. */

    struct Menu *MenuStrip;     /* Pointer to the window's */
                                /* first menu. */

    UBYTE *Title;               /* The window's title. */

    struct Requester *FirstRequest;
    struct Requester *DMRequest;
    SHORT ReqCount;

    struct Screen *WScreen;     /* A pointer to the Screen */
                                /* which the window is */
                                /* connected to. */
    struct RastPort *RPort;     /* The window's RastPort. */

    BYTE BorderLeft, BorderTop, BorderRight, BorderBottom;

    struct RastPort *BorderRPort; /* If your window is a */
                                  /* Gimmezerozero this a */
                                  /* pointer the the Outer */
                                  /* window's RastPort. */

    struct Gadget *FirstGadget;
    struct Window *Parent, *Descendant;

    USHORT *Pointer;            /* Data for the Pointer. */
    BYTE PtrHeight;             /* Height of the pointer. */
    BYTE PtrWidth;              /* Width of the pointer. */
    BYTE XOffset, YOffset;      /* "Hot Spot" of the pointer. */

    ULONG IDCMPFlags;           /* The IDCMP flags. */

    struct MsgPort *UserPort, *WindowPort;
```

```
    struct IntuiMessage *MessageKey;

    UBYTE DetailPen, BlockPen;
    struct Image *CheckMark;

    UBYTE *ScreenTitle;         /* The title of the screen */
                                /* which the window is */
                                /* connected to. */

    /* These are only used if you have opened a */
    /* Gimmezerozero window: */
    SHORT GZZMouseX; /* Position of the mouse relative to */
    SHORT GZZMouseY; /* the inner window. */
    SHORT GZZWidth;  /* Size of the inner window. */
    SHORT GZZHeight;

    UBYTE *ExtData;
    BYTE *UserData;
    struct Layer *WLayer;
    struct TextFont *IFont;
};
```

2.8 OPEN A SUPERBITMAP WINDOW

The difference in opening a normal window and a SuperBitMap window is that Intuition will allocate display memory for the normal window automatically, while a SuperBitMap window has to allocate the memory itself. However, it is actually not so hard as it may seem. You just need to follow these steps:

1. Declare and initialize a NewWindow structure with your requirements.
   (Set the Flags = SUPER_BITMAP, and BitMap=NULL)

```
   struct NewWindow my_new_window={ .... };
```

2. Declare a BitMap structure:

```
   struct BitMap my_bitmap;
```

3. Initialize your own BitMap by calling the function:

```
   InitBitMap( &my_bitmap, Depth, Width, Height );
     &my_bitmap: A pointer to the my_bitmap structure.
     Depth:      Number of bitplanes to use.
     Width:      The width of the BitMap.
     Height:     The height of the BitMap.
```

4. Allocate display memory for the BitMap:

```
   for( loop=0; loop < Depth; loop++)
     if( (my_bitmap.Planes[loop] = AllocRaster( Width,
```

```
      Height )) == NULL )
    {
      /* PANIC! Not enough memory */
    }
```

5. After you have allocated the display memory, it normally best to clear the Bitplanes:

```
for( loop=0; loop < Depth; loop++)
  BltClear(my_bitmap.Planes[loop], RASSIZE(Width, Height), 0);
```

6. Make sure the NewWindow's BitMap pointer is pointing to your

   BitMap structure:

```
  my_new_window.BitMap=&my_bitmap;
```

7. At last you can open the window:

```
my_window = OpenWindow( &my_new_window );
```

8. Do not forget to close the window, AND deallocate the display memory:

```
CloseWindow( my_window );

for( loop=0; loop < Depth; loop++)
 if( my_bitmap.Planes[loop] )
   FreeRaster( my_bitmap.Planes[loop], Width, Height );
```

2.9 MAKE YOUR OWN CUSTOM POINTER

Each window can have its individual pointer. When the window becomes active the pointer can change image. This makes it easier for the user to see which window is active, and can also give the user a hint about what the computer is doing. For example, a wordprocessor can have a pointer that looks similar to a pencil, and when the computer is busy, the pointer change to a little Zzz symbol.

If your window is going to use a "custom" pointer you need to:

1. Allocate and initialize a Sprite data structure.
2. Use the function SetPointer() to change the window's pointer.

When you are going to make a new image for your pointer, you first need to sketch how it should look like. The pointer is actually a Sprite (See chapter 10 SPRITES for more information about Sprites), and can therefore be up to 16 pixels wide, and be as tall as you want. You may use 3 colours, and transparent. (Since the pointer is a Sprite (no. 0) it can move between

different screens with different resolutions, without destroying the display.)

A nice "arrow" pointer (16 pixels wide, 16 lines high) may look something like this:

```
0000000200000000      0: Transparent
0000002200000000      1: Red
0000023200000000      2: Black
0000231200000000      3: White
0002311200000000
0023111222222200
0231111133333320
2311111111111132
0211111111111112
0021111222221112
0002111200023112
0000211200023112
0000021200023112
0000002200023112
0000000200023112
0000000000022222
```

You now need to translate this into Sprite Data. Each line of the pointer will be translated into two words of data. The first word represents the first Bitplane, and the second word the second Bitplane. The idea is that if you want colour 0 both Bitplane zero and one should be 0, if you want colour 1 Bitplane zero should be 1 and Bitplane one 0 and so on:

Colour	Bitplane One	Bitplane Zero	Since
0	0	0	Binary 00 = Decimal 0
1	0	1	" 01 = " 1
2	1	0	" 10 = " 2
3	1	1	" 11 = " 3

The data for the pointer would then look like this:

```
   Bitplane ZERO          Bitplane ONE

0000 0000 0000 0000    0000 0001 0000 0000
0000 0000 0000 0000    0000 0011 0000 0000
0000 0010 0000 0000    0000 0111 0000 0000
0000 0110 0000 0000    0000 1101 0000 0000
0000 1110 0000 0000    0001 1001 0000 0000
0001 1110 0000 0000    0011 0001 1111 1100
0011 1111 1111 1100    0111 0000 1111 1110
0111 1111 1111 1110    1100 0000 0000 0011
0011 1111 1111 1110    0100 0000 0000 0001
0001 1110 0000 1110    0010 0001 1111 0001
0000 1110 0000 1110    0001 0001 0001 1001
0000 0110 0000 1110    0000 1001 0001 1001
0000 0010 0000 1110    0000 0101 0001 1001
0000 0000 0000 1110    0000 0011 0001 1001
0000 0000 0000 1110    0000 0001 0001 1001
```

```
0000 0000 0000 0000    0000 0000 0001 1111
```

The last step is to translate the binary numbers to base hexadecimal. Group the binary number in four and translate it to Hexadecimal:

```
0000 =  0
0001 =  1
0010 =  2
0011 =  3
0100 =  4
0101 =  5
0110 =  6
0111 =  7
1000 =  8
1001 =  9
1010 =  A
1011 =  B
1100 =  C
1101 =  D
1110 =  E
1111 =  F
```

The result will look like this:

```
  ZERO:    ONE:
  ------------
  0000    0100
  0000    0300
  0200    0700
  0600    0d00
  0E00    1900
  1E00    31FC
  3FFC    60FE
  7FFE    c003
  3FFE    4001
  1E0E    21F1
  0E0E    1119
  060E    0919
  020E    0519
  000E    0319
  000E    0119
  0000    001F
```

Since the Amiga need to store the position of the pointer, the size etc, you should also declare two empty words at the top, and to empty words at the bottom of the Sprite data. These words will be initialized and maintained by Intuition, so you do not need to bother about them.

A declaration and initialization of a nice arrow as Sprite data for a pointer would therefore be: (A hexadecimal number as 3FFE will written in C be 0x3FFE.)

```
UWORD chip my_sprite_data[36]=
```

```
{
 0x0000, 0x0000, /* Used by Intuition only. */


 0x0000, 0x0100,
 0x0000, 0x0300,
 0x0200, 0x0700,
 0x0600, 0x0D00,
 0x0E00, 0x1900,
 0x1E00, 0x31FC,
 0x3FFC, 0x60FE,
 0x7FFE, 0xC003,
 0x3FFE, 0x4001,
 0x1E0E, 0x21F1,
 0x0E0E, 0x1119,
 0x060E, 0x0919,
 0x020E, 0x0519,
 0x000E, 0x0319,
 0x000E, 0x0119,
 0x0000, 0x001F,

 0x0000, 0x0000  /* Used by Intuition only. */
};
```

It is actually not so complicated once you got used to it, but to make the life a bit easier I have written some utilities which will help you with translating graphics to sprite data. See the TOOL drawer for more information.

The last step is to call the function SetPointer() which would look like this:

```
SetPointer( my_window, my_sprite_data, 16, 16, 0, -7);
```

```
  my_window:      Pointer to the window.
  my_sprite_data: Pointer to the Sprite Data.
  16:             Height.
  16:             Width. (Must be 16 or less!)
  0:              XOffset, left side. (Position of the Hot Spot)
  -7:             YOffset, 7 lines down.         -"-
```

The "Hot Spot" is the pixel which should be the sensitive spot on the pointer. On the default pointer it is on the top left side, but on our custom pointer it should be on the left side (0), half way down (-7).

To restore the default pointer you simply call the function ClearPointer():

```
ClearPointer( my_window );
```

Since you are going to use graphics data it is important that the data is placed in the Chip Memory. Chip Memory is the first 512 KB of RAM, and the graphics routines in the Amiga demands that Sprite Data, as well as other graphics data, is placed

there. (The reason why is that the CPU on the Amiga can reach the first 9MB of RAM, while the Blitter (a graphics co-processor) only can reach the first 512 KB.

There will soon be a new Chip set for the Amiga 500 and 2000 which enables the Blitter etc to reach the first 1 MB, but up to now the data MUST be placed somewhere in the memory between $00000 to $7FFFF.)

If you are using Lattice C Compiler V5.0 or higher it is very simple. You only need to place the little word "chip" in front of the data. Eg:

```
UWORD chip graphics_data[]={ ... };
```

On the Lattice C Compiler V4.0 you can compile the source with the added command: "-acdb" which will load everything into the Chip Memory. Eg:

```
lc -acdb -Lm my_program.c
```

This will do both the lc1 and lc2 compilation, and it will also link the code together with the standard math library. Everything is loaded into Chip Memory.

If you do not have one of these C compilers you will probably find out how to do it in your manual. REMEMBER! If you do not tell the compiler to always put the graphics in the Chip Memory you can end up with horrible bugs. It may sometimes work, especially if you have an unexpanded Amiga, but it will probably crash on an expanded Amiga.

2.10 FUNCTIONS

Here are some functions that are often used together with windows:

OpenWindow()

This function will open a window with the characteristics defined in the NewWindow structure. It returns a pointer to a Window structure.

If you are going to use the Workbench screen, and it has been closed, it will automatically reopen. If you on the other hand is going to connect the window to a Custom screen, you need to open it yourself before calling the OpenWindow() function.

```
Synopsis:      my_window = OpenWindow( my_new_window );

my_window:     (struct Window *) Pointer to a Window structure
               or NULL if the window could not be opened.

my_new_window: (struct NewWindow *) Pointer to a NewWindow
               structure which has been initialized with
               your requirements.
```

CloseWindow()

This function will close a window you have previously opened. Remember that you need to close all windows connected to a screen before you may close the screen, and all opened windows must have been closed before your program quits.

Synopsis: CloseWindow(my_window);

my_window: (struct Window *) Pointer to a Window structure which has previously been initialized by an OpenWindow() call.

MoveWindow()

This function will move a window. It has the same effect as if the user would have moved the window by using the Drag Gadget.

Synopsis: MoveWindow(my_window, delta_x, delta_y);

my_window: (struct Window *) Pointer to a Window structure which has previously been initialized by an OpenWindow() call.

delta_x: (long) Deltamovement horizontally.

delta_y: (long) Deltamovement vertically.

SizeWindow()

This function will change the size of the window as desired. It has the same effect as if the user would have resized the window by using the Size Gadget.

Synopsis: SizeWindow(my_window, delta_x, delta_y);

my_window: (struct Window *) Pointer to a Window structure which has previously been initialized by an OpenWindow() call.

delta_x: (long) Number of pixels the horizontally size of the window will change.

delta_y: (long) Number of pixels the vertically size of the window will change.

WindowToFront()

This function will put the window in front of all other

windows.

Synopsis: WindowToFront(my_window);

my_window: (struct Window *) Pointer to a Window structure which has previously been initialized by an OpenWindow() call.

WindowToBack()

This function will push the window behind all other windows.

Synopsis: WindowToBack(my_window);

my_window: (struct Window *) Pointer to a Window structure which has previously been initialized by an OpenWindow() call.

SetWindowTitles()

This function allows you to change the window title after the window has been opened.

Synopsis: SetWindowTitles(my_window, window_t, screen_t);

my_window: (struct Window *) Pointer to a Window structure which has previously been initialized by an OpenWindow() call.

window_t: (char *) Pointer to a NULL-terminated string which will become the window's title, or
0 : clear title bar, or
-1 : keep the old title.

screen_t: (char *) Pointer to a NULL-terminated string which will become the window's screen title, or
0 : clear title bar, or
-1 : keep the old title.

WindowLimits()

This function will change the maximum/minimum size limits of the window. Any values which are set to 0 will remain unchanged.

Synopsis: WindowLimits(my_window, min_w, min_h, max_w, max_h);

my_window: (struct Window *) Pointer to a Window structure which has previously been initialized by an OpenWindow() call.

min_w: (long) Minimum width of the window.

min_h: (long) Minimum height of the window.

max_w: (long) Maximum width of the window.

max_h: (long) Maximum height of the window.

SetPointer()

This function allows you to change the window's pointer.

Synopsis: SetPointer(my_window, data, height, width, x, y);

my_window: (struct Window *) Pointer to a Window structure which has previously been initialized by an OpenWindow() call.

data: (short *) Pointer to the Sprite data.

width: (long) The width of the pointer. Less or equal to 16.

height: (long) The height of the pointer. Can be any height.

x: (long) The pointer's "Hot Spot" x position.

y: (long) The pointer's "Hot Spot" y position.

ClearPointer()

This will remove the "custom" pointer, and replace it with Intuition's default pointer.

Synopsis: ClearPointer(my_window);

my_window: (struct Window *) Pointer to a Window structure which has previously been initialized by an OpenWindow() call.

ReportMouse()

You can call this function if you want the window to start/ stop reporting the mouse position. (See chapter 8 IDCMP for more information about REPORTMOUSE.)

Synopsis: ReportMouse(my_window, boolean);

my_window: (struct Window *) Pointer to a Window structure which has previously been initialized by an OpenWindow() call.

boolean: (long) Set to TRUE if you want the window to start reporting mouse position, else set to FALSE, and the window will stop reporting.

BeginRefresh()

This function will speed up your redrawing of the window. You should call this function before you start to refresh the window, and only the parts that needs to be redrawn are redrawn.

Synopsis: BeginRefresh(my_window);

my_window: (struct Window *) Pointer to a Window structure which has previously been initialized by an OpenWindow() call.

EndRefresh()

This function will tell Intuition that you have finished with your redrawings. IMPORTANT! If you receive a REFRESHWINDOW message, you must call the functions BeginRefresh() and EndRefresh(), even if you do not want to redraw anything.

Synopsis: EndRefresh(my_window);

my_window: (struct Window *) Pointer to a Window structure which has previously been initialized by an OpenWindow() call.

2.11 EXAMPLES

We have now looked at different types of windows, how to connect System Gadgets, the steps to customize your pointer and much more. It is now again time to look at some examples.

The printed version of these examples can be found in Appendix A (Part IV: Appendices).

Example1

This program will open a normal window which is connected to the Workbench Screen. It will display it for 30 seconds, and then close it.

Example2

This program will open a high resolution 16 colour Custom Screen and a normal window which is connected to it. It will display it for 30 seconds, and then close the Custom Screen and the window.

Example3

This program will open a normal window which is connected to the Workbench Screen. The window will use all System Gadgets, and will automatically Activate the window. It will display it for 30 seconds, and then close it. (Remember that the Close Gadget does NOT close the window by itself, it will only inform you that the user wants to close it. But in this example we will not listen to what the user wants.)

Example4

This program will open two normal windows which are connected to the Workbench Screen. The windows will use all System Gadgets. It will display them for 30 seconds, and then close them.

Example5

This program will open a Borderless window which is connected to the Workbench Screen. It will display it for 30 seconds, and then quit.

Example6

Same as Example5 except that the window will also use all System Gadgets.

Example7

This program will open three windows, two are normal and the third is a Backdrop window. The windows will use all System Gadgets, except the Backdrop window, which only can use the close-window gadget. After 30 seconds the program quits. (Try to push either window 1 or 2 behind the Backdrop window.)

Example8

This program will open a SuperBitMap window which is connected to the Workbench Screen. Since it is a SuperBitMap we also make the window into a Gimmezerozero window. The window will use all System Gadgets, and some boxes will be drawn. It will display the window for 30 seconds, and then close it. (Shrink the window, and then enlarge it again, and you will noticed that the lines are still there!)

Example9

This program will open a normal window with all system gadgets connected to it. If you activate the window, the pointer will change shapes into a "nice" arrow.

Example10

This program will open a two normal windows with all system gadgets connected to them. If the first window is Activated, the pointer will change shapes into a Zzz symbol, if the second window is activated, the pointer will look like a pistol.

3 GRAPHICS

3.1 INTRODUCTION

We have now looked at how to open different types of screens, and how to open different types of windows connected to the screens etc. It is all very good, but what would Intuition be without any graphics to display in the screens/windows?

Intuition's main idea is to make the communication between the program and the user as easy as possible. Graphics enables you to display the results that the computer has calculated in an understandable way (a picture says more than...). Graphics makes it also much easier to use the program, and can warn the user if something dangerous is going to happen: It is very easy to press the wrong button if you have got a question like "OK to erase disk?", but if there also had been a picture showing a cross with the text "RIP" on, you would probably have been a bit more careful.

There exist both a low- and a high-level way of making graphics. The low-level approach (Graphics Primitives) is not described in this chapter since it is not related to Intuition. The low-level graphics routines are very fast, but if you are not careful they can trash menus etc. We will here concentrate our self on the high-level approach which is supported by Intuition, and is a safe way of drawing.

3.2 LINES TEXT PICTURES

Intuition gives you three different methods of making graphics:
- You can draw lines (Borders).
- Print text (IntuiText).
- Or you can directly print graphics images (Images).

3.3 BORDERS

Border has a bit misleading name since you are not limited to only draw borders, you may draw any kind of shapes which is made out of connecting lines. A Border structure may also be connected to other Border structures, so everything which can be drawn with lines, can be drawn with Intuition's Border structure.

3.3.1 THE BORDER STRUCTURE

When you want to draw lines you need to declare and initialize a Border structure which look like this:

```
struct Border
{
  SHORT LeftEdge, TopEdge;
  SHORT FrontPen, BackPen, DrawMode;
  SHORT Count;
  SHORT *XY;
  struct Border *NextBorder;
};
```

LeftEdge, TopEdge: Start position of the lines.

FrontPen: Colour register used to draw the lines.

BackPen: This variable is for the moment unused.

DrawMode: Must be either JAM1 or XOR. If the JAM1 flag is set, the colour specified (FrontPen) will be used to draw the lines regardless what the background colour is. The XOR flag makes the lines to be drawn with the binary complement of the background colours.

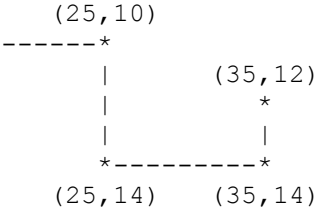
Count: The number of coordinates there is in the XY array. (See below for more information.)

XY: A pointer to an array of coordinates used to draw the lines. (See below for more information.)

NextBorder: A pointer to the next Border structure if there exist one, else NULL.

3.3.2 COORDINATES

If you want to draw this:



The array of coordinates would look like this:

```
SHORT my_points[]=
{
  10,10, /* Start at position (10,10) */
  25,10, /* Draw a line to the right to position (25,10) */
  25,14, /* Draw a line down to position (25,14) */
  35,14, /* Draw a line to the right to position (35,14) */
  35,12  /* Finish of by drawing a line up to position (35,12) */
};
```

The array contains 5 pair of coordinates, so the variable Count should be set accordingly. The entire Border structure would therefore look something like this:

```
struct Border my_border=
{
  0, 0,         /* LeftEdge, TopEdge. */
  3,            /* FrontPen, colour register 3. */
  0,            /* BackPen, for the moment unused. */
  JAM1,         /* DrawMode, draw the lines with colour 3. */
  5,            /* Count, 5 pair of coordinates in the array. */
  my_points,    /* XY, pointer to the array with the */
                /* coordinates. */
                /* (Remember my_points == &my_points[0]) */
  NULL          /* NextBorder, no other Border structures */
                /* comes after this one. */
};
```

3.3.2 HOW TO USE THE BORDER STRUCTURE

The Border structure is either used to draw lines in a screen or window, but can also be used to draw lines connected to a Gadget, Requester or Menu. (See chapter 4, 5 and 7 for more information about Gadgets, Requesters and Menus.)

If you want to connect a Border structure with a Gadget etc, you simply initialize the Gadget structure with a pointer to the Border structure, and Intuition will draw the lines for you. This will be explained later.

If you want to draw the lines in a screen or window you need to tell Intuition that you want it to use the structure to make some lines. You do it by calling the function DrawBorder():

Synopsis: DrawBorder(rast_port, border, x, y);

rast_port: (struct RastPort *) Pointer to a RastPort.

If the lines should be drawn in a window, and my_window is a pointer to that window, you write: my_window->RPort.

If the lines should be drawn in a Screen, and my_screen is a pointer to that screen, you write: my_screen->RastPort.

border: (struct Border *) Pointer to a Border structure which has been initialized with your requirements.

x: (long) Number of pixels added to the x coordinates.

y: (long) Number of lines added to the y coordinates.

A call to draw some lines using my_border structure would look something like this:

```
DrawBorder( my_window->RPort, &my_border, 0, 0 );
```

If you have created a Border structure which, for example, draws a box, you can draw several boxes at different places just by changing the LeftEdge, RightEdge fields of the Border structure, or by changing the x, y fields in the function call. This shows how versatile Intuition's high-level way of drawing is.

3.4 TEXT

Printing text in the Display is supported by the IntuiText structure. It is quite similar to the Border structure, and is executed in the same way. The only difference is that you need to tell Intuition what Font you want to use, and what text to print.

3.4.1 THE INTUITEXT STRUCTURE

When you want to print text you need to declare and initialize a IntuiText structure which look like this:

```
struct IntuiText
{
  UBYTE FrontPen, BackPen;
  UBYTE DrawMode;
  SHORT LeftEdge;
  SHORT TopEdge;
  struct TextAttr *ITextFont;
  UBYTE *IText;
  struct IntuiText *NextText;
};
```

FrontPen: Colour register used to draw the text with.

BackPen: Colour register used to draw the background of the text.

DrawMode: There exist three different way of printing the text:

- JAM1 The colour specified (FrontPen) will be used to draw the text with, the background is unchanged.
- JAM2 The FrontPen colour will be used to draw the text with, the background is drawn with the BackPen colour.
- XOR The text is drawn with the the binary complement of the background.

LeftEdge, TopEdge: Start position of the text.

ITextFont: A pointer to a TextAttr structure which tells Intuition what font, size and style etc to print the text with. Set to NULL if you want to use the default font. (See below for more information.)

IText: A pointer to a NULL-terminated string that should be printed.

NextText: A pointer to the next IntuiText structure if there exist one, else write NULL.

3.4.2 FONTS

You can tell Intuition to print the text with a specific font/ style. You only need to declare and initialize a TextAttr structure, and give your IntuiText structure a pointer to the TextAttr structure.

The TextAttr structure look like this:

```
struct TextAttr
{
  STRPTR ta_Name;
  UWORD ta_YSize;
  UBYTE ta_Style
  UBYTE ta_Flags;
};
```

We will for the moment only discuss how to access the ROM-fonts (the fonts which are always available in ROM), and will wait with Disk-fonts (fonts which your program can access from the disk).

There exist only one ROM-font for the moment, and that one is called Topaz. It exist in two sizes, 64/32 and 80/40 (high-/low-resolution) characters per line, and can be printed in five different styles, normal, bold, italic, underlined and extended, which can be combined as desired.

ta_Name: Name of the font. Set it to "topaz.font" for the moment.

ta_YSize: Font height. Can either be TOPAZ_SIXTY (64/32) or TOPAZ_EIGHTY (80/40).

ta_Style: Style (normal, bold, underlined, italic and extended). Set the desired flags:

FS_NORMAL	No special style.
FSF_EXTENDED	Extended font, wider than normal.
FSF_ITALIC	Italic, slanted 1:2 to the right.
FSF_BOLD	Bold, thicker than normal.
FSF_UNDERLINED	Underlined, line under the baseline.

ta_Flags: Preferences. Should for the moment be set to FPF_ROMFONT.

Here is an example on how you can print some text with underlined, italic characters:

```
struct TextAttr my_font=
{
  "topaz.font",                 /* Topaz font. */
  TOPAZ_EIGHTY,                 /* 80/40 characters. */
  FSF_ITALIC | FSF_UNDERLINED,  /* Underlined italic chars. */
  FPF_ROMFONT                   /* Exist in ROM. */
};

UBYTE my_text[]="This is the text that will be printed!";

struct IntuiText my_intui_text=
{
  2,         /* FrontPen, colour register 2. */
  3,         /* BackPen, colour register 3. */
  JAM2,      /* DrawMode, draw the characters with colour 2, */
             /* on a colour 3 background. */
  0, 0,      /* LeftEdge, TopEdge. */
  &my_font,  /* ITextFont, use my_font. */
  my_text,   /* IText, the text that will be printed. */
             /* (Remember my_text = &my_text[0].) */
  NULL       /* NextText, no other IntuiText structures are */
             /* connected. */
};
```

3.4.3 HOW TO USE THE INTUITEXT STRUCTURE

The IntuiText structure is either used to print text in a screen or window, but can also be used to print text connected to a Gadget, Menu or Requester. Same as with drawing lines with the Border structure.

When you want to print text in a window/screen you simply call the function PrintIText():

Synopsis: PrintIText(rast_port, intui_text, x, y);

rast_port: (struct RastPort *) Pointer to a RastPort.

If the text should be printed in a window, and my_window is a pointer to that window, you write: my_window->RPort.

If the text should be printed in a Screen, and my_screen is a pointer to that screen, you write: my_screen->RastPort.

intui_text: (struct IntuiText *) Pointer to a IntuiText structure which has been initialized with your requirements.

x: (long) Number of pixels added to the x position of the characters.

y: (long) Number of lines added to the y position of the characters.

A call to print some text using my_intui_text structure would look something like this:

```
PrintIText( my_window->RPort, &my_intui_text, 0, 0 );
```

3.5 IMAGES

We have now looked at how to draw lines and print text. We will finish off by looking at how to print a whole image directly. The procedure can be devided into three steps:

1. Declare and initialize the data which should be drawn.
2. Declare and initialize an Image structure.
3. Call the function DrawImage() to draw the image.

3.5.1 IMAGE DATA

Intuition wants to have the image data structured into blocks of 16-bit memory words (UWORD), organized into rectangular areas (BitPlanes). You may have up to 6 BitPlanes (4 if you are displaying the image on a high-resolution screen). The colour of each pixel is determined by the binary number of the BitPlanes. For example, to get a pixel with colour 6 on a screen with a Depth of 4, BitPlane zero and three should be 0, and BitPlane one and two should be 1, since the binary number 0110 is equal to the decimal number 6.

A little arrow can for example look like this:
(1 BitPlane = 2 colours)

```
Image      16-Bit memory words    Hexadecimal representation
-----------------------------------------------------------

0001000    0001 0000 0000 0000    1 0 0 0
0011100    0011 1000 0000 0000    3 8 0 0
0111110    0111 1100 0000 0000    7 C 0 0
1111111    1111 1110 0000 0000    F E 0 0
0001000    0001 0000 0000 0000    1 0 0 0
0001000    0001 0000 0000 0000    1 0 0 0
0001000    0001 0000 0000 0000    1 0 0 0
0001000    0001 0000 0000 0000    1 0 0 0
```

Even if the Image only is 7 pixels wide, we needed to make the Image data 16 (bits) pixels wide, since Intuition wants the data ordered into 16-bit memory words. (Of course, when we print it out we can tell Intuition that the image should be only 7 pixels wide.)

Each group of four pixels is then translated into a hexadecimal value. See table:

```
  Binary => Hexadecimal
  ---------------------
  0000 =    0
  0001 =    1
  0010 =    2
  0011 =    3
  0100 =    4
  0101 =    5
  0110 =    6
  0111 =    7
  1000 =    8
  1001 =    9
  1010 =    A
  1011 =    B
  1100 =    C
  1101 =    D
  1110 =    E
  1111 =    F
```

A declaration and initialization of the data for the arrow image will therefore look like this:

```
UWORD chip my_image_data[]=
{
  0x1000, /* BitPlane ZERO */
  0x3800,
  0x7C00,
  0xFE00,
  0x1000,
  0x1000,
  0x1000,
  0x1000
};
```

Since the image data is made out of one BitPlane, the arrow will be drawn in colour 1, and the background in colour 0.

If the image is wider than 16 pixels you need more than one word per line. Here is an image which is 20 pixels wide, and will therefore require two words per line:

```
Image                  16-Bit memory words (2 words)
-------------------------------------------------------------

11111111111111111111   1111 1111 1111 1111  1111 0000 0000 0000
00111110000001111100   0011 1110 0000 0111  1100 0000 0000 0000
00001111100111110000   0000 1111 1001 1111  0000 0000 0000 0000
00111110000001111100   0011 1110 0000 0111  1100 0000 0000 0000
11111111111111111111   1111 1111 1111 1111  1111 0000 0000 0000
```

We translate the binary numbers into hexadecimal, and we end up with:

```
UWORD chip my_image_data[]=
{
  0xFFFF, 0xF000, /* BitPlane ZERO */
  0x3E07, 0xC000,
  0x0F9F, 0x0000,
  0x3E07, 0xC000,
  0xFFFF, 0xF000
};
```

If we want more than two colours we need more than one BitPlane. A four-colour face can look something like this:

```
003333300000     0: Blue (Normal Workbench colours)
033333330000     1: White
332232233000     2: Black
323333323000     3: Orange
331131133000
331131133000
331232133000
331232133000
```

```
333333333000
332333233000
033222330000
033333330000
003333300000
```

We translate it into 16-Bit memory words organized into two Bitplanes (Two BitPlanes = 4 colours):

Colour	Bitplane ONE	Bitplane ZERO	SINCE
0	0	0	00 (b) = 0 (d)
1	0	1	01 (b) = 1 (d)
2	1	0	10 (b) = 2 (d)
3	1	1	11 (b) = 3 (d)

```
Bitplane ONE          Bitplane ZERO
----------------------------------------

0011 1110 0000 0000   0011 1110 0000 0000
0111 1111 0000 0000   0111 1111 0000 0000
1111 1111 1000 0000   1100 1001 1000 0000
1111 1111 1000 0000   1011 1110 1000 0000
1100 1001 1000 0000   1111 1111 1000 0000
1100 1001 1000 0000   1111 1111 1000 0000
1101 1101 1000 0000   1110 1011 1000 0000
1101 1101 1000 0000   1110 1011 1000 0000
1111 1111 1000 0000   1111 1111 1000 0000
1111 1111 1000 0000   1101 1101 1000 0000
0111 1111 0000 0000   0110 0011 0000 0000
0111 1111 0000 0000   0111 1111 0000 0000
0011 1110 0000 0000   0011 1110 0000 0000
```

Each group of four pixels is then translated into a hexadecimal value:

```
Bitplane ONE          Bitplane ZERO
-----------------------------------

3 E 0 0               3 E 0 0
7 F 0 0               7 F 0 0
F F 8 0               C 9 8 0
F F 8 0               B E 8 0
C 9 8 0               F F 8 0
C 9 8 0               F F 8 0
D D 8 0               E B 8 0
D D 8 0               E B 8 0
F F 8 0               F F 8 0
F F 8 0               D D 8 0
7 F 0 0               6 3 0 0
7 F 0 0               7 F 0 0
3 E 0 0               3 E 0 0
```

A declaration and initialization of the data for the face image will therefore look like this:

```
USHORT chip my_image_data[]=
{
  0x3E00, /* Bitplane ZERO */
  0x7F00,
  0xC980,
  0xBE80,
  0xFF80,
  0xFF80,
  0xEB80,
  0xEB80,
  0xFF80,
  0xDD80,
  0x6300,
  0x7F00,
  0x3E00,

  0x3E00, /* Bitplane ONE */
  0x7F00,
  0xFF80,
  0xFF80,
  0xC980,
  0xC980,
  0xDD80,
  0xDD80,
  0xFF80,
  0xFF80,
  0x7F00,
  0x7F00,
  0x3E00
};
```

Remember that all Image Data MUST be in Chip Memory!

3.5.2 THE IMAGE STRUCTURE

The Image structure look like this:

```
struct Image
{
  SHORT LeftEdge, TopEdge;
  SHORT Width, Height, Depth;
  SHORT *ImageData;
  UBYTE PlanePick, PlaneOnOff;
  struct Image *NextImage;
};
```

LeftEdge: X position of the Image.

TopEdge: Y position of the Image.

Width:	The actual width of the Image.
Height:	The height of the Image.
Depth:	Number of Bitplanes used for the Image.
ImageData:	A pointer to the Image data.
PlanePick:	Which Bitplanes of the displaying element should be changed/affected by the Image. (See below for more information.)
PlaneOnOff:	What should happen to the Bitplanes of the displaying element that were not affected. Should they be filled with 1's or 0's. (See below for more information.)
NextImage:	A pointer to the next Image structure if there exist one, else NULL.

3.5.3 PLANEPICK

The advantages with the PlanePick/PlaneOnOff fields are that you can print the Image into a display of any depth, print the same Image in different colours, and it will eliminate unnecessary memory-waste when using coloured Images.

PlanePick tells Intuition which Bitplanes of the display to be affected by the Image. For every "picked" plane, the next successive Bitplane of the image is printed there. For example:

If we set PlanePick to 1, Bitplane zero would be affected. If we then printed the arrow Image, we would get the arrow drawn with colour 1, and the background drawn with colour 0.

If we had instead set PlanePick to 2, Bitplane one would have been affected, which would result in that the arrow would have been drawn with colour 2, and the background still drawn with colour 0.

If our Image would consist of several Bitplanes, the lowest picked plane would be affected by the lowest Bitplane, and so on. If we take our face Image for example, and we want to draw it with colour 3, 2 1 and 0, we want to affect Bitplane zero and BitPlane one of the display. We would therefore need to set PlanePick to 3, see table:

Bitplane to affect	PlanePick
No planes affected	0000 = 0
Plane 0	0001 = 1

```
Plane 1              0010 = 2
Plane 1 and 0        0011 = 3
Plane 2              0100 = 4
Plane 2 and 0        0101 = 5
Plane 2 and 1        0110 = 6
Plane 2, 1 and 0     0111 = 7
Plane 3              1000 = 8
Plane 3 and 0        1001 = 9
Plane 3 and 1        1010 = A
and so on...
```

If we want to draw it with colour 5, 4, 1 and 0, we want to affect Bitplane two and zero:

```
(000=colour 0, 001=colour 1, 100=colour 4, 101=colour 5)
 ^ ^           ^ ^           ^ ^           ^ ^
 210           210           210           210 (BitPlanes)
```

We would then set PlanePick to 5 (0101).

3.5.4 PLANEONOFF

PlaneOnOff decides what should happen to the Bitplanes of the displaying element that were not affected by PlanePick. Should these Bitplanes be filled with 1's or 0's.

For example, if we want to print the little arrow in colour 5 and the background in colour 1 the pixels should have the values 0001 (colour 1) and 0101 (colour 5). We want the Image to affect Bitplane two, so PlanePick is set to 4 (0100). But we also want Bitplane zero to be filled with 1's, so PlaneOnOff is set to 1 (0001).

With help of PlanePick and PlaneOnOff you can even print filled rectangles without any Image data. You only need to set Width and Height as desired, and then set Depth to 0 (no Bitplanes), PlanePick to 0000 (no Bitplanes should be used) and PlaneOnOff to the colour you want.

3.5.5 HOW TO USE THE IMAGE STRUCTURE

The Image structure is either used to print images in a screen or window, but can also be used to print images connected to a Gadget, Menu or Requester. Same as Intuition's other Graphics structures.

When you want to print an image in a window/screen you simply call the function DrawImage():

Synopsis: DrawImage(rast_port, image, x, y);

rast_port: (struct RastPort *) Pointer to a RastPort.

If the images should be drawn in a window, and my_window is a pointer to that window, you write: my_window->RPort.

If the images should be drawn in a Screen, and my_screen is a pointer to that screen, you write: my_screen->RastPort.

image: (struct Image *) Pointer to an Image structure which has been initialized with your requirements.

x: (long) Number of pixels added to the x position of the image.

y: (long) Number of lines added to the y position of the image.

A call to print an image using my_image structure would look something like this:

```
DrawImage( my_window->RPort, &my_image, 0, 0 );
```

3.6 FUNCTIONS

Here are the three functions you use when you want to draw lines/characters/images into a RastPort (Screen/Window):

DrawBorder()

This function draws the specified Borders into a RastPort (Screen/Window).

Synopsis: DrawBorder(rast_port, border, x, y);

rast_port: (struct RastPort *) Pointer to a RastPort.

If the lines should be drawn in a window, and my_window is a pointer to that window, you write: my_window->RPort.

If the lines should be drawn in a Screen, and my_screen is a pointer to that screen, you write: my_screen->RastPort.

border: (struct Border *) Pointer to a Border structure which has been initialized with your requirements.

x: (long) Number of pixels added to the x coordinates.

y: (long) Number of lines added to the y coordinates.

PrintIText()

This function prints text into a RastPort (Screen/Window).

Synopsis: PrintIText(rast_port, intui_text, x, y);

rast_port: (struct RastPort *) Pointer to a RastPort.

If the text should be printed in a window, and my_window is a pointer to that window, you write: my_window->RPort.

If the text should be printed in a Screen, and my_screen is a pointer to that screen, you write: my_screen->RastPort.

intui_text: (struct IntuiText *) Pointer to a IntuiText structure which has been initialized with your requirements.

x: (long) Number of pixels added to the x position of the characters.

y: (long) Number of lines added to the y position of the characters.

DrawImage()

This function draws the specified images into a RastPort (Screen/Window).

Synopsis: DrawImage(rast_port, image, x, y);

rast_port: (struct RastPort *) Pointer to a RastPort.

If the images should be drawn in a window, and my_window is a pointer to that window, you write: my_window->RPort.

If the images should be drawn in a Screen, and my_screen is a pointer to that screen, you write: my_screen->RastPort.

image: (struct Image *) Pointer to an Image structure which has been initialized with your requirements.

x: (long) Number of pixels added to the x position of the image.

y: (long) Number of lines added to the y position of the image.

3.7 EXAMPLES

Here are some examples on how to draw lines (Border), print text (IntuiText) and draw pictures (Image).

The printed version of these examples can be found in Appendix A (Part IV: Appendices).

Example1

This program will open a normal window which is connected to the Workbench Screen. We will then draw a strange line with help of Intuition's Border structure.

Example2

This program will open a normal window which is connected to the Workbench Screen. We will then draw two rectangles with different colours. This shows how you can link Border structures to each other in order to get the desired effects.

Example3

This program will open a normal window which is connected to the Workbench Screen. We will then print a text string with help of Intuition's IntuiText structure.

Example4

Same as Example3 except that the text will be printed with underlined italic characters.

Example5

This program will open a normal window which is connected to the Workbench Screen. We will then draw the little nice arrow we talked so much about.

Example6

Same as Example5 except that we will draw it several times in different colours. This shows how PlanePick/PlaneOnOff works.

Example8

This program will open a normal window which is connected to a 16-colour Custom screen. In the window we will draw the famous AMIGA-logo.

4 GADGETS

4.1 INTRODUCTION

In this chapter we will look at how the user can communicate with the program. All programs which uses Intuition should, if possible, be controlled by a mouse, since it is the most commonly used input device. Intuition's Gadgets will play a big role here.

A gadget can be a "button" which the user can click on, but it can also be a small knob which can be dragged (like a volume control on a radio). A gadget can even be a box where the user can enter a text string or a value. The advantages of using gadgets are almost uncountable. They are very well supported by Intuition which means that your program does hardly need to do anything, but still have an outstanding user interface which is both easy to understand as well as use.

4.2 DIFFERENT TYPES OF GADGETS

There exist two types of gadgets: System gadgets, which we already have discussed in chapter 2.3 SYSTEM GADGETS, and Custom gadgets. For every window System gadgets always look the same, and are always placed in the same places. Custom Gadgets, however, can be placed wherever you want, and it is you who decide how they should look like.

4.3 CUSTOM GADGETS

There exist four different types of Custom gadgets:

- Boolean gadget	On/Off (True/False) button.
- Proportional gadget	A small knob which can be moved around inside a container.
- String gadgets	Gadget which enables the user to enter a string.
- Integer gadget	Same as the String gadget, except that the user can only enter integer numbers.

4.3.1 GRAPHICS FOR CUSTOM GADGETS

You can render the gadget with help of the high-level graphics utilities which are supported by Intuition (See chapter 3 GRAPHICS for more information). You can render the gadget with help of a Border structure, or an Image structure. You can even have a different rendering when the gadget is selected, highlighted.

It is of course possible to open gadgets with no rendering at all. The Drag gadget (System gadget) is a good example.

4.3.2 POSITION

You can position the gadget everywhere on the display. The position is normally relative to the top left corner of the displaying element (Window, Requester etc). If you want you can position the gadget relative to some other sides than the top left corner:

If you want the gadget to be placed 10 pixels out and 20 lines down from the top left corner of a window, you set LeftEdge to 10, and TopEdge to 20. (LeftEdge etc are elements in the Gadget structure which is described later in the chapter.)

If you on the other hand want the gadget to always be 20 pixels above the bottom border of the window, and 10 pixels to the left of the right border, you set LeftEdge to -10 and TopEdge to -20 together with the Flags GRELRIGHT and GRELBOTTOM. (More about this later.)

4.3.3 SIZE

You decide the width and height of the gadget (the select box of the gadget) by setting the Width and Height variables as desired.

If you set the Height to 25, the gadget will be 25 lines high.

If you on the other hand set the Height to -25 and you set the GRELHEIGHT flag, the gadget will always be 25 lines smaller than the containing element (window etc).

Same applies for the Width. If you set the Width to 50, the gadget will always be 50 pixels wide.

If you on the other hand set the Width to be -50 and you set the flag GRELWIDTH, the gadget will always be 50 pixels smaller than the containing element.

4.4 INITIALIZE A CUSTOM GADGET

When you want to use a Custom gadget you need to declare and initialize a Gadget structure which look like this:

```
struct Gadget
{
  struct Gadget *NextGadget;
  SHORT LeftEdge, TopEdge, Width, Height;
  USHORT Flags;
  USHORT Activation;
  USHORT GadgetType;
  APTR GadgetRender;
  APTR SelectRender;
  struct IntuiText *GadgetText;
  LONG MutualExclude;
  APTR SpecialInfo;
  USHORT GadgetID;
  APTR UserData;
}
```

NextGadget: A pointer to the next gadget in the list if there exist one, else NULL.

LeftEdge, TopEdge: Position of the gadget's select box relative to the displaying element.

Width, Height: Width and height of the gadget's select box.

Flags: You must set one of the following four highlighting flags: (The gadget is highlighted when it is selected)

- GADGHCOMP: Complement the colours of all pixels in the gadget's select box.
- GADGHBOX: Draw a box around the the gadget's select box.
- GADGHIMAGE: Display an alternative Image/ Border.
- GADGHNONE: No highlighting.

If the gadget should be rendered as an Image set the flag GADGIMAGE, otherwise (render it with a Border structure, or no rendering at all) clear this flag.

If you want the position and/or size of the gadget to be relative to the size of the displaying element, set the desired flags:

GRELBOTTOM: TopEdge is used as an offset relative to the bottom of the

displaying element, instead of an offset relative to the top of the displaying element.

GRELRIGHT: LeftEdge is used as an offset relative to the right edge of the displaying element, instead of an offset relative to the left edge of the displaying element.

GRELWIDTH: Width describes an increment to the width of the displaying element, instead of describing the absolute width of the gadget.

GRELRIGHT: Height describes an increment to the height of the displaying element, instead of describing the absolute height of the gadget.

If this gadget is a toggle-select gadget (see Activation flags) you can set the SELECTED flag, and the gadget will be selected and highlighted when opened. If not, the gadget will be unselected and non-highlighted when opened.

You can also examine this field to see if the SELECTED flag is set or not. If it is set, the gadget is selected, else it is unselected.

If you want that the gadget should be disabled when opened, you set the flag GADDISABLED. Your program can later change this by calling the functions OnGadget() (enables the gadget) and OffGadget() (disables the gadget).

Activation: Set the flags for the desired effects: (More about these later...)

GADGIMMEDIATE: If you want your program to know immediately when the user selects this gadget, you should set this flag.

RELVERIFY: If you want your program to receive a message when the user releases the gadget and

the pointer is still inside the gadget's select box, you should set this flag.

FOLLOWMOUSE: Set this flag if you want your program to receive mouse positions every time the user moves the mouse while this gadget is selected.

TOGGLESELECT: Each time the user selects this gadget, the on/off state of the gadget (as well as the image) is toggled. Your program can later check the status of the gadget by examining the SELECTED bit in the Flags field.

BOOLEXTEND: Set this flag if your gadget has a BoolInfo structure connected to it.

If your gadget is connected to a window you can set the following flags in order to change the size of the window's borders. (You can then put the gadget there.):

RIGHTBORDER: The width of the window's right border is calculated with help of the gadget's position and width.

LEFTBORDER: The width of the window's left border is calculated with help of the gadget's position and width.

TOPBORDER: The height of the window's top border is calculated with help of the gadget's position and height.

BOTTOMBORDER: The height of the window's bottom border is calculated with help of the gadget's position and height.

If this gadget is connected to a requester you can set the ENDGADGET flag. The requester will only go away when a gadget with the ENDGADGET flag has been selected. (See chapter 5 REQUESTERS for more information about gadgets connected to requesters.)

Intuition will only care about these flags if the gadget is a String/Integer gadget:

STRINGRIGHT: Set this flag if you want the characters in the string to be right-justified.

STRINGCENTER: Set this flag if you want the characters in the string to be center-justified.

(The default is left-justified.)

LONGINT: Set this flag if you want that the user should only be able to enter a 32-bit signed integer value.

ALTKEYMAP: Set this flag if you want to use an alternative keymap. Remember to give the AltKeyMap pointer in the StringInfo structure a pointer to the keymap.

GadgetType: You must set one of the following three flags:

BOOLGADGET: Set this flag if you want a Boolean gadget.

STRGADGET: Set this flag if you want a String/Integer gadget. If you want an Integer gadget you also need to set the Activation flag LONGINT.

PROPGADGET: Set this flag if you want a Proportional gadget.

The following two flags are only for gadgets connected to Gimmezerozero windows and requesters.

GZZGADGET: If the gadget is connected to a Gimmezerozero window and you have set this flag, the gadget will be put in the outer window, and will not destroy any drawings etc in the inner window.

REQGADGET: Set this flag if the gadget is connected to a requester.

GadgetRender: A pointer to an Image or Border structure (See chapter 3 GRAPHICS for more information) which will be used to render the gadget. (If you supply a pointer to an Image structure you need to set the Flags variable to GADGIMAGE.) Set it to NULL if you do not want to supply the gadget with any graphics.

SelectRender: A pointer to an alternative Image or Border structure which will be used when the gadget is highlighted. Remember to set the GADGHIMAGE bit in the Flags variable if you want to use an alternative Image/Border. (GadgetRender and SelectRender must point to the same type of data. If you have specified that you want to use an Image for the GadgetRender (GADGIMAGE), SelectRender must then also point to an Image structure.)

GadgetText: A pointer to an IntuiText structure (See chapter 3 GRAPHICS for more information) which will be used to print some text in the gadget. Set it to NULL if you do not want any text connected to your gadget.

MutualExclude: This field represent the first 32 gadgets in the list. If this gadget is selected, all specified gadgets in the MutualExclude field are deselected automatically. For example, if you want that gadget number 0, 2, 5 and 8 should be deselected (mutual excluded) when this gadget is selected, the MutualExclude field should be set to 293. (293(d) == 100100101(b))

Remember, the mutual exclude works only with toggle-select gadgets.

SpecialInfo: If the gadget is a Proportional gadget you should here give Intuition a pointer to a PropInfo structure, or if the gadget is a String (Integer) gadget you should give Intuition a pointer to a StringInfo structure.

If the gadget is a Boolean gadget you can connect a BoolInfo structure which will place a mask on the gadget's select box. (See below for more information.)

GadgetID This variable is left for your own use. Intuition ignores this field.

UserData: A pointer to any structure you may want to connect to the gadget. Intuition ignores this field.

4.5 BOOLEAN GADGET

If you want a Boolean gadget you should declare and initialize the Gadget structure something like this:

```
struct my_gadget=
{
  NULL,          /* NextGadget, no more gadgets in the list. */
  40,            /* LeftEdge, 40 pixels out. */
  20,            /* TopEdge, 20 lines down. */
  60,            /* Width, 60 pixels wide. */
  20,            /* Height, 20 pixels lines high. */
  GADGHCOMP,     /* Flags, when this gadget is highlighted, */
                 /* the gadget will be rendered in the      */
                 /* complement colours:                     */
                 /* (Colour 0 (00) will become colour 3 (11) */
                 /* (Colour 1 (01)        - " -       2 (10) */
                 /* (Colour 2 (10)        - " -       1 (01) */
                 /* (Colour 3 (11)        - " -       0 (00) */
  GADGIMMEDIATE| /* Activation, our program will receive a  */
  RELVERIFY,     /* message when the user has selected this */
                 /* gadget, and when the user has released  */
                 /* it.                                     */
  BOOLGADGET,    /* GadgetType, a Boolean gadget. */
  &my_border,    /* GadgetRender, a pointer to our Border */
                 /* structure.                            */
  NULL,          /* SelectRender, NULL since we do not    */
                 /* supply the gadget with an alternative */
                 /* image. (We complement the colours     */
                 /* instead.)                             */
  &my_text,      /* GadgetText, a pointer to our IntuiText */
                 /* structure.                              */
  NULL,          /* MutualExclude, no mutual exclude. */
  NULL,          /* SpecialInfo, no BoolInfo connected to it. */
  0,             /* GadgetID, no id. */
  NULL           /* UserData, no user data connected to the */
                 /* gadget. */
};
```

It is possible to connect a mask to a Boolean gadget. In that case the gadget would only be selected when the user clicks inside the selected (masked) area, and only that area would be highlighted. If you want to connect a mask to a Boolean gadget you need to declare and initialize a BoolInfo structure together with its mask.

The BoolInfo structure look like this:

```
struct BoolInfo
```

```
{
  USHORT Flags;
  UWORD *Mask;
  ULONG Reserved;
};
```

Flags: There exist only one flag for the moment: BOOLMASK.

Mask: Pointer to the binary mask. (The width and height of the mask must be the same as the width and height of the select box.)

Reserved: This field is reserved and should therefore be set to 0.

The binary mask is built up as an Image plane. Only the selected (1's) parts of the mask will be sensitive and highlighted.

A mask for a gadget with the width of 16 pixels, and the height of 8 pixels can look something like this: (Only the inner part of the select box will be sensitive and highlighted.)

Mask	16-Bit memory words	Hexadecimal
0000001111000000	0000 0011 1100 0000	03C0
0000111111110000	0000 1111 1111 0000	0FF0
0011111111111100	0011 1111 1111 1100	3FFC
1111111111111111	1111 1111 1111 1111	FFFF
1111111111111111	1111 1111 1111 1111	FFFF
0011111111111100	0011 1111 1111 1100	3FFC
0000111111110000	0000 1111 1111 0000	0FF0
0000001111000000	0000 0011 1100 0000	03C0

The mask would in this case be declared/initialized like this:

```
UWORD my_mask[]=
{
  0x03C0,
  0x0FF0,
  0x3FFC,
  0xFFFF,
  0xFFFF,
  0x3FFC,
  0x0FF0,
  0x03C0
};
```

See Example6 for more information about the BoolInfo structure.

4.6 STRING/INTEGER GADGET

String and Integer gadgets are a bit more complicated to declare since you need to supply the Gadget structure with a StringInfo structure. However, String and Integer gadgets allows the user to enter a string or an integer value without much effort from your side. Intuition takes care of most of the work, and you almost only need to decide how long and where the string gadget should be.

4.6.1 STRINGINFO STRUCTURE

The StringInfo structure look like this:

```
struct StringInfo
{
  UBYTE *Buffer;
  UBYTE *UndoBuffer;
  SHORT BufferPosition;
  SHORT MaxChars;
  SHORT DispPos;
  SHORT UndoPos;
  SHORT NumChars;
  SHORT DispCount;
  SHORT CLeft, CTop;
  struct Layer *LayerPtr;
  LONG LongInt;
  struct KeyMap *AltKeyMap;
};
```

Buffer: A pointer to a NULL-terminated string.

UndoBuffer: A pointer to a NULL-terminated string which is used by Intuition to store an undo string. This must be at least as long as the Buffer string. When the user selects this gadget Intuition makes a copy of the Buffer string which will be copied back if the user presses AMIGA + Q. Since only one String/ Integer gadget can be active at a time several String/Integer gadgets can use the same undo string.

MaxChars: The maximum number of characters which may be entered. (Number of characters in the buffer + the NULL '\0' sign.)

BufferPos: Cursor position in the buffer string.

DispPos: Position of the first character which is displayed.

These variables are initialized and maintained by Intuition:

UndoPos:	Cursor position in the undo string.
NumChars:	Current number of characters in the buffer.
DispCount:	Current number of visible characters in the container.
CLeft, CTop:	Top left offset of the container.
LayerPtr:	Pointer to the Layer structures.
LongInt:	If this is an Integer gadget you can examine the value here to find out what the user has entered.
AltKeyMap:	A pointer to an alternative keymap. (Remember to set the flag ALTKEYMAP in the Activation field.)

4.6.2 INITIALIZE A STRING/INTEGER GADGET

This is an example on how you can initialize a string gadget:

```
UBYTE my_buffer[50];
UBYTE my_undo_buffer[50];

struct StringInfo my_string_info=
{
  my_buffer,       /* Buffer, pointer to a NULL-terminated s. */
  my_undo_buffer,  /* UndoBuffer, pointer to a NULL-          */
                   /* terminated string. (Remember my_buffer  */
                   /* is equal to &my_buffer[0])              */
  0,               /* BufferPos, initial position of the */
                   /* cursor.                            */
  50,              /* MaxChars, 49 characters + NULL-sign. */
  0,               /* DispPos, first character in the string */
                   /* should be first character in the       */
                   /* display.                               */

  /* Intuition initializes and maintains these variables: */

  0,               /* UndoPos */
  0,               /* NumChars */
  0,               /* DispCount */
  0, 0,            /* CLeft, CTop */
  NULL,            /* LayerPtr */
  NULL,            /* LongInt */
  NULL,            /* AltKeyMap */
};
```

```
struct Gadget my_gadget=
{
  NULL,           /* NextGadget, no more gadgets in the list. */
  68,             /* LeftEdge, 68 pixels out. */
  30,             /* TopEdge, 30 lines down. */
  198,            /* Width, 198 pixels wide. */
  8,              /* Height, 8 pixels lines high. */
  GADGHCOMP,      /* Flags, draw the select box in the        */
                  /* complement colours. Note: it is actually */
                  /* only the cursor which will be drawn in   */
                  /* the complement colours (yellow). If you  */
                  /* set the flag GADGHNONE the cursor will   */
                  /* not be highlighted, and the user will    */
                  /* therefore not be able to see it.         */
  GADGIMMEDIATE|  /* Activation, our program will receive a  */
  RELVERIFY,      /* message when the user has selected this */
                  /* gadget, and when the user has released  */
                  /* it.                                      */
  STRGADGET,      /* GadgetType, a String gadget. */
  &my_border,     /* GadgetRender, a pointer to our Border */
                  /* structure.                           */
  NULL,           /* SelectRender, NULL since we do not    */
                  /* supply the gadget with an alternative */
                  /* image.                               */
  &my_text,       /* GadgetText, a pointer to our IntuiText */
                  /* structure.                              */
  NULL,           /* MutualExclude, no mutual exclude. */
  &my_string_info,/* SpecialInfo, a pointer to a StringInfo */
                  /* structure. */
  0,              /* GadgetID, no id. */
  NULL            /* UserData, no user data connected to the */
                  /* gadget.                                  */
};
```

The only difference between declaring and initializing a String gadget and an Integer gadget, is that when you initialize an Integer gadget you also need to:

1. Set the flag LONGINT in the Activation field.
2. Copy an integer string into the buffer string.
   eg: strcpy(my_buffer, "0");

4.6.3 USING A STRING/INTEGER GADGET

Once you have declared and initialized the appropriate structures Intuition takes care of everything else. While the user is entering a string he/she can even use some special keys:

<-	Moves the cursor to the left.
->	Moves the cursor to the right.

SHIFT and <-	Moves the cursor to the beginning of the string.
SHIFT and ->	Moves the cursor to the end of the string.
BACKSPACE	Deletes the character to the left of the cursor.
DEL	Deletes the character under the cursor.
AMIGA and Q	Undo the last changes of the string.
AMIGA and X	Clears the buffer string.
RETURN	Releases the gadget. If we have set the activation flag RELVERIFY we will receive a message telling us that the user has finished.

4.7 PROPORTIONAL GADGET

A proportional gadget is roughly a knob which can be moved horizontally, vertically or both inside a container. It can be like a volume control on a radio, or it can be used to show the user how much more data there exist in the file etc.

4.7.1 PROPINFO STRUCTURE

The PropInfo structure look like this:

```
struct PropInfo
{
  USHORT Flags;
  USHORT HorizPot;
  USHORT VertPot;
  USHORT HorizBody;
  USHORT VertBody;
  USHORT CWidth;
  USHORT CHeight;
  USHORT HPotRes, VPotRes;
  USHORT LeftBorder;
  USHORT TopBorder;
};
```

Flags: You normally should set one or both of the following two bits:

FREEHORIZ	Set this bit if you want the user to be able to move the knob horizontally.
FREEVERT	Set this bit if you want the user to be able to move the knob vertically.
AUTOKNOB	Set this bit if you want that the size of the knob to be

controlled by Intuition. (HorizBody and VertBody affects the size of the Autoknob.)

- If you want to use Intuition's Autoknob you should give GadgetRender a pointer to an Image structure. (You do not need to initialize the Image structure since Intuition takes care of it.)

- If you on the other hand would like to use your own knob image, you give GadgetRender a pointer to your Image structure, which you have initialized yourself.

PROPBORDERLESS Set this bit if you do not want any border around the container.

KNOBHIT This is a flag which is set by Intuition if this gadget is selected.

HorizPot: This variable contains the actual (horizontally) proportional value. If the user has moved the knob 25% to the right, HorizPot is 25% of MAXPOT (0xFFFF). (0xFFFF * 0.25 = 0x3FFF)

VertPot: Same as HorizPot except that this is the vertically proportional value.

HorizBody: Describes how much HorizPot should change every time the user clicks inside the container. If the volume of a melody can be between 0-63 (64 steps), HorizPot should change 1/64 each time. The HorizBody should therefore be initialized to:
1/64 * MAXBODY (0xFFFF) == 3FF

HorizBody describes also how much the user can see/use of the entire data. For example, if you have a list of 32 file names, and the user only can see 8 names at one time (25%), the knob (AUTOKNOB) should fill 25% of the container. HorizBody should in this case be initialized to:
MAXBODY * 8 / 32 (25% of 0xFFFF) == 3FFFF

VertBody: Same as HorizBody except that it affects VertPot, and the vertical size of the knob (AUTOKNOB).

These variables are initialized and maintained by Intuition:

CWidth: Width of the container.

CHeight: Height of the container.

HPotRes, VPotRes: Pot increments.

LeftBorder: Position of the container's left border.

TopBorder: Position of the container's top border.

4.7.2 INITIALIZE A PROPORTIONAL GADGET

This is an example on how you can initialize a proportional gadget which can, for example, be used to change the volume of a melody (64 positions):

```
/* We need to declare an Image structure for the knob, but */
/* since Intuition will take care of the size etc of the   */
/* knob, we do not need to initialize the Image structure: */
struct Image my_image;

struct PropInfo my_prop_info=
{
  FREEHORIZ|      /* Flags, the knob should be moved         */
  AUTOKNOB,       /* horizontally, and Intuition should take */
                  /* care of the knob image.                 */
  0,              /* HorizPot, start position of the knob. */
  0,              /* VertPot, 0 since we will not move the */
                  /* knob vertically. */
  MAXBODY * 1/64, /* HorizBody, 64 steps. */
  0,              /* VertBody, 0 since we will not move the */
                  /* knob vertically. */

  /* These variables are initialized and maintained by */
  /* Intuition:                                        */

  0,              /* CWidth */
  0,              /* CHeight */
  0, 0,           /* HPotRes, VPotRes */
  0,              /* LeftBorder */
  0               /* TopBorder */
};

struct Gadget my_gadget=
{
  NULL,            /* NextGadget, no more gadgets. */
  80,              /* LeftEdge, 80 pixels out. */
  30,              /* TopEdge, 30 lines down. */
  200,             /* Width, 200 pixels wide. */
  12,              /* Height, 12 pixels lines high. */
```

```
  GADGHCOMP,        /* Flags, no highlighting. */
  GADGIMMEDIATE|    /* Activation, our program will receive a  */
  RELVERIFY,        /* message when the user has selected this */
                    /* gadget, and when the user has released  */
                    /* it.                                     */
  PROPGADGET,       /* GadgetType, a Proportional gadget. */
  &my_image,        /* GadgetRender, a pointer to our Image */
                    /* structure. (Intuition will take care */
                    /* of the knob image. See chapter 3     */
                    /* GRAPHICS for more information about  */
                    /* images.)                             */
  NULL,             /* SelectRender, NULL since we do not    */
                    /* supply the gadget with an alternative */
                    /* image.                                */
  &my_text,         /* GadgetText, pointer to a IntuiText */
                    /* structure. */
  NULL,             /* MutualExclude, no mutual exclude. */
  &my_prop_info,    /* SpecialInfo, pointer to a PropInfo */
                    /* structure. */
  0,                /* GadgetID, no id. */
  NULL              /* UserData, no user data connected to */
                    /* the gadget. */
};
```

4.8 MONITORING THE GADGETS

Once you have decided which gadgets to use and how they should look like, it is time to decide what information they should send to your program. You need to decide if they should send a message when the user has selected them, or when the user has released them etc. All this work with handling the input can be easily done with help of Intuition's IDCMP system. IDCMP stands for Intuition's Direct Communications Message Ports system. Hard name but very easy use.

The IDCMP system is also explained, in more detail, in chapter 8 IDCMP.

If you want to use the IDCMP system you only need to follow these steps:

1. Decide what events your gadget should report. You do it by setting the appropriate flags in the Activation field in the Gadget structure:

   GADGIMMEDIATE Set this flag if you want your program to receive a message immediately when the user selects this gadget.

   RELVERIFY Set this flag if you want your program to receive a message when the user releases (while still pointing at it) this gadget. If

the user releases the gadget after having moved away the pointer from the select box, your program will not receive any message.

FOLLOWMOUSE — Set this flag if you want your program to receive a message every time the mouse is moved while this gadget is selected.

2. If the gadgets are connected to a window you need to tell Intuition which messages should be allowed to pass by. You do it by setting the appropriate flags in the IDCMPFlags field in the NewWindow structure:

   GADGETDOWN — If a gadget connected to a window has the GADGIMMEDIATE flag set, you should set the GADGETDOWN flag.

   GADGETUP — If a gadget connected to a window has the RELVERIFY flag set, you should set the GADGETUP flag.

   MOUSEMOVE — If a gadget connected to a window has the FOLLOWMOUSE flag set, you should set the MOUSEMOVE flag.

   CLOSEWINDOW — If you have connected the Close window gadget (System gadget) to your window, you can set this bit, and your program will receive a message when the user selects this gadget. Remember, Intuition does not close the window automatically when the user clicks on the Close window gadget. It is up to your program to decide what to do when you receive a CLOSEWINDOW event. (See Example1)

3. Once your program is running it is time to try to collect and examine the messages sent to us by Intuition. One very commonly used way of doing it is to put the program to sleep [Wait()] and it will wake first when a message has arrived. We will then try to collect it [GetMsg()], and then (if success) examine the message. When we are finished with it we send it back [ReplyMsg()] so Intuition can send us another message if there is one. We can then put the program to sleep again, and so on.

When we collect a message with help of the function GetMsg() we actually receive a pointer to an IntuiMessage structure or NULL if there was nothing for us. The IntuiMessage structure look like this:

```
struct IntuiMessage
{
  struct Message ExecMessage;
```

```
  ULONG Class;
  USHORT Code;
  USHORT Qualifier;
  APTR IAddress;
  SHORT MouseX, MouseY;
  ULONG Seconds, Micros;
  struct Window *IDCMPWindow;
  struct IntuiMessage *SpecialLink;
};
```

This structure is fully explained in chapter 8 IDCMP, so you do not need to bother so much about it for the moment. However, there is two variables in the structure that we need to understand. They are:

Class: When a message is sent this field contains the reason for why it was sent. It contains an IDCMP flag which tells us what has happened. For example, if the user has selected a gadget with the GADGIMMEDIATE flag set, this variable is equal to GADGETDOWN.

IAddress: This is a pointer to the gadget (or similar) which sent the message. For example, if a program receives a message telling us that a gadget was selected, we need to know which gadget since there may be several gadget connected to the same window. This pointer points to that gadget.

Here is an example of how a program which collects IDCMP messages can look like:

```
main()
{
  /* Declare a variable in which we will store the IDCMP */
  /* flag: */
  ULONG class;

  /* Declare a pointer in which we will store the address */
  /* of the object (gadget) which sent the message: */
  APTR address;

  /* Declare a pointer to an IntuiMessage structure: */
  struct IntuiMessage *my_message;


  /* ... */


  /* (This is an endless loop) */
  while( TRUE )



  {
    /* 1. Put our program to sleep, and wake up first when   */
    /*    we have received a message. Do not bother for the  */
```

```
/*    moment about all these funny thing we put inside   */
/*    the Wait () function. I will talk about that later */
/*    on.                                                */

/* Wait until we have received a message: */
Wait( 1 << my_window->UserPort->mp_SigBit );



/* 2. We have now received a message and we shall now     */
/*    try to collect it. If success the function GetMsg() */
/*    will return a pointer to an IntuiMessage            */
/*    structure, otherwise it will return NULL.           */

/* Collect the message: */
my_message = GetMsg( my_window->UserPort );



/* 3. If we have collected a message successfully we save */
/*    some important values which we later can use.       */

if( my_message )
{
  /* Save the code variable: */
  class = my_message->Class;

  /* Save the address of the gadget: */
  address = my_message->IAddress;


  /* 4. After we have saved all important values we     */
  /*    reply as fast as possible. Once we have replied */
  /*    we can NOT use the IntuiMessage structure any   */
  /*    more!                                           */

  ReplyMsg( my_message );


  /* 5. We can now check what message was sent to us. */

  /* Check which IDCMP flag was sent: */
  switch( class )
  {
    case GADGETDOWN: /* The user has selected a gadget: */

           /* If we want to check which gadget sent the */
           /* message we simply need to check the       */
           /* address pointer:                          */

           if( address == &my_first_gadget)
             /* The gadget "my_first_gadget" selected. */
             /* Do what ever you want... */

           if( address == &my_second_gadget)
             /* The gadget "my_second_gadget" selected. */
             /* Do what ever you want... */
```

```
                break;

        case GADGETUP: /* The user has released a gadget: */
                /* Do what ever you want... */
                break;

        case MOUSEMOVE: /* The user has moved the mouse */
                        /* while a gadget was selected. */
                /* Do what ever you want... */
                break;

        case CLOSEWINDOW: /* The user has selected the close */
                          /* window gadget. Time to quit.    */
                /* Do what ever you want... */
                break;
      }
    }
  }


  /* ... */
}
```

4.9 FUNCTIONS

Here are some commonly used functions:

RefreshGadgets()

This function redraws all the gadgets in the list, starting by the specified gadget. If you for example has added or deleted a gadget you need to call this function to see the changes. On the other hand, if you have changed the imagery of a gadget, or the gadget's image has been trashed by something, you can also use this function to refresh the display.

Synopsis: RefreshGadgets(gadget, window, requester);

gadget: (struct Gadget *) Pointer to the gadget where the redrawing should start. This gadget, and all the following gadgets in the list will be redrawn.

window: (struct Window *) Pointer to the window which the gadgets are connected to.

requester: (struct Requester *) If the gadget is connected to a requester, set this pointer to point to that requester, else NULL. Important, if this gadget is connected to a requester, it must be displayed when you execute this command! (See chapter 5 REQUESTERS for more information about requesters.)

AddGadget()

This function adds a gadget to the gadget list.

Synopsis: result = AddGadget(window, gadget, position);

result: (long) The actual position of the gadget when it has been added.

window: (struct Window *) Pointer to the window, to which the gadget should be added.

gadget: (struct Gadget *) Pointer to the gadget which will be added.

position: (long) Position in the gadget list. (Starts from zero). Eg:
0 -> Before all other gadgets.
1 -> After the first gadget, but before the second.
If a too big value is entered (or -1), the gadget will be placed last in the list.

Important, after your program has added the necessary gadgets, you need to call the function RefreshGadgets() in order to see your changes. You may add (or take away) several gadgets, but when you are finished you must call that function.

RemoveGadget()

This function removes a gadget from the list:

Synopsis: result = RemoveGadget(window, gadget);

result: (long) The position of the removed gadget or -1 if something went wrong.

window: (struct Window *) Pointer to the window that the gadget is connected to.

gadget: (struct Gadget *) Pointer to the gadget which will be removed.

Important, after your program has removed the necessary gadgets, you need to call the function RefreshGadgets() in order to see your changes. You may take away (or add) several gadgets, but when you are finished you must call that function.

OnGadget()

This function enables a gadget (removes the GADGDISABLED bit in the gadget structure's Flags field):

Synopsis: OnGadget(gadget, window, requester);

gadget: (struct Gadget *) Pointer to the gadget which will be enabled.

window: (struct Window *) Pointer to the window that the gadget is attached to.

requester: (struct Requester *) If the gadget is connected to a requester, set this pointer to point to that requester, else NULL. Important, if this gadget is connected to a requester, it must be displayed when you execute this command!

Remember, as long as the gadget is disabled the user can not select it, and it will not broadcast any messages. A disabled gadget is drawn as usual except that it "ghosted".

OffGadget()

This function disables a gadget (sets the GADGDISABLED bit in the gadget structure's Flags field):

Synopsis: OffGadget(gadget, window, requester);

gadget: (struct Gadget *) Pointer to the gadget which will be disabled.

window: (struct Window *) Pointer to the window that the gadget is attached to.

requester: (struct Requester *) If the gadget is connected to a requester, set this pointer to point to that requester, else NULL. Important, if this gadget is connected to a requester, it must be displayed when you execute this command!

ModifyProp()

This function modifies a proportional gadget's values and knob. For example, if your program is reading files from the disk, VertBody was maybe equal to 0xFFFF (MAXBODY) in the beginning, but as more files are collected from the disk, you maybe want to change the size of the knob etc. You then simply call this function and it will change the values as well as redraw the gadget.

Synopsis: ModifyProp(gadget, window, requester, flags, horiz_pot, vert_pot, horiz_body, vert_body);

gadget: (struct Gadget *) Pointer to the proportional gadget which should be changed and redrawn.

window: (struct Window *) Pointer to the window which the proportional gadget is connected to.

requester: (struct Requester *) If the gadget is connected to a requester, set this pointer to point to that requester, else NULL. Important, if this gadget is connected to a requester, it must be displayed when you execute this command!

flags: (long) Here is the list of all flags you may use:

FREEHORIZ	Set this bit if you want the user to be able to move the knob horizontally.
FREEVERT	Set this bit if you want the user to be able to move the knob vertically.
AUTOKNOB	Set this bit if you want that the size of the knob to be controlled by Intuition. (HorizBody and VertBody affects the size of the Autoknob.) - If you want to use Intuition's Autoknob you should give GadgetRender a pointer to an Image structure. (You do not need to initialize the Image structure since Intuition takes care of it.) - If you on the other hand would like to use your own knob image, you give GadgetRender a pointer to your Image structure, which you have initialized yourself.
PROPBORDERLESS	Set this bit if you do not want any border around the container.

(See chapter 4.7 for more information.)

horiz_pot: (long) This variable contains the actual (horizontally) proportional value. If the knob should be moved 25% to the right, HorizPot should be set to 25% of MAXPOT (0xFFFF). (0xFFFF * 0.25 = 0x3FFF)

vert_pot: (long) Same as HorizPot except that this is the vertically proportional value.

horiz_body: (long) Describes how much HorizPot should change every time the user clicks inside the container. If the volume of a melody can be between 0-63 (64 steps), HorizPot should change 1/64 each time. The HorizBody should therefore be set to: 1/64 * MAXBODY (0xFFFF) == 3FF

HorizBody describes also how much the user can see/use of the entire data. For example, if you have a list of 32 file names, and the user only can see 8 names at one time (25%), the knob (AUTOKNOB) should fill 25% of the container. HorizBody should in this case be set to: MAXBODY * 8 / 32 (25% of 0xFFFF) == 3FFFF

vert_body: Same as HorizBody except that it affects VertPot, and the vertical size of the knob (AUTOKNOB).

4.10 EXAMPLES

The printed version of these examples can be found in Appendix A (Part IV: Appendices).

Example1

This program will open a normal window which is connected to the Workbench Screen. The window will use all System Gadgets, and will close first when the user has selected the System gadget Close window. (Same as Example3 in chapter 2 WINDOWS, except that we have added an IDCMP check on the Close window gadget.)

Example2

Same as Example1 except that we have added a Boolean gadget with the text "PRESS ME".

Example3

Same as Example2 except that the on/off state of the gadget is toggled each time the user hits the gadget.

Example4

This program will open a normal window which is connected to the Workbench Screen. The window will use all System Gadgets, and will close first when the user has selected the System gadget Close window. Inside the window we have put two Boolean gadgets with the text "GADGET 1" and "GADGET 2".

Example5

This program will open a normal window which is connected to the Workbench Screen. The window will use all System Gadgets, and will close first when the user has selected the System gadget Close window. Inside the window we have put a Boolean gadget with two Image structures connected to it. Each time the user clicks on the gadget it will change images, lamp on/lamp off.

Example6

This program will open a normal window which is connected to the Workbench Screen. The window will use all System Gadgets, and will close first when the user has selected the System gadget Close window. Inside the window we have put a Boolean gadget with a connecting mask. The gadget will only be highlighted when the user selects this gadget while pointing inside the specified (masked) area.

Example7

This program will open a normal window which is connected to the Workbench Screen. The window will use all System Gadgets, and will close first when the user has selected the System gadget Close window. Inside the window we have put a String gadget.

Example8

Same as Example7 except that it is an Integer gadget.

Example9

Same as Example7 except that it is a Proportional gadget.

Example10

Same as Example9 except that the Proportional gadget uses a custom image knob.

Example11

This program will open a normal window which is connected to the Workbench Screen. The window will use all System Gadgets, and will close first when the user has selected the System gadget Close window. Inside the window we have put a Proportional gadget where the knob can be moved both horizontally and vertically.

Example12

This program will open a SuperBitmap window which is connected to the Workbench Screen. The window will use all System Gadgets, and will close first when the user has selected the System gadget Close window. Inside the window we have put two Proportional gadgets, one on the right side, and one at the bottom. With help of these two gadgets, the user can move around the BitMap.

This example is for experienced programmers only, since it uses some functions etc which we have not discussed yet. I have, however, included it here since it is a good example on how you can combine Proportional gadgets with SuperBitmap windows.

ACM - Amiga C Manual
Anders Bjerin

Part II: Requesters, Alerts, Menus, IDCMP, Miscellaneous

http://aminet.net/package/dev/c/ACM

The complete boiled-down C manual for the Amiga which describes how to open and work with Screens, Windows, Graphics, Gadgets, Requesters, Alerts, Menus, IDCMP, Sprites, VSprites, AmigaDOS, Low Level Graphics Routines, Hints and Tips, etc. The manual also explains how to use your C Compiler and gives you important information about how the Amiga works and how your programs should be designed. The manual consists of 15 chapters together with more than 100 fully executable examples with source code.

5 REQUESTERS

5.1 INTRODUCTION

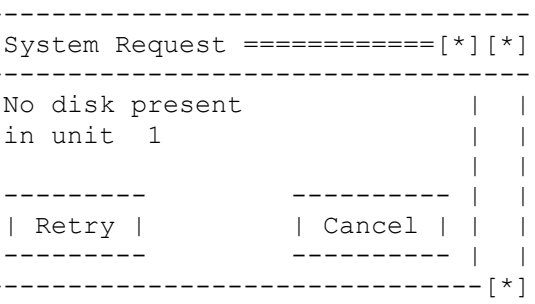
Requesters are boxes filled with some sort of information which the user need to respond to. It can be as simple as a question to insert a disk in the internal drive, or a fully functional file requester. Requesters are like small windows with some gadgets connected to it, and they will first disappear when the user has "satisfied the request".

5.2 DIFFERENT TYPES OF REQUESTERS

There exist three different types of requesters:
- System requesters
- Application requesters
- Double-menu requesters

5.2.1 SYSTEM REQUESTERS

System requesters are opened and maintained by Intuition, and your program has no control over them. If the user, for example, is trying to load a file from drive df1:, and there is no disk present, the operating system would open the following request:



One important thing about System requesters is that they are like small windows. You can move around them, push them behind or in front of other windows/requesters, and resize them.

5.2.2 APPLICATION REQUESTERS

This is the type of requester your program can open. They can be of any size (limited only by the size of the screen), and can be as simple or as complicated as you want.

If you only want a Yes/No (True/False) requester you can call the function AutoRequest(), and Intuition will open and take care of the rest. If you on the other hand want a more sophisticated requester you need to declare and initialize a Requester structure with your requirements, and call the function Request().

It is important to notice that an Application requester can not be moved around, resized etc (the window containing the requester can still, of course, be moved around and resized). It is only the System requesters, and the very simple requesters opened by the AutoRequest function, which acts like small windows.

There is also one other big difference. When an Application requester has been activated the user can no longer select gadgets connected to the window. The window has been "frozen". Eg. try Example4, and you will notice that you can not select the close-window gadget as long as the requester is active. On the other hand, if a System requester has been activated, the user can still click on the close-window gadget, and a CLOSEWINDOW message is sent.

5.2.3 DOUBLE-MENU REQUESTERS

Double-menu requesters are like normal Application requesters except that they will open first, and only, when the user double-clicks on the mouse menu button. To create a Double-menu requester you need to declare and initialize a Requester structure, and then call the function SetDMRequest(). Whenever the user from now on double-clicks on the menu button on the mouse, the requester will be opened. Call the function ClearDMRequest() when you do not want the user to be able to open the requester any more.

Only one DM-requester may be connected to each window.

5.3 GRAPHICS FOR REQUESTERS

You can render a requester in two different ways. You can either tell Intuition what you want, and everything is rendered for you, or you can supply Intuition with your own customized Bitmap (which has been rendered by yourself).

If you want Intuition to draw the requester for you, you only need to decide what colour the background of the requester should be filled with, and declare and initialize one or more Border and IntuiText structures.

If you supply your own customized Bitmap Intuition will not draw anything itself. That means that all gadget connected to that requester does not need to have any Border/IntuiText/Image structures connected to them, since Intuition will not bother about them.

5.4 POSITION

The requester can either be positioned relative to the top left corner of the window, or relative to the pointer position. If you want to position the requester relative to the window, you simply set the LeftEdge and TopEdge variables as desired. If you, on the other hand, want it relative to the pointer position, set the POINTREL flag, and initialize RelLeft and RelTop as desired.

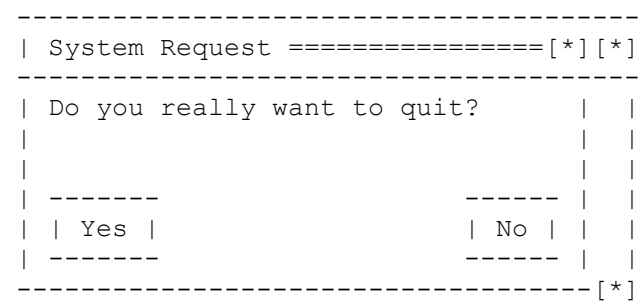
5.5 REQUESTERS AND GADGETS

When you create a requester, except when you call the function AutoRequest(), you need to connect at least one gadget to the requester. Gadgets connected to a requester works exactly the same as gadgets connected to a window but with two important differences:

1. Every gadget connected to a requester MUST have the REQGADGET flag set in the GadgetType field.

2. At least one gadget must satisfy the request, which means at least one gadget must have the ENDGADGET flag set in the Activation field. (Once a gadget with an ENDGADGET flag set is selected, the requester is satisfied, and will close.)

Remember that the user should always be able to find a safe way to leave the requester without affecting anything. The "way out gadget" (CANCEL, RESUME etc) should also always be placed on the right side in the requester. Here is an example:



Note that the Yes/True/Retry button is always on the left side and that the No/False/Cancel button is always on the right side.

IMPORTANT! Make sure that the Gadgets are inside the requester, since Intuition will not do any boundary checking.

5.6 SIMPLE REQUESTERS

If you just want a simple requester with a True/False (or just False) option, you can use the function AutoRequest(). You give Intuition some information about how the requester should look like (height, text etc), and Intuition takes care of everything else. It opens the requester, put your program to rest, and when finished, returns a boolean value which tells your program what the user selected. If the user selected the left gadget (positive) it returns TRUE, and if the user selected the right gadget (negative) it returns FALSE.

Example:

```
result = AutoRequest( window, info_text, pos_text, neg_text,
                      pos_IDCMP, neg_IDCMP, width, height );
```

window: Pointer to a window if there exist one, else NULL.

info_text: Pointer to an IntuiText structure containing the "body text".

pos_text: Pointer to an IntuiText structure containing the "positive text". Eg: "TRUE", "YES", "RETRY" etc. (Optional)

neg_text: Pointer to an IntuiText structure containing the "negative text". Eg: "FALSE", "NO", "CANCEL" etc.

pos_IDCMP: The IDCMP flags which satisfy the "positive" gadget. (The flag RELVERIFY is already set.)

pos_IDCMP: The IDCMP flags which satisfy the "negative" gadget. (The flag RELVERIFY is already set.)

width: How many pixels wide the requester should be.

height: How many lines high the requester should be.

5.7 OPEN A REQUESTERS

If you want to use a more complicated requester instead of the Simple requester you need to:

1. Declare and initialize a Requester structure with your requirements.

2. Declare and initialize the Gadget structures which are going to be connected to the requester.

3. Call the function Request() in order to display the requester as a normal requester, or call the function SetDMRequest() to enable the user to bring up the requester as a Double-menu requester.

5.7.1 INITIALIZE A REQUESTER

If you want to use a requester you need to declare and initialize a Requester structure which look like this:

```
struct Requester
{
  struct Requester *OlderRequest;
  SHORT LeftEdge, TopEdge;
  SHORT Width, Height;
  SHORT RelLeft, RelTop;
  struct Gadget *ReqGadget;
  struct Border *ReqBorder;
  struct IntuiText *ReqText;
  USHORT Flags;
  UBYTE BackFill;
  struct Layer *ReqLayer;
  UBYTE ReqPad1[32];
  struct BitMap *ImageBMap;
  struct Window *RWindow;
  UBYTE ReqPad2[36];
};
```

OlderRequest: Initialized and maintained by Intuition. Set to NULL.

LeftEdge, TopEdge: Position of the requester relative to the top left corner of the window (if the POINTREL flag is not set).

Width, Height: Size of the requester.

RelLeft, RelTop: If the POINTREL flag is set, these values describes the position of the requester relative to the pointer.

ReqGadget: Pointer to the first gadget in the linked list. Remember, there must exist at least one gadget connected to the requester with the ENDGADGET flag set.

ReqBorder: Pointer to a Border structure used to render the requester.

ReqText: Pointer to an IntuiText structure used to print text in the requester.

Flags: There exist two flags which you may set:

POINTREL: Set this flag if you want to position the requester relative to the pointer. Set the RelLeft, RelTop as desired.

PREDRAWN: If you supply with your own customized Bitmap set this flag, and Intuition will not try to draw anything itself.

Intuition sets these flags:

REQACTIVE: This flag is set when the requester is activated, and cleared when the requester is closed (deactivated).

REQOFFWINDOW: This flag is set if the requester is active and positioned outside the window. (The user has maybe shrinked the window so it is smaller than you thought.)

SYSREQUEST: This flag is set if the requester is of the type System requester.

BackFill: Colour register used to fill the requester with before any drawing takes place. For example, if you want to render the graphics on an orange background, you set the BackFill field to 3 (orange, normal WB colours).

ReqLayer: Pointer to the Layer structure for this requester. Initialized and maintained by Intuition. Set to NULL.

ReqPad1: Initialized and maintained by Intuition. Set to NULL. (Used by the system.)

ImageBMap: If the bit PREDRAWN is set, this field should contain a pointer to a customized Bitmap, else NULL.

RWindow: Initialized and maintained by Intuition. Set to NULL. (Used by the system, points back to the Window which this requester is connected to.)

ReqPad2: Initialized and maintained by Intuition. Set to NULL. (Used by the system.)

Here is an example of how to initialize a Requester structure:

```
struct Requester my_requester=
{
  NULL,              /* OlderRequester, used by Intuition.   */
  40, 20,            /* LeftEdge, TopEdge, 40 pixels out, 20 */
                     /* lines down.                          */
  320, 100,          /* Width, Height, 320 pixels wide, 100  */
                     /* lines high.                          */
  0, 0,              /* RelLeft, RelTop, Since POINTREL flag */
                     /* is not set, Intuition ignores these  */
                     /* values.                              */
  &my_first_gadget,  /* ReqGadget, pointer to the first      */
                     /* gadget.                              */
  &my_border,        /* ReqBorder, pointer to a Border       */
                     /* structure.                           */
  &my_text,          /* ReqText, pointer to a IntuiText      */
                     /* structure.                           */
  NULL,              /* Flags, no flags set.                 */
  3,                 /* BackFill, draw everything on an      */
                     /* orange background.                   */
  NULL,              /* ReqLayer, used by Intuition. Set to  */
                     /* NULL.                                */
  NULL,              /* ReqPad1, used by Intuition. Set to   */
                     /* NULL.                                */
  NULL,              /* ImageBMap, no predrawn Bitmap. Set   */
                     /* to NULL. (The PREDRAWN flag was not  */
                     /* set.)                                */
  NULL,              /* RWindow, used by Intuition. Set to   */
                     /* NULL.                                */
  NULL               /* ReqPad2, used by Intuition. Set to   */
                     /* NULL.                                */
};
```

If you would like to supply your own customized and predrawn Bitmap, instead of letting Intuition render the requester you need to:

1. Set the flag PREDRAWN in the Flags field.

2. Set the ImageBMap to point at your BitMap structure.

Remember that Intuition will now not draw anything for you. The ReqBorder, ReqText and BackFill variables are ignored, and should therefore be set to NULL. Intuition will also not render the gadgets connected to the requester. Because of this it is important that the rendering of the Bitmap is done carefully, and that the graphics correspond to where the gadgets are etc.

5.7.2 HOW TO ACTIVATE AN APPLICATION REQUESTER

If you have declared and initialized a Requester structure you only need to call the function Request() in order to activate it.

Example:

```
result = Request( my_requester, my_window );
```

my_requester: Pointer to the Requester structure.

my_window: Pointer to the Window structure which the requester should be connected to.

result: Boolean value returned. If Intuition could successfully open the requester the function returns TRUE, else (something went wrong, not enough memory etc) the function returns FALSE.

If you on the other hand would like the requester to be a Double-menu requester you should call the function SetDMRequest(), which will allow the user to activate the requester by double clicking the mouse menu button.

Example:

```
result = SetDMRequest( my_window, my_requester );
```

my_window: Pointer to the Window structure which the requester should be connected to.

my_requester: Pointer to the Requester structure.

result: Boolean value returned. If Intuition could successfully open the requester the function returns TRUE, else (some thing went wrong, not enough memory or the a DM requester is already connected to the window etc) the function returns FALSE.

You can after you have called the SetDMRequest() function successfully, take away the ability for the user to open the requester by calling the function ClearDMRequest().

Example:

```
result = ClearDMRequest( my_window );
```

my_window: Pointer to the Window structure which the requester is connected to.

result: If the function could disable the user to activate the DM-requester it returns TRUE, else (something went wrong, the requester is in use etc) it returns FALSE.

5.8 IDCMP FLAGS

There exist three IDCMP flags which are special for the requesters. When a requester is activated you can, if you want to, receive a message telling you that a requester was activated. This is especially useful if you are using Double-menu requesters, since this is the only way to check if the requester was opened.

If you want to get a message every time a requester is activated, you need to set the flag REQSET in the IDCMPFlags field in the NewWindow structure. Set the flag REQCLEAR if you want to receive a message every time a requester is deactivated. See Example5 for more details.

There exist also one special IDCMP flag called REQVERIFY. If you set this flag in the IDCMPFlags field in the NewWindow structure your program will receive a message when the user is trying to activate a Double-menu requester. The interesting thing about this flag is that the requester will not be opened until your program has replied, ReplyMsg(). Your program can therefore finish of something (like finish of with drawing something etc) before the requester is displayed, and when your program is ready, it can reply, and the requester is activated. See Example6 for more information.

5.9 FUNCTIONS

Here are some commonly used functions:

AutoRequest()

This function opens a Simple requester. Intuition will automatically activate it and take care of the response from the user. It will return TRUE if the left gadget was selected, and FALSE if the right gadget was selected.

```
  Synopsis:  result = AutoRequest( my_window, info_txt, pos_txt,
                                   neg_txt, pos_IDCMP, neg_IDCMP,
                                   width, height );

  my_window: (struct Window *) Pointer to a window if there
             exist one, else NULL.

  info_txt:  (struct IntuiText *) Pointer to an IntuiText
             structure containing the "body text".

  pos_txt:   (struct IntuiText *) Pointer to an IntuiText
             structure containing the "positive text". Eg:
             "TRUE", "YES", "RETRY" etc. (Optional)

  neg_txt:   (struct IntuiText *) Pointer to an IntuiText
             structure containing the "negative text". Eg:
             "FALSE", "NO", "CANCEL" etc.

  pos_IDCMP: (long) IDCMP flags which satisfy the "positive"
             gadget. (The flag RELVERIFY is already set.)

  pos_IDCMP: (long) IDCMP flags which satisfy the "negative"
             gadget. (The flag RELVERIFY is already set.)

  width:     (long) How many pixels wide the requester should
             be.

  height:    (long) How many lines high the requester should
             be.

  result:    (long) Boolean value. The function returns TRUE if
             the positive gadget was satisfied, and FALSE if
             the negative gadget was satisfied.


Request()

  This function activates a requester connected to a window.

  Synopsis:     result = Request( my_requester, my_window );

  my_requester: (struct Requester *) Pointer to the Requester
                structure.

  my_window:    (struct Window *) Pointer to the Window
                structure which the requester should be
                connected to.

  result:       (long) Boolean value returned. If Intuition
                could successfully open the requester the
                function returns TRUE, else (something went
                wrong, not enough memory etc) the function
                returns FALSE.


EndRequest()
```

This function deactivates a requester which has been activated.

Synopsis: EndRequest(my_requester, my_window);

my_requester: (struct Requester *) Pointer to the Requester structure which will be removed.

my_window: (struct Window *) Pointer to the Window structure which the requester is connected to.

SetDMRequest()

This function allows the user to activate a Double-menu requester by clicking twice on the mouse menu button.

Synopsis: result = SetDMRequest(window, requester);

window: (struct Window *) Pointer to the Window structure which the requester should be connected to.

requester: (struct Requester *) Pointer to the Requester structure.

result: (long) Boolean value returned. If Intuition could successfully open the requester the function returns TRUE, else (something went wrong, not enough memory or a DM requester is already connected to the window, etc) the function returns FALSE.

ClearDMRequest()

This function disables a Double-menu requester. The user can not open the requester any more.

Synopsis: result = ClearDMRequest(my_window);

my_window: (struct Window *) Pointer to the Window structure which the requester is connected to. The DMRequest pointer in the Window structure is set to NULL.

result: (long) If the function could disable the DM-requester it returns TRUE, else (something went wrong, the requester is in use etc) it returns FALSE.

5.10 EXAMPLES

The printed version of these examples can be found in Appendix A (Part IV: Appendices).

Example1

This example opens a Simple requester by calling the function AutoRequest. It displays a message "This is a very simple requester!", and has only one gadget connected to it (on the right side of the requester) with the text "OK".

Example2

Same as Example1, except that the requester displays a message "Do you really want to quit?", and allows the user to choose between "Yes" and "No". The program will continue to reopen the requester until the user has chosen "Yes".

Example3

Same as Example1, except that this requester displays a message "Insert a disk in any drive!", and allows the user to choose between "Yes" and "No". The program will continue to reopen the requester until the user has chosen "Yes" or inserted a disk.

Example4

This program will open a normal window which is connected to the Workbench Screen. The window will use all System Gadgets, and will close first when the user has selected the System gadget Close window. Inside the window we have activated an Application requester with a connecting gadget. The requester will first be satisfied when the user has selected the gadget, and will then be deactivated. The window can now be closed.

Example5

Same as Example4, except that the requester is first activated when the user double-clicks on the right mouse button. This example shows how to create a Double-menu requester, and how to monitor the IDCMP flags REQSET and REQCLEAR.

Example6

Same as Example5, except that whenever the user double-clicks on the right mouse button, we will receive a REQVERIFY message, and first when we have replied, will the requester be activated. This example shows how to use the REQVERIFY flag.

Example7

This program will open a normal window which is connected to the Workbench Screen. The window will use all System Gadgets, and will close first when the user has selected the System gadget Close window. Inside the window we have activated an Application requester with three connecting gadgets. Two are Boolean gadgets ("OK and "CANCEL"), and one is a String gadget.

Example8

Same as Example7, except that it is an Integer gadget.

Example9

Same as Example8, except that it is a Proportional gadget.

6 ALERTS

6.1 INTRODUCTION

Alerts is the last resource your program can use in order to inform the user about a problem. When your program displays an alert all screens are moved down, and a black and red box is opened at the top of the display. It not only gives the user a hart attack, but it will also tell him/her what went wrong, and if there is a way out.

6.2 DIFFERENT LEVELS OF WARNINGS

There exist three levels of warnings:

1. If you want to alert the user that something went wrong, but it is nothing serious, you can flash the screens. You do it by calling the function DisplayBeep(), and the background colour of all screens will be flashed.

2. If something went wrong, and you want to inform the user, maybe even want that he/she does something, you should open a Requester. (Described in chapter 5 REQUESTERS.)

3. If something went TERRIBLE WRONG, (the system is crashing etc) your program should activate an Alert message, DisplayAlert().

6.3 HOW TO USE THE DISPLAYALERT() FUNCTION

Synopsis: result = DisplayAlert(nr, message, height);

nr: (long) Value which describes if it is a RECOVERY_ALERT or a DEADEND_ALERT.

message: (char *) Pointer to an array of characters (char). It contains the strings we want to display, and some extra information (position etc). The string itself is divided into substrings, which all contain information about its position etc.

Each substring consists of:

- 2 bytes (16-bit) which are used for the x position of the text.

- 1 byte (8-bit) which is used for the y position of the text.
- The text string which ends with a NULL ('\0') sign.
- A Continuation byte. If it is TRUE there is another substring after this one, else this was the last substring.

(See below for more information)

height: (long) The height of the Alert box.

result: (long) The function DisplayAlert() returns a boolean value. If it is a RECOVERY_ALERT and the user pressed the left mouse button it returns TRUE else, if the user pressed the right mouse button, it returns FALSE. If it is a DEADEND_ALERT the function will immediate return FALSE.

6.4 EXAMPLES OF STRINGS AND SUBSTRINGS

If we want to display the following Alert message:

```
-----------------------------------------------------------------
|                                                               |
|  ERROR! Not enough memory!                                    |
|                                                               |
-----------------------------------------------------------------
```

the string would be declared and initialized like this:

```
  /* Declare the array of char (the string): */
  char message[30]; /* 30 bytes needed. */

  /* Fill the string with the message: (Remember to give    */
  /* space for the first 3 bytes which will contain the x/y */
  /* position.)                                             */
  strcpy( message, "   ERROR! Not enough memory!" );
  /* The NULL sign is automatically placed at position 28. */

  /* Fill the string with the position (x,y) (first 3 bytes)  */
  message[0]=0;  /* X position is less than 256, therefore 0. */
  message[1]=32; /* 32 pixels out. */
  message[2]=16; /* 16 lines down. */

  /* Set the Continuation byte to FALSE since there are no */
  /* more substrings after this one:                       */
  message[29]=FALSE;
```

If we on the other hand want to display the following Alert message:

```
-------------------------------------------------------------
|                                                           |
|  ERROR! Not enough memory!                                |
|                                                           |
|  Buy a memory expansion!                                  |
|                                                           |
-------------------------------------------------------------
```

the string would be declared and initialized like this:

```
  /* Declare the array of char (the string): */
  char message[58]; /* 58 bytes needed. */

  /* Fill the array with the first substring: (Remember to */
  /* give space for the first 3 bytes which will contain   */
  /* the x/y position.)                                    */
  strcpy( message, "   ERROR! Not enough memory!" );

  /* Add the second substring: (Remember this time to give */
  /* space for 5 bytes in the beginning of the string.     */
  /* They are used for the NULL sign which finish off the  */
  /* first substring, the Continuation byte, and three     */
  /* bytes for the position for the second substring.)     */
  strcat( message, "     Buy a memory expansion!");
  /* The NULL sign which finish of the second substring is */
  /* automatically placed at position 56. */

  /* Fill the first substring with the position (x,y) */
  /* (first 3 bytes) */
  message[0]=0;  /* X position is less than 256, therefore 0. */
  message[1]=32; /* 32 pixels out. */
  message[2]=16; /* 16 lines down. */

  /* Add the NULL sign which finish of the first substring: */
  /* (It was deleted when we connected the two strings with */
  /* the strcat() function.)                                */
  message[28]='\0';

  /* Set the Continuation byte to TRUE which tells     */
  /* Intuition that there is another substring coming: */
  message[29]=TRUE;

  /* Fill the second substring with the position (x,y): */
  message[30]=0;  /* X position is less than 256. */
  message[31]=32; /* 32 pixels out. */
  message[32]=32; /* 32 lines down. */

  /* Set the Continuation byte to FALSE since there are */
  /* no more substrings after this one: */
  message[57]=FALSE;
```

6.5 FUNCTIONS

DisplayAlert()

This function activates an Alert message.

Synopsis: result = DisplayAlert(nr, message, height);

nr: (long) Value which describes if it is a RECOVERY_ALERT or a DEADEND_ALERT.

message: (char *) Pointer to an array of characters (char). It contains the strings we want to display, and some extra information (position etc). The string itself is divided into substrings, which all contain information about its position etc.

- 2 bytes (16-bit) which are used for the x position of the text.
- 1 byte (8-bit) which is used for the y position of the text.
- The text string which ends with a NULL ('\0') sign.
- A Continuation byte. If it is TRUE there is another substring after this one, else this was the last substring.

height: (long) The height of the Alert box.

result: (long) The function DisplayAlert() returns a boolean value. If it is a RECOVERY_ALERT and the user pressed the left mouse button it returns TRUE else, if the user pressed the right mouse button, it returns FALSE. If it is a DEADEND_ALERT the function will immediate return FALSE.

6.6 EXAMPLES

The printed version of this example can be found in Appendix A (Part IV: Appendices).

Example1

This example displays an Alert message at the top of the display.

7 MENUS

7.1 INTRODUCTION

If you have been working on the Amiga you have probably used menus a lot. They are flexible tools which are easy to access and are not complicated to set up. In this chapter we will look at how you can create your own menus, how to use all the special features offered by Intuition, and how the communication between the user and the menus is executed.

7.2 MENU DESIGN

You can connect a "menu strip" to every window, and when the user presses down the right mouse button, the active window's menu strip is shown at the top of the display.

An example of a "menu strip":

```
Project  Block  Windows  Search
```

The user can now move the pointer to one of the menu headings, while still holding the right mouse button pressed, and that menu's "item box" is displayed.

An example of a "item box":

```
PROJECT  Block  Windows  Search
|----------|
| Open     |
| Save     |
| Save as  |
| Print    |
| Info     |
| Quit     |
|----------|
```

The user can now choose the "menu item" he/she wants. The user does it by moving the pointer to the desired menu item, and releases the right mouse button. If the user wants he/she can pick several menu items at the same "menu action" by holding down the right mouse button as normal, but click with the left mouse button on each menu item the user wants to select. The user can also "drag" (holding both mouse buttons down) over several menu items if he/she wants to select all of them.

You can connect a separate menu to an item if you want. That

menu is usually referred as "submenu", and it consists of one or more "subitems".

An Example of "subitems": (Note that the subitem box overlap the item box!)

```
Project  Block  Windows  Search
|----------|
| Open     |
| Save     |
| Save a--------------
| PRINT | to printer |
| Info  | to a file  |
| Quit  --------------
|----------|
```

7.3 HOW TO ACCESS MENUS FROM THE KEYBOARD

If you want you can allow the user to select certain menu items from the keyboard. This is very handy if it is a command the user will often use, and you want to provide him/her with a shortcut. The user can access the menu item by pressing the right "Amiga key" + another key. For example, if the user should be able to access the menu item "Open" from the keyboard by pressing the "Amiga key" + "O", Intuition will automatically add the sign "[A] O" after the item's name:

```
PROJECT  Block  Windows  Search
|------------|
| Open [A] O |
| Save       |
| Save as    |
| Print      |
| Info       |
| Quit       |
|------------|
```

Your program will not notice any difference between a normal menu access, and a shortcut.

Remember to give enough space on the right side of the item box, so the Amiga character + sign will fit. If the item box was before 100 pixels wide you can set the width to 100 + COMMWIDTH, and everything will fit in nicely. (COMMWIDTH is a constant declared in the intuition.h file.) If the window is connected to a low-resolution screen, use the constant LOWCOMMWIDTH instead.

7.4 MENU ITEMS

There exist two different types of menu items:
- Action items
- Attribute items

Action items can be selected over and over again, and every time they are selected they will send your program a message.

Attribute items on the other hand can only be selected once, and once they have been selected they need to be deselected before they can be selected again. The only way to deselect an Attribute item is to mutual exclude them (explained later in this chapter).

When an attribute item is selected Intuition will put a small checkmark, [v], before the item's name. If you want to use your own designed checkmark instead of the default one, you set the CheckMark pointer in your NewWindow structure to point to an Image structure which you have declared and initialized as desired.

If you use any attribute item in your menu you need to leave enough space at the left side of the box for the checkmark. If you use the default checkmark you can move the text in the menu CHECKWIDTH amount of pixels to the right. (CHECKWIDTH is a constant declared in the intuition.h file.) If the window is connected to a low-resolution screen (320 pixels wide), you should use the constant LOWCHECKWIDTH instead of CHECKWIDTH.

7.5 MUTUAL EXCLUDE

This special and very useful function applies only on attribute items. The idea with mutual exclude is that when the user selects a certain item, some other attribute items are deselected automatically. For example, you are writing a wordprocessor and the user should be able to print the text as normal (plain) or with some special style (Bold, Underline or Italic). The user would then only be allowed to select Plain or one or more of the other styles (Bold and underlined for example). If the user selects Plain, all other styles should be deselected (mutual excluded). If the user on the other hand selects Bold, Underline or Italic the Plain item should be deselected.

Mutual exclude works like this:

1. If the flag CHECKIT (see the MenuItem structure) is not set, the item is not an attribute item, it is an action item, and mutual exclude will have no effect on this item.

2. If the flag CHECKIT is set, the item is an attribute item, and Intuition will make sure that the CHECKED flag is not

set. If it is, Intuition will automatically remove it, and the item is unselected.

Every item has a MutualExclude field in which you can specify which items should be deselected when this item is activated. The first bit in the field refers to the first item in the item list, the second bit to the second item and so on.

In our previous example we would set the Plain item's MutualExclude field to 0xFFFFFFFE (0xFFFFFFFE is equal to 11111111111111111111111111111110). This would mutual exclude all items except the first one. (Plain is the first item in the list, and should of course not be mutual excluded.)

The other items (Bold, Underline and Italic) should have their field set to 0x00000001 (0x00000001 is equal to 00000000000000000000000000000001). This would only mutual exclude the first item (the Plain item).

7.6 OPEN A MENU

If you want to attach a menu strip to a window you need to:

1. Declare and initialize one or more Menu structures. The Menu structures should be linked together, and each Menu structure has its own list of items. Each Menu structure represent one "menu heading"

2. Declare and initialize one or more MenuItem structures which are connected to the Menu structures. Each MenuItem structure represent one item. You can connect a list of "subitems" to an Item structure if you want. (Subitems use the same data structure as normal items.)

3. Call the function SetMenuStrip(). Important, before you close the window, or change the "menu strip" you must disconnect the menu from the window. You do it by calling the function ClearMenuStrip().

7.6.1 INITIALIZE A MENU STRUCTURE

The Menu structure look like this:

```
struct Menu
{
  struct Menu *NextMenu;
  SHORT LeftEdge, TopEdge, Width, Height;
  USHORT Flags;
  BYTE *MenuName;
```

```
  struct MenuItem *FirstItem;
  SHORT JazzX, JazzY, BeatX, BeatY;
};
```

NextMenu: Pointer to the next Menu structure in the menu strip list. The last Menu structure in the list must have the NextMenu set to NULL.

LeftEdge: X position of the menu.

TopEdge: Y position of the menu. For the moment this field is unused since all menus are positioned at the top of the display. Set it to 0.

Width: The width of the menu.

Height: The height of the menu. For the moment this field is unused since all menus are as high as the height of the title bar of the screen. Set it to 0.

Flags: There exist only two different flags for the moment:

MENUENABLED: If this flag is set the menu is enabled, and the user can select items from it. If it is not set, the menu is ghosted and the user can not select any items from it.

If you want the menu to be enabled when you submit it to Intuition you should set this flag. You can later change the status by calling the functions OnMenu() and OffMenu().

MIDRAWN: This flag is set by Intuition when this menu is displayed to the user.

MenuName: Pointer to a NULL-terminated string. The text will appear in the menu box at the top of the screen.

FirstItem: Pointer to the first MenuItem in the item list.

These funny-named variables are for internal use only:

JazzX, JazzY, BeatX, BeatY: Used by Intuition. Set to 0.

Here is an example of how to declare and initialize a Menu structure:

```
struct Menu my_menu=
{
  NULL,         /* NextMenu, no other menu structures   */
                /* connected.                           */
  0,            /* LeftEdge, 0 pixels to the right.     */
  0,            /* TopEdge, for the moment ignored by   */
                /* Intuition. Set to 0.                 */
```

```
  60,          /* Width, 60 pixels wide.               */
  0,           /* Height, for the moment ignored by    */
               /* Intuition. Set to 0.                 */
  MENUENABLED, /* Flags, this menu should be enabled.  */
  "Project",   /* MenuName, the title of the menu.     */
  &first_item, /* FirstItem, pointer to the first item */
               /* in the list.                         */

  0, 0, 0, 0   /* JazzX, JazzY, BeatX, BeatY           */
};
```

7.6.2 INITIALIZE A MENUITEM STRUCTURE

The MenuItem structure look like this:

```
struct MenuItem
{
  struct MenuItem *NextItem;
  SHORT LeftEdge, TopEdge, Width, Height;
  USHORT Flags;
  LONG MutualExclude;
  APTR ItemFill;
  APTR SelectFill;
  BYTE Command;
  struct MenuItem *SubItem;
  USHORT NextSelect;
};
```

NextItem: Pointer to the next MenuItem structure in the item list. The last item should have its NextItem field set to NULL.

LeftEdge: X position of the item's select box relative to the LeftEdge of the Menu structure.

TopEdge: Y position of the item's select box relative to the topmost position allowed by intuition. The topmost position is automatically calculated by Intuition which examines the font size, how many items above etc. Set TopEdge to 0 in order to select the topmost position.

Width: The width of the item's select box.

Height: The height of the item's select box.

Flags: Here are the flags you and Intuition can use:

CHECKIT: Set this flag if you want the item to be an attribute item, else it will be an action item. If you set this flag Intuition will put the small checkmark in front of the

itmetext when this item is selected, so make sure there is enough space for it.

CHECKED: If the item is an attribute item (the CHECKIT flag is set), this flag is set by Intuition when the item is selected by the user. The flag is also automatically removed by Intuition when this item is mutual excluded.

You can set this flag yourself before you submit the menu strip to Intuition, if you want the item to be initially selected.

HIGHITEM: The item will be highlighted, and this flag set, when the user moves the pointer over the item's select box.

You need to set one of the following four flags which will tell Intuition what kind of highlighting you want:

HIGHCOMP: Complement the colours of all the pixels in the item's select box.

HIGHBOX: Draw a box around the item's select box.

HIGHIMAGE: Display an alternative Image or Text.

HIGHNONE: No highlighting.

ITEMENABLED: If this flag is set the item is enabled, and the user can select it. If this flag is not set, the item is ghosted and the user can not select it.

If you want the item to be enabled when you submit it to Intuition you should set this flag. You can later change the status by calling the functions OnMenu() and OffMenu().

ITEMTEXT: If the item is rendered with text you should set this flag. Intuition will then expect ItemFill and SelectFill to point to IntuiText structures. If this flag is not set the item will be rendered with an

image, and Intuition will expect ItemFill and SelectFill to point to Image structures.

COMMSEQ: If the item also should be accessible from the keyboard you need to set this flag. Remember to tell Intuition which key should be pressed together with the "Amiga key" in order to select the item. You do it by giving the variable Command the desired character.

ISDRAWN: This flag is set by Intuition when this item's subitems are displayed, and cleared when not.

MutualExclude: This variable will tell Intuition which attribute items should be mutual excluded (deselected when this item is selected). The first bit refers to the first item in the active list (item list or subitem list), the second bit refers to the second item and so on. For example:

The number 0x0000000B (hexadecimal) is equal to 00000000000000000000000000001011 (binary), and Intuition would therefore try to mutual exclude the first, the second and the fourth item.

ItemFill: Pointer to an Image structure (the ITEMTEXT flag not set) or to an IntuiText structure (the ITEMTEXT flag set), which will be used by Intuition to render this item.

SelectFill: Pointer to an Image structure (the ITEMTEXT flag not set) or to an IntuiText structure (the ITEMTEXT flag set), which will render this item when it is highlighted. (This field is only used if the flag HIGHIMAGE is set.)

Command: If the user should be able to access this item from the keyboard, you need to tell Intuition which key should be used. This field should therefore contain the desired character. (Remember to set the flag COMMSEQ in the Flags field.)

SubItem: Pointer to a subitem list if there exist on, else NULL. If this item is already a subitem, this field is ignored by Intuition. (Remember, the subitem list consists of a linked list of MenuItem structures.)

NextSelect: This field is set by Intuition and will contain a special number if more items after this one was selected at the same menu event, else it

will contain MENUNULL. (This will be discussed later in the chapter.) Set it to MENUNULL.

Here is an example of how to declare and initialize a MenuItem structure:

```
struct MenuItem my_first_item=
{
  &my_second_item,   /* NextItem, pointer to the next item   */
                     /* in the list.                         */
  0,                 /* LeftEdge, 0 pixels to right.         */
  0,                 /* TopEdge, the topmost position.       */
  150,               /* Width, 150 pixels wide.              */
  10,                /* Height, 10 lines high.               */
  CHECKIT|           /* Flags, an attribute item.            */
  CHECKED|           /*        should initially be selected. */
  COMMSEQ|           /*        accessible from the keyboard. */
  HIGHCOMP|          /*        complement the colours when   */
  ITEMENABLED,       /*        highlighted. The item should  */
                     /*        be enabled. Not ghosted.      */
  0x00000037,        /* MutualExclude, deselect if possible  */
                     /* item: 1, 2, 3, 5 and 6.              */
  &my_image          /* ItemFill, pointer to an Image        */
                     /* structure used to render the item.   */
                     /* Since the flag ITEMTEXT was not set, */
                     /* Intuition will assume that this is a */
                     /* pointer to an Image structure, and   */
                     /* not a pointer to an IntuiText        */
                     /* structure.                           */
  NULL,              /* SelectFill, NULL since we complement */
                     /* the colours instead of displaying an */
                     /* alternative image.                   */
  'O',               /* Command, the character which should  */
                     /* be pressed together with the "Amiga  */
                     /* key" in order to execute the         */
                     /* shortcut.                            */
  NULL,              /* SubItem, this item has no subitems   */
                     /* connected to it.                     */
  MENUNULL           /* NextSelect, set later by Intuition.  */
};
```

7.6.3 HOW TO SUBMIT AND REMOVE A MENU STRIP TO/FROM A WINDOW

Once you have declared and initialized the necessary Menu and MenuItem structures and linked them together you only need to call the function SetMenuStrip(), and the menu strip is connected to your window. Remember that you need to first open the window, before you can submit a menu strip to the window.

If you have a pointer to an already opened window, my_window, and a pointer to the first Menu structure in your list, my_first_menu, you attach the menu strip like this:

```
SetMenuStrip( my_window, my_first_menu );
```

If you want to change a menu strip which is already attached to a window you first need to remove it. You do it by calling the function ClearMenuStrip(). You can then change what ever you want to change, and attach it once again.

To remove a menu strip from a window, you only need a pointer to that window, eg my_window, and then call the function ClearMenuStrip() like this:

```
ClearMenuStrip( my_window );
```

If a menu strip is connected to a window, and you want to close that window, you should first remove the menu. If your program does not remove the menu strip before it closes the window, there is a risk that the user has activated the menu, and you would end up with a nice system crash. So, to be on the safe side, always call the function ClearMenuStrip() before you close a window which has a menu strip connected to it.

7.7 SPECIAL IDCMP FLAGS

There exist two special IDCMP flags which are close related to menus. They are MENUPICK and MENUVERIFY. The IDCMP flag MENUPICK tells your program that the user has picked zero or more items from a menu. MENUVERIFY allows your program to finish off something before the menu is displayed, and you can even abort the whole menu operation if you want.

7.7.1 MENUPICK

If you want your program to receive a message every time the user has picked zero or more items/subitems from the menu, you need to set the flag MENUPICK in the IDCMPFlags field in your NewWindow structure. You can then handle this message as normal IDCMP messages. (See chapter 8 IDCMP for more information about IDCMP messages.)

Once you receive a MENUPICK message you need to check if any items were selected. The Code field of the IntuiMessage structure will contain a special code number ("Menu Number") which we can examine in order to see which item was selected. The menu number is equal to MENUNULL if no item was selected. Use the function ItemAddress() with a pointer to the menu strip and the menu number as parameters, and it will return the address of the selected item.

The problem which will arise is that the user can select several items during one single menu operation, and your program needs to check them all. There is however an easy solution for this problem. If an item was selected, we can examine the NextSelect field of that item's structure, and that number is either MENUNULL (no more items selected), or another special menu number. (If it was not equal to MENUNULL you should again call the function ItemAddress() so you get a pointer to the second item's structure, and you should then examine the NextSelect field of that structure and so on...).

Here is a fragment of a program which handles MENUPICK event:

```
if( class == MENUPICK )
{
  /* The user has released the right mouse button. */

  /* The variable menu_number is initialized with the code */
  /* value:                                                */
  menu_number = code;

  /* As long as menu_number is not equal to MENUNULL we */
  /* stay in the while loop: */
  while( menu_number != MENUNULL )
  {
    /* Get the address of the item: */
    item_address = ItemAddress( &first_menu, menu_number );

    /* Process this item... */

    /* Get the following item's menu number: */
    menu_number = item_address->NextSelect;
  }

  /* All items which were selected has been processed! */
}
```

If you have a menu number you can use Intuitions three macros to check which menu/item/subitem was selected. They all return 0, if it was the first menu/item/subitem in the list, 1 if it was the second and so on.

`MENUNUM( menu_number );` Is equal to 0 if the item was selected from the first menu, 1 if it was from the second and so on. It is equal to NOMENU if the user did not select any item from the menus.

`ITEMNUM( menu_number );` Is equal to 0 if the firs item was selected in this item list, 1 if it was the second and so on. It is equal to NOITEM if the user did not select any item.

```
SUBNUM( menu_number );  Is equal to 0 if it the firs subitem
                        was selected in this subitem list, 1 if
                        it was the second and so on. It is
                        equal to NOSUB if the user did not
                        select any subitem.
```

7.7.2 MENUVERIFY

The MENUVERIFY message tells your program that the user has pressed the right mouse button, and a menu will be activated. However, the menu strip will first be activated when you program has replied, and you can therefore, thanks to this warning, wait with the menu event for a while, or even stop it totally, if necessary.

If you want your program to receive a message every time the a menu will be activated you need to set the flag MENUVERIFY in the IDCMPFlags field in your NewWindow structure. The MENUVERIFY flag acts like the REQVERIFY flag for requesters.

Here is an example on how your program should handle this verification message, and how you can cancel a menu operation:

```
/* Wait until we receive one or more messages: */
Wait( 1 << my_window->UserPort->mp_SigBit );

/* As long as we can collect messages successfully, we will */
/* stay in this while loop:                                 */
while(my_message = GetMsg( my_window->UserPort ));
{
  /* After we have collected the message we can read it,  */
  /* and save any important values which we maybe want to */
  /* check later:                                         */
  class = my_message->Class;
  code = my_message->Code;


  /* Before we reply we need to see if we have received a */
  /* MENUVERIFY message:                                  */

  if( class == MENUVERIFY )
  {
    /* Yes, we have received a MENUVERIFY message!          */
    /* The user wants to activate a menu, but the problem   */
    /* is that we do not know if it is our window's menu    */
    /* that will be activated, or some other window's menu. */
    /* We can however check it by examining the Code field  */
    /* of the message. If it is equal to MENUWAITING, it    */
    /* means that it is not your window's menu that will be */
    /* activated, but if Code is equal to MENUHOT it means  */
    /* it is is your window's menu that will be activated!  */

    if( code == MENUWAITING )
```

```
{
  /* It is not your window's menu that will be activated! */

  /* Your program can take a pause if necessary. You  */
  /* maybe want to finish of with some drawings, so   */
  /* your program does not trash any menus. This is   */
  /* especially important if you are using the low-   */
  /* level graphics routines since they do not bother */
  /* about, menus etc, and will draw over and destroy */
  /* anything in their way.                           */

  /* Once the program is ready it should reply the */
  /* message, and the menu will be activated. */
}
else
  if( code == MENUHOT )
  {
    /* It is your window's menu that will be activated!  */

    /* You can now take a pause and finish of with       */
    /* something before you let Intuition activate the   */
    /* menu, or you can even stop the whole menu         */
    /* operation if necessary. If you are writing a      */
    /* paint program you maybe only want the user to be  */
    /* able to activate the menu if the pointer is at    */
    /* the top of the display. That would mean that the  */
    /* user can draw with the right mouse button, and    */
    /* when the user wants to make a menu choice, he/    */
    /* she simply moves the pointer to the top of the    */
    /* display, and then presses the right mouse button. */

    /* We will now check if the pointer is somewhere at  */
    /* the top of the display:                           */
    if( my_window->MouseY < 10)
    {
      /* The Y coordinate of the pointer is at least */
      /* less than 10 lines below the TopEdge of the */
      /* window.                                     */

      /* The menu operation should continue as soon  */
      /* as possible!                                */
    }
    else
    {
      /* The pointer is below the Title bar of the */
      /* window! Cancel the whole menu operation!  */

      /* To cancel a menu operation you need to change   */
      /* the Code field to MENUCANCEL. IMPORTANT! Do not */
      /* change the code variable since it is just a     */
      /* copy of the real Code value. What we need to do */
      /* is to change the real value, and that is still  */
      /* OK since we have not replied yet.               */

      my_message->Code=MENUCANCEL;
    }
  }
```

```
  }

  /* After we have read the message we reply as fast as    */
  /* possible: REMEMBER! Do never try to read or change a  */
  /* message after you have replied! Some other process is */
  /* maybe using it.                                       */
  ReplyMsg( my_message );
```

If you set the RMBTRAP flag in your NewWindow's Flags field you tell Intuition that you do not want to use any menus at all for this window. Your program could then use the right mouse button for something else. (See chapter 8 IDCMP for more information about RMBTRAP.)

7.8 MENU NUMBERS

Here is a description of how the "menu numbers" actually work. (This can change later so do not calculate anything yourself, let Intuition's macros do it instead.)

The menu number is a 16-bit variable, where the first five bits are used for the menus, the following six bits are used for the items, and the last five bits are used for the subitems:

```
c c c c c b b b b b b a a a a a
|         |           |
|         |           > Menu
|         > Item
> SubItem
```

This means the Intuition can "only" handle up to 31 Menus, each with up to 63 Items, where each item can have up to 31 SubItems. I do not think this will be any restrictions for your program.

If a field is equal to 0 it means the first menu/item/subitem, if it is equal to 1 it means the second and so on. If all bits are on it means that no menu/item/subitem was selected. See following example:

The menu number 0xF803 means:

No SubItem, the first item, from the fourth menu.
0xF803 (h) = 1111 1000 0000 0011 (b)

```
Menu number  11111 000000 00011  (b)
             |     |      |
             |     |      >  00011 =  3 = fourth Menu
             |     >        000000 =  0 = first Item
             >               11111 = 31 = NOSUB
```

To calculate this you should use Intuition's macros:

```
MENUNUM( 0xF803 ) == 3  the fourth menu.
ITEMNUM( 0xF803 ) == 0  the first item.
SUBNUM( 0xF803 )  == 31 (NOSUB).
```

It sometimes happens that you want to go the other way, and calculate your own menu number. You should then use the macros SHIFTMENU, SHIFTITEM and SHIFTSUB which does the opposite of MENUNUM etc. If you are going to use the functions OnMenu() and OffMenu() (explained later), you need to send a menu number as one of the parameters. If you for example want to "ghost" the second item in the third menu, you calculate the menu number like this:

```
menu_number = SHIFTMENU(2) + SHIFTITEM(1) + SHIFTSUB(NOSUB);
              [third menu]   [second item]  [no subitem]
```

7.9 FUNCTIONS

Here are some commonly used function which affects menus:

SetMenuStrip()

This function connects a menu strip to a window. Remember that the window must have been opened before you may connect a menu strip to that window.

Synopsis: SetMenuStrip(my_window, my_menu);

my_window: (struct Window *) Pointer to the window which the menu strip should be connected to.

my_menu: (struct Menu *) Pointer to the first Menu structure in the menu strip.

ClearMenuStrip()

This function removes a menu strip from a window. Remember to always remove the menu strip before you close the window, or changes the menu strip.

Synopsis: ClearMenuStrip(my_window);

my_window: (struct Window *) Pointer to the window which menu strip should be removed.

ItemAddress()

This function returns a pointer to the Menu or Item structure which is specified by the menu number.

Synopsis: ItemAddress(my_menu, menu_number);

my_menu: (struct Menu *) Pointer to the first Menu structure in the menu strip.

menu_number: (USHORT) This menu number specifies a subitem/ item/menu.

OffMenu()

This function can disable a subitem, an item or even a whole menu. The image or text of the disabled items etc will be "ghosted", and the user can not select them.

Synopsis: OffMenu(my_window, menu_number);

my_window: (struct Window *) Pointer to the window which the menu strip is connected to.

menu_number: (USHORT) This menu number specifies what should be disabled. Use the macros SHIFTMENU, SHIFTITEM and SHIFTSUB to calculate the correct menu number. If you just specify a menu, all items to that menu will be disabled. If you specify a menu and an item, that item will be disabled, and so all subitems connected to it if there are any.

OnMenu()

This function can enable a subitem, an item or even a whole menu. The image or text of the enabled items etc, will become normal (not "ghosted") and the user can now select them.

Synopsis: OnMenu(my_window, menu_number);

my_window: (struct Window *) Pointer to the window which the menu strip is connected to.

menu_number: (USHORT) This menu number specifies what should be enabled. Use the macros SHIFTMENU, SHIFTITEM and SHIFTSUB to calculate the correct menu number. If you just specify a menu, all items to that menu will be enabled. If you specify a menu and an item, that item will be enabled, so all subitem connected to it if there are any.

7.10 MACROS

Here are some commonly used macros:

These three macros takes a "menu number" as the only parameter, and examines it. These are very handy if you have received a menu number, and you want to check which menu and which item was selected etc:

`MENUNUM( menu_number );` Is equal to 0 if the menu number specifies the first menu, 1 if it specifies the second, and so on. It is equal to NOMENU if it does not specify any menus at all.

`ITEMNUM( menu_number );` Is equal to 0 if the menu number specifies the first item, 1 if it specifies the second, and so on. It is equal to NOITEM if it does not specify any items at all.

`SUBNUM( menu_number );` Is equal to 0 if the menu number specifies the first subitem, 1 if it specifies the second, and so on. It is equal to NOSUB if it does not specify any subitems at all.

These three macros does the opposite of the first three. With help of these macros you can make your own menu number just by specifying which menu, item and subitem you want.

`SHIFTMENU( menu );` Is equal to a menu number which specifies this menu. ("menu" is a value between 0 and 30, or NOMENU if no menu should be specified.)

`SHIFTITEM( item );` Is equal to a menu number which specifies this item. ("item" is a value between 0 and 62, or NOITEM if no item should be specified.)

`SHIFTSUB( sub );` Is equal to a menu number which specifies this subitem. ("sub" is a value between 0 and 30, or NOSUB if no subitem should be specified.)

7.11 EXAMPLES

The printed version of these examples can be found in Appendix A (Part IV: Appendices).

Example1

This program opens a normal window to which we connect a menu strip. The menu consists of four items: Plain, Bold, Underlined and Italic. The user can select either Plain or a combination of the other styles. (If the user selects Plain all other modes will be mutual excluded, but if the user on the other hand selects Bold, Underlined or Italic, only the Plain option will be mutual excluded.

This example also shows how a program should handle the IDCMP flags, and how to collect several messages from one single menu event.

Example2

This example is very similar to Example1, but we have this time put the edit styles in a subitem box which is connected to the one and only item box called "Style".

Example3

This example is very similar to Example2, but the user can this time also access the subitems from the keyboard. For example, to select Bold the user only needs to press the right Amiga key [A] together with the "B" key.

Example4

This program opens a normal window to which we connect a menu strip. The menu consists of two items: Readmode and Editmode. The readmode item is selected and ghosted, and when the user selects the editmode item, it will become disabled (ghosted) while the readmode item will be enabled (not ghosted). This means that if the program is in "readmode", the user should only be able to chose the "editmode", and v.v. The purpose with this program is to show how you can use the OnMenu and OffMenu functions in order to make an "user-friendly interface".

Example5

Exactly as Example1 except that we have changed Intuition's checkmark to our own customized "arrow".

Example6

This program opens a normal window to which we connect a menu strip. The menu consists of six small dices which are all action items. This example shows how you can use Images inside a menu.

Example7

This program opens a normal window to which we connect a menu strip. The menu consists of one small action item with two images.

Example8

Same as Example1 except that we this time will verify any menu operations. If the user tries to activate this program's menu we check if the position of the pointer is somewhere at the top of the window (less than 10 lines down). In that case the menu operation will continue as normal, otherwise we cancel the menu operation.

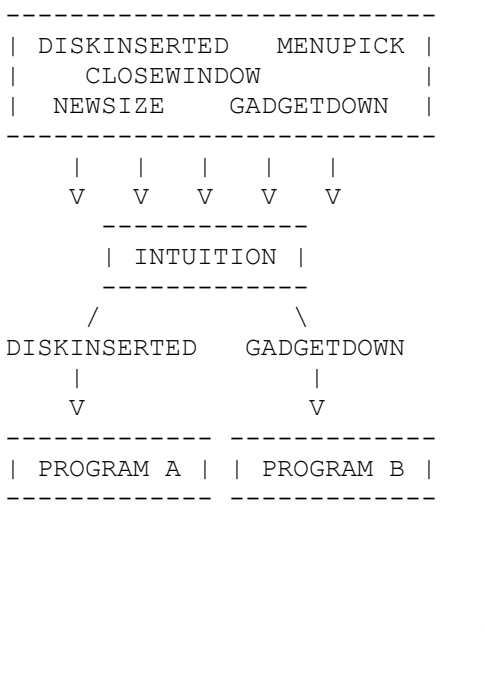
8 IDCMP

8.1 INTRODUCTION

We have in the last four chapters talked about IDCMP, but have never actually explained what it is. But in this chapter we will at last satisfy your hunger for more information. IDCMP stands for Intuition's Direct Communications Messages Ports (nice), and is the most commonly used way of receiving messages from Intuition, and the only way to communicate to Intuition.

8.2 IDCMP PORTS

When something happens inside the Amiga, a disk is inserted, a gadget is selected etc, a message is automatically created. Intuition will then examine the messages and check if your program is interested to hear about it. (Every window has a list of none or more IDCMP flags which tells Intuition what this window is interested of.) If a program is interested of the message, that program will receive a special message (IntuiMessage), which contains interesting information about the message.



Messages created inside the Amiga. For example, a disk is inserted, a menu is activated, a window is closed etc...

Intuition examines every message and checks if any program is interested of it.

If program A is interested of DISKINSERTED messages, Intuition will pass that one along to program A. If program B is interested in GADGETDOWN messages program B will receive a message every time a gadget is selected in program B's window.

All other messages which no program were interested in will be thrown away.

Intuition will always start to examine the active window

first, and many IDCMP messages will probably be "swallowed". If program B is active, and Intuition has found a GADGETDOWN message, that message would be passed to program B, and all other programs would never hear about it.

Some messages are important for all programs, such as DISKINSERTED, and will be passed to ALL "interested" windows. (If a window has its IDCMPFlags field set to DISKINSERTED, that window is interested about disk inserted messages.)

8.3 HOW TO RECEIVE IDCMP MESSAGES

If you want to receive messages from Intuition you need to:

1. Open an IDCMP port.
2. Wait for messages.
3. Collect the messages.
4. Examine them.
5. Reply. (Tell Intuition that you have read the message)

8.3.1 OPEN IDCMP PORTS

The IDCMP port can be automatically allocated by Intuition when you open a window. You only need to specify in the NewWindow structure's IDCMPFlags field which messages you want to receive from Intuition, and the rest is done for you. For example, if you open a window with the IDCMPFlags field set to GADGETDOWN, a port would be allocated for you, and your program would receive a message every time a gadget has been selected.

If you already have opened a window you can later change the IDCMP ports by calling the function ModifyIDCMP().

8.3.2 WAIT FOR MESSAGES

Once an IDCMP port has been opened your program only need to wait until one or more messages arrives. There are two different ways of waiting for messages:

1. The Passive Way. Your program is halted and will only wake up when a message arrives. This will increase the speed of other applications since the CPU does not need to bother about your program as long as it is sleeping. Use the function Wait().

2. The Active Way. Your program tries to collect a message, and if it could not find any message it tries again. This is necessary if you want your program to continue doing something, and not stop waiting for a message to arrive.

8.3.3 COLLECT MESSAGES

When you collect a message you should use the function GetMsg(). It will return a pointer to an IntuiMessage structure, which contains all important information about the message, or NULL if it could not collect any message.

8.3.4 EXAMINE THE MESSAGE

Once you have collected a message successfully you can examine the IntuiMessage structure. The IntuiMessag structure look like this:

```
struct IntuiMessage
{
  struct Message ExecMessage;
  ULONG Class;
  USHORT Code;
  USHORT Qualifier;
  APTR IAddress;
  SHORT MouseX, MouseY;
  ULONG Seconds, Micros;
  struct Window *IDCMPWindow;
  struct IntuiMessage *SpecialLink;
};
```

ExecMessage: Used by the Exec so do not touch it.

Class: Contains the IDCMP flag.

Code: Contains some special values which is closely related to the IDCMP flag (Class). For example, if you receive a MENUVERIFY message you can examine the Code field to see if it is your program's Menu (Code == MENUHOT) or some other program's Menu (Code == MENUWAITING) that will be displayed. If you on the other hand receives RAWKEY or VANILLAKEY messages the Code field will contain the key code and so on.

Qualifier: If your program receives RAWKEY messages (untranslated keycodes), this field contains information like if the SHIFT or CTRL key was

pressed or not. (A copy of the ie_Qualifier.)

MouseX: X position of the pointer relative to the top left corner of your window.

MouseY: Y position of the pointer relative to the top left corner of your window.

Seconds: Copy of the system clock's seconds.

Micros: Copy of the system clock's microseconds.

IAddress: Pointer to that object that created the message. For example, if you receive a GADGETDOWN message, this field contains a pointer to the Gadget that was selected.

IDCMPWindow: Pointer back to the window that sent this message.

SpecialLink: Used by Exec and Intuition so do not touch it.

The best way to examine this structure is to copy all important values and then reply as fast as possible. (Intuition will be slowed down if you do not reply fast.) You can then (after you have replied) check the copied values. (Remember! Do not use the IntuiMessage structure any more once you have replied.)

8.3.5 REPLY

Once your program has read everything it is interested of it should reply back to Intuition by calling the function ReplyMsg(). This tells Intuition that you have finished with reading the message.

Important! Once you have replied you may NOT examine/change the IntuiMessage structure any more. Some other program or Intuition may use it!

8.3.6 EXAMPLE

Here is an example of how your program should collect and process IDCMP messages: (This is the Passive Way of collecting messages.)

```
/* 1. Wait until a message arrive: */
Wait( 1 << my_window->UserPort->mp_SigBit );

/* (my_window is a pointer to a window.)     */
```

```
/* (Do not bother to much about the funny formula.) */


/* 2. Try to collect a message: */
my_message = GetMsg( my_window->UserPort );


/* Could we collect a message? (If message == NULL we have */
/* not collected any message.                              */
if( my_message )
{
  /* YES! We have collected a message successfully. */


  /* 3. We should now examine the IntuiMessage structure, */
  /*    and save any important values. (If we save the    */
  /*    information we can reply as soon as possible, and */
  /*    later examine the values.)                        */
  class   = my_message->Class;
  code    = my_message->Code;
  address = my_message->IAddress;


  /* 4. We should now reply since we have finished reading */
  /*    the IntuiMessage structure. (Your program should   */
  /*    always reply as fast as possible.)                 */
  ReplyMsg( my_message );


  /* 5. We can now examine what the message actually was */
  /*    about: (We have replied so we may NOT examine    */
  /*    the IntuiMessage structure any more, but we may  */
  /*    of course check the variables: class, code and   */
  /*    address.)                                        */
  switch( class )
  {
    case GADGETDOWN: /* A gadget has been pressed.       */
    case GADGETUP:   /* A gadget has been released.      */
    case MENUPICK:   /* A menu item has been selected... */
    /* And so on...                                      */
  }
}
```

This example is very fine, but what happens if two messages arrives at the same time? Well, the first message would be dealt with as normal, but the second message would never be processed. After we have waited for a message we should actually be prepared to process many messages. We should therefore use a while-loop, and stay in the loop as long as we can collect messages successfully. Once we could not collect any more we should put our task to sleep again.

Here is a modified, and better, version of the example:

```
/* 1. Wait until a message arrive: */
Wait( 1 << my_window->UserPort->mp_SigBit );


/* As long as we can collect messages successfully we stay */
/* in the while-loop: */
while( my_message = GetMsg( my_window->UserPort ) )
{
  /* The rest is as normal... */


  /* 3. We should now examine the IntuiMessage structure, */
  /*    and save any important values. (If we save the    */
  /*    information we can reply as soon as possible, and */
  /*    later examine the values.)                        */
  class   = my_message->Class;
  code    = my_message->Code;
  address = my_message->IAddress;


  /* 4. We should now reply since we have finished reading */
  /*    the IntuiMessage structure. (Your program should   */
  /*    always reply as fast as possible.)                 */
  ReplyMsg( my_message );


  /* 5. We can now examine what the message actually was */
  /*    about: (We have replied so we may NOT examine    */
  /*    the IntuiMessage structure any more, but we may  */
  /*    of course check the variables: class, code and   */
  /*    address.)                                        */
  switch( class )
  {
    case GADGETDOWN: /* A gadget has been pressed.       */
    case GADGETUP:   /* A gadget has been released.      */
    case MENUPICK:   /* A menu item has been selected... */
    /* And so on...                                      */
  }
}
```

8.4 IDCMP FLAGS

We have used several IDCMP flags in the last chapter, such as GADGETDOWN, MENUPICK, REQVERIFY etc, but here is the total list of all IDCMP flags:

Here are the three IDCMP flags which are closely related to gadgets:

GADGETDOWN: Your program will this message if a gadget has been selected. (Note, the gadget must have the

GADGIMMEDIATE flag set in its Activation field. If not there will be no GADGETDOWN message created.)

GADGETUP: Your program will this message if a gadget has been released. (Note, the gadget must have the RELVERIFY flag set in its Activation field. If not there will be no GADGETUP message created.)

CLOSEWINDOW: Your program will receive this message if the user selects the Close Window gadget (System Gadget). Remember, the window will not be automatically closed. It is up to your program if it want to close it or not.

Here are the three IDCMP flags which are closely related to requesters:

REQSET: Your program will receive this message if a requester has been opened in your window. This is very handy way to detect if a Double-menu requester has been activated in your window. (You will receive a message both when a DM-requester as well as a normal requester has been activated.)

REQCLEAR: This message tells you that a requester has been cleared from your window. Note, you will receive this message only when all requesters in your window has been cleared.

REQVERIFY: Your program will receive this message if a requester is going to be activated. However, the requester will first be displayed once you have replied, and this will therefore allow you to finish of with something before the requester will be activated. Note, your program will only receive this message when the first requester is opening in the window. Intuition will then assume that you have stopped any output to that window.

Here are the two IDCMP flags which are closely related to menus:

MENUPICK: Your program will receive this message if the user has pressed the right mouse button. (Note, It does not necessarily mean that an item has been selected.) The Code field of the message will contain the menu number. If no item was selected Code will be equal to MENUNULL. Use the Macros described in chapter 7 MENUS, to check which menu/item/subitem was selected.

MENUVERIFY: Your program will receive this message if a

menu is going to be activated. The menu will, however, first be displayed once you have replied, and this will therefore allow you to finish of with something before the menu will be opened.

Check if the Code field of the message is equal to MENUHOT which means it is your menu that will be displayed, or if Code is equal to MENUWAITING, which means that some other window's menustrip will be displayed. You can even cancel the whole menu-operation if you want by changing the Code field to MENUCANCEL before you reply.

Here are the three IDCMP flags which are closely related to the mouse:

MOUSEBUTTONS: Your program will receive this message if any of the mouse buttons have been pressed or released. Examine the Code field of the message to see if it is equal to:

SELECTDOWN: The left mouse button has been pressed.
SELECTUP: The left mouse button has been released.
MENUDOWN: The right mouse button has been pressed.
MENUUP: The right mouse button has been released.

Important! If the user presses the left mouse button while the pointer is somewhere on a gadget's select box, your program will NOT receive any message. Remember also that if the user presses the right mouse button a menu strip will be displayed, and your program will not receive any MENUDOWN message. You need to set the RMBTRAP flag in the NewWindow structure's Flags field in order to receive any right mouse buttons event. (No menu can then be used for that window.)

MOUSEMOVE: Your program will receive this message if the mouse has been moved. Note, your window needs to have the REPORTMOUSE flag set in the NewWindow structure's Flags field, or a gadget in the window needs to have the FOLLOWMOUSE flag set in the Gadget structure's Activation field.

(Be prepared to receive many messages!)

DELTAMOVE: Same as MOUSEMOVE except that the movements will be reported as delta movements instead of

absolute positions. Note, your program will continue to receive delta movements even if the pointer has reached the border of the display.

(Be prepared to receive many messages!)

Here are the five IDCMP flags which are closely related to windows:

NEWSIZE: Your program will receive this message if the user has resized the window. Check the windows' Window structure's Width and Height fields to discover the new size.

SIZEVERIFY: Your program will receive this message if the user tries to resize the window. However, Intuition will change the size first when your program has replied, and this allows you to finish of with something before the window will change size.

REFRESHWINDOW: Your program will receive this message if the window needs to be redrawn. This applies only to SIMPLE_REFRESH and SMART_REFRESH windows. Remember, if you receive a REFRESHWINDOW event, you need to call the functions BeginRefresh() and EndReresh() even if you do not redraw anything.

ACTIVEWINDOW: Your program will receive this message if your window is activated.

INACTIVEWINDOW: Your program will receive this message if your window is deactivated.

Other IDCMP flags:

RAWKEY: Your program will receive this message every time the user presses a key. The Code field of the message will contain the raw keykodes (see Appendix C), and the Qualifier field of the message contains the qualifier (SHIFT, CTRL etc).

VANILLAKEY: Your program will receive this message every time the user presses a key. The Code field of the message will contain the ASCII character (see Appendix D).

DISKINSERTED: Your program will receive this message if a disk is inserted. All interested programs will hear about it.

DISKREMOVED: Your program will receive this message if a disk is removed. All interested programs will

hear about it.

NEWPREFS: If the the system Preferences is changed (by the user or another program), your program will receive a NEWPREFS message. Call the function GetPrefs() to get the new system Preferences. (See chapter 9 MISCELLANEOUS for more information about Preferences etc.)

INTUITICKS: Your program will receive this message ten times a second (approximately). However, if you have not replied on the first INTUITICKS message, the next message will not be sent.

WBENCHMESSAGE: If the workbench is opened or closed by a program (calling the functions OpenWorkBench()/ CloseWorkBench() your program will receive a WORKBENCHMESSAGE. The Code field of the message will be equal to WBENCHOPEN if the WorkBench was opened, or WBENCHCLOSE if the workbench was closed.

8.5 FUNCTIONS

GetMsg()

This function tries to get a message from a message port.

Synopsis: my_message = GetMsg(my_message_port);

my_message: (struct Message *) Pointer to a Message structure, in this case a pointer to an IntuiMessage structure, or NULL if no message was collected.

my_message_port: (struct MsgPort *) Pointer to an MsgPort. If you have opened a window, you can find your window's message port in the Window structure. (my_window->UserPort)

ReplyMsg()

This function tells Intuition that you have finished reading the message. Remember, once you have replied you may not examine or change the IntuiMessage structure any more.

Synopsis: ReplyMsg(my_message);

my_message: (struct Message *) Pointer to a Message structure, in this case a pointer to an IntuiMessage structure.

ModifyIDCMP()

This function changes the Window structure's IDCMPFlags field.

Synopsis: ModifyIDCMP(my_window, IDCMPFlags);

my_window: (struct Window *) Pointer to an already opened window.

IDCMPFlags: (long) None or more IDCMP flags.

If you call this function with no IDCMP flags set, the window's IDCMP Ports will be closed. On the other hand, if you call this function, with one or more IDCMP flags set, a Port will be, if necessary, opened for you.

8.6 EXAMPLES

See the examples in chapter 4 (GADGETS) to 7 (MENUS). They will explain how to use the IDCMP flags: GADGETDOWN, GADGETUP, CLOSEWINDOW, REQSET, REQCLEAR, REQVERIFY, MENUPICK and MENUVERIFY.

The printed version of these examples can be found in Appendix A (Part IV: Appendices).

Example1

This program explains how to use the IDCMP flag MOUSEBUTTONS.

Example2

This program explains how to use the IDCMP flag MOUSEMOVE.

Example3

This program explains how to use the IDCMP flags: NEWSIZE, ACTIVEWINDOW and INACTIVEWINDOW.

Example4

This program explains how to use the IDCMP flag SIZEVERIFY.

Example5

This program explains how to use the IDCMP flag RAWKEY.

Example6

This program explains how to use the IDCMP flag VANILLAKEY.

Example7

This program explains how to use the IDCMP flags:
DISKINSERTED and DISKREMOVED.

Example8

This program explains how to use the IDCMP flag INTUITICKS.

9 MISCELLANEOUS

9.1 INTRODUCTION

This is the last chapter about Intuition, and we will therefore finish of by discussing some special features.

9.2 MEMORY

If you write C programs you will often need to allocate some memory. This process is normally referred as "dynamic memory allocation" and is used whenever your program needs some more memory while it is running.

When you allocate memory you need to specify how much you want, and what type of memory (Fast or Chip). "Exec" will then try to allocated the memory, and will return a pointer to the memory, or NULL if it could not find a block big enough.

Once you have allocated the memory, you can use it as much as you want. You just have to remember that before the program terminates it must have deallocated all allocated memory. If you forget to deallocate the memory when you leave, it will be lost, and the only way to get it back is to reboot.

9.2.1 ALLOCATE MEMORY

You usually use the function AllocMem() if you want to allocate memory. You only need to specify how much memory you need, and what type of memory (Chip, Fast etc), and AllocMem() will return a pointer to the new memory, or NULL if no memory could be allocated.

Synopsis: memory = AllocMem(size, type);

memory: (void *) Pointer to the new allocated memory, or NULL if no memory could be allocated. Remember! Never use memory which you have not successfully allocated.

size: (long) The size (in bytes) of the memory you want. (AllocMem() always allocates memory in multiples of eight bytes. So if you only ask for 9 bytes, Exec would actually give you 16 Bytes (2*8).)

type: (long) You need to choose one of the three following types of memory (see chapter 0 INTRODUCTION for more

information about Chip and Fast memory). (The flags are declared in the header file "exec/memory.h".)

MEMF_CHIP Chip memory. This memory can be accessed by both the main processor, as well as the Chips. Graphics/Sound data MUST therefore be placed in Chip memory. If it does not matter what type of memory you get (Fast or Chip), you should try to allocate Fast memory before you allocate Chip memory. (Chip memory is more valuable than Fast memory.)

MEMF_FAST Fast memory. This memory can only be accessed by the main processor. (Graphics and Sound data can NOT be stored in Fast memory, use Chip memory.) This memory is normally a little bit faster than Chip memory, since only the main processor is working with it, and it is not disturbed by the Chips.

MEMF_PUBLIC If it does not matter what type of memory you get (you do not intend to use the memory for Graphics/Sound data), you should use Fast memory. However, all Amigas do not have Fast memory, since you need to by a memory expansion in order to get it. If want to tell Exec that you would like to use Fast memory if there is any, else use Chip memory, you should ask for MEMF_PUBLIC.

If you want the allocated memory to be cleared (initialized to zeros), you should set the flag MEMF_CLEAR.

If you need to allocate 150 bytes of Chip memory, you would write:

```
/* Declare a pointer to the memory you are going to allocate: */
CPTR memory; /* CPTR means memory pointer, declared in the    */
             /* header file "exec/types.h". "exec/types.h" is */
             /* automatically included when you include the   */
             /* header file "intuition/intuition.h".          */

/* Allocate 150 bytes of Chip memory: */
memory = (CPTR) AllocMem( 150, MEMF_CHIP );

/* Have we allocated the memory successfully? */
if( memory == NULL )
  /* ERROR! Have not allocated any memory! */
```

9.2.2 DEALLOCATE MEMORY

All memory that has been allocated MUST be deallocated before your program terminates. If you forget to deallocate memory which you have allocated, that memory would be forever lost. The only way of freeing such memory is to reboot. (Not to be recommended. We are not Demo-writers, are we?)

You free allocated memory by calling the function FreeMem():

Synopsis: FreeMem(memory, size);

memory (void *) Pointer to some memory which has previously been allocated. Remember! Never deallocate memory, which you have not allocated, and never use memory which has been deallocated.

size (long) The size (in bytes) of the memory you want to deallocate.

To deallocate the 150 bytes of Chip memory we have previously allocated, we write:

```
/* Free 150 bytes of memory: */
FreeMem( memory, 150 );
```

9.2.3 REMEMBER MEMORY

If you allocate memory several times, and in different quantities, it can be very hard to remember to deallocate all memory correctly. However, do not despair. Intuition have a solution for you. You can use the function AllocRemember() which calls the function AllocMem(), but will also allocate memory for a Remember structure which is automatically initialized for you. Every time you allocate memory with the AllocRemember() function, the Remember structures are linked together, and the information about the memory (size and position) is remembered. Once you want to deallocate the memory, you only need to call the function FreeRemember(), and all memory is deallocated for you automatically.

The Remember structure look like this:

```
struct Remember
{
  struct Remember *NextRemember;
  ULONG RememberSize;
  UBYTE *Memory;
};
```

The AllocRemember() function is called like this: (Very similar to the AllocMem() function.)

Synopsis: memory = AllocRemember(remember, size, type);

memory: (char *) Pointer to the new allocated memory, or NULL if no memory could be allocated. Remember! Never use memory which you have not successfully allocated.

remember: (struct Remember **) Address of a pointer to a Remember structure. Before you call the AllocRemember() function for the first time you should set this pointer to NULL. (Note that it is a pointer to a pointer!)

size: (long) The size (in bytes) of the memory you want. (AllocMem() always allocates memory in multiples of eight bytes. So if you only ask for 29 bytes, Exec would actually give you 32 Bytes (4*8).)

type: (long) You need to choose one of the three following types of memory (see chapter 0 INTRODUCTION for more information about Chip and Fast memory):

MEMF_CHIP Chip memory. This memory can be accessed by both the main processor, as well as the Chips. Graphics/Sound data MUST therefore be placed in Chip memory. If it does not matter what type of memory you get (Fast or Chip), you should try to allocate Fast memory before you allocate Chip memory. (Chip memory is more valuable than Fast memory.)

MEMF_FAST Fast memory. This memory can only be accessed by the main processor. (Graphics and Sound data can NOT be stored in Fast memory, use Chip memory.) This memory is normally a little bit faster than Chip memory, since only the main processor is working with it, and it is not disturbed by the Chips.

MEMF_PUBLIC If it does not matter what type of memory you get (you do not intend to use the memory for Graphics/Sound data), you should use Fast memory. However, all Amigas do not have Fast memory, since you need to by a memory expansion in order to get it. If want to tell Exec that you would like to use Fast memory if there is any, else use Chip memory, you should ask for MEMF_PUBLIC.

If you want the allocated memory to be cleared (initialized to zeros), you should set the flag MEMF_CLEAR.

When you want to deallocate the memory you only need to call the function FreeRemember() function. You can if you want, deallocate only the Remember structures, and take care of the deallocation yourself, but be careful. The FreeRemember() function is called like this:

Synopsis: FreeRemember(remember, everything);

remember: (struct Remember **) Address of a pointer to the first Remember structure (initialized by the AllocRemember() function). (Note that it is a pointer to a pointer!)

everything: (long) A boolean value. If everything is equal to TRUE, all memory (both the allocated memory and the Remember structures) are deallocated. However, if everything is equal to FALSE, only the Remember structures are deallocated, and you have to deallocate the memory yourself.

Here is a short example which shows how to use the AllocRemember(), and the FreeRemember() functions:

```
struct Remember *remember = NULL;
CPTR memory1, memory2, memory3;

/* Allocate 350 bytes of Chip memory, which is cleared: */
memory1 = AllocRemember( &remember, 350, MEMF_CHIP|MEMF_CLEAR );

if( memory1 == NULL )
  Clean up and leave;

/* Allocate 900 bytes of memory (any type, Fast if possible): */
memory2 = AllocRemember( &remember, 900, MEMF_PUBLIC );

if( memory2 == NULL )
  Clean up and leave;

/* Allocate 100 bytes of Chip memory: *
memory3 = AllocRemember( &remember, 100, MEMF_CHIP );

if( memory3 == NULL )
  Clean up and leave;

...


/* Deallocate all memory with one single call: */
FreeRemember( &remember, TRUE );
```

If we wanted, we could have deallocated all Remember structures, and then deallocate the memory ourself with help of the normal FreeMem() function:

```
/* Deallocate only the Remember structures: */
FreeRemember( &remember, FALSE );
FreeMem( memory1, 350 );
FreeMem( memory2, 900 );
FreeMem( memory3, 100 );
```

9.3 PREFERENCES

When the system starts up, Intuition is initialized to some default values. However, if there exist a file called: "system-configuration" in the "devs" directory on the system disk, that file is used to set the system values, instead of the default values.

The user can with the "Preferences" program change the system values, and/or the "system-configuration" file. If the user selects the "USE" command, only the system values are changed, while if the user selects the "SAVE" command, both the system values, as well as the "system-configuration" file are updated.

Your program can get a copy of the system values by calling the function GetPrefs(). If you want to get a copy of the default values you use the function GetDefPrefs(). When you get a copy of the Preferences data, you can decide how much data you want copied. You can therefore, if you want, only copy parts of the structure, and do not need to allocate memory for the whole structure. The most important values are because of that placed in the beginning of the structure. The Preferences structure look like this: (See file "intuition/preferences.h" for more information. It is on the fourth Lattice C disk, in the "Compiler_Headers" directory.)

```
struct Preferences
{
  BYTE FontHeight;
  UBYTE PrinterPort;
  USHORT BaudRate;
  struct timeval KeyRptSpeed;
  struct timeval KeyRptDelay;
  struct timeval DoubleClick;
  USHORT PointerMatrix[POINTERSIZE];
  BYTE XOffset;
  BYTE YOffset;
  USHORT color17;
  USHORT color18;
  USHORT color19;
  USHORT PointerTicks;
  USHORT color0;
  USHORT color1;
  USHORT color2;
  USHORT color3;
  BYTE ViewXOffset;
```

```
  BYTE ViewYOffset;
  WORD ViewInitX, ViewInitY;
  BOOL EnableCLI;
  USHORT PrinterType;
  UBYTE PrinterFilename[FILENAME_SIZE];
  USHORT PrintPitch;
  USHORT PrintQuality;
  USHORT PrintSpacing;
  UWORD PrintLeftMargin;
  UWORD PrintRightMargin;
  USHORT PrintImage;
  USHORT PrintAspect;
  USHORT PrintShade;
  WORD PrintThreshold;
  USHORT PaperSize;
  UWORD PaperLength;
  USHORT PaperType;
  UBYTE SerRWBits;
  UBYTE SerStopBuf;
  UBYTE SerParShk;
  UBYTE LaceWB;
  UBYTE WorkName[FILENAME_SIZE];
  BYTE RowSizeChange;
  BYTE ColumnSizeChange;
  UWORD PrintFlags;
  UWORD PrintMaxWidth;
  UWORD PrintMaxHeight;
  UBYTE PrintDensity;
  UBYTE PrintXOffset;
  UWORD wb_Width;
  UWORD wb_Height;
  UBYTE wb_Depth;
  UBYTE ext_size;
};
```

You call the function GetPrefs() like this:

Synopsis: pref = GetPrefs(buffer, size);

pref: (struct Preferences *) Pointer to your preferences. Same as your memory pointer (buffer), but is returned so you can check if you got a copy or not. If you could not get a copy of the preferences, the function returns NULL.

buffer: (struct Preferences *) Pointer to the memory buffer which should be used to store a copy of the preferences in.

size: (long) The number of bytes you want to copy to the buffer. Important, the buffer must be at least as big as the number of bytes you want to copy.

You call the function GetDefPrefs() like this:

Synopsis: pref = GetPrefs(buffer, size);

pref: (struct Preferences *) Pointer to the default preferences. If the function could not make a copy of the preferences, the function returns NULL.

buffer: (struct Preferences *) Pointer to the memory buffer which should be used to store a copy of the default preferences in.

size: (long) The number of bytes you want to copy to the buffer. Important, the buffer must be at least as big as the number of bytes you want to copy.

Here is an example on how to get a copy of the current preferences:

```
/* Declare a preferences structure: */
struct Preferences pref;

/* Try to get a copy of the current preferences: */
if( GetPrefs( &pref, sizeof(pref) ) == NULL )
  /* Could not get a copy of the preferences! */
```

Once you have got a copy of the current/default preferences we can start to check the settings.

You can, if you want, change some values, and then save them as new preferences. IMPORTANT! You should never mess with the settings since it is the user's own preferences, and should therefore NOT be changed unless the user really WANT to alter them.

You set new preferences by calling the function SetPrefs():

Synopsis: SetPrefs(pref, size, doit);

pref: (struct Preferences *) Pointer to your modified Preferences structure.

size: (long) The number of bytes you want to change.

doit: (long) Boolean value which if FALSE, changes the preferences, but will not send a NEWPREFS message. If doit is equal to TRUE, the settings will be changed, and a NEWPREFS message will be sent. As long as the user is changing the values, doit should be FALSE, but when the user has finished, set it to TRUE, and all programs will get a NEWPREFS message.

9.4 WARNINGS

There exist three levels of warnings you can give the user when something goes wrong:

1. If you just want to get the users attention about something, but nothing really dangerous has happened, you should call the function DisplayBeep(). It will flash the colours of the screen.

   Synopsis: DisplayBeep(screen);

   screen: (struct Screen *) Pointer to the screen, which colours you want to flash. If you have not opened a screen yourself (you are using the Workbench Screen), you can find a pointer to that screen in the Window structure: (my_window is a pointer to an opened window)
   DisplayBeep(my_window->WScreen);

2. If something important has happened, or if you want the user to take some action (select something) etc, open a Requester. (See chapter 5 REQUESTERS for more information.)

3. If something TERRIBLE is happening, the system is breaking down, open an Alert. (See chapter 6 ALERTS for more information.)

9.5 DOUBLE CLICK

There are five different mouse actions:

1. Press a button	The user can press on a button.
2. Release a button	The user can release a previously pressed button.
3. Click a button	Quickly pressing and releasing a button.
4. Double-click a button	Quickly pressing/releasing a button twice.
5. Drag	Hold a button pressed, while moving the mouse.

Use the IDCMP flags SELECTDOWN, SELECTUP, MENUDOWN, and MENUUP in order to receive mouse-buttons events. (See chapter 8 IDCMP for more information.)

If you want to check if the user double-clicked one of the mouse buttons, you can use the function DoubleClick(). You give the function the current as well as the previous time when the button was pressed, and it will check the preferences and return TRUE if the two button events happened within the time limit. (The user can change the time limit for double-clicking events.)

Synopsis: double = DoubleClick(sec1, mic1, sec2, mic2);

double: (long) If the two button events happened within the current time limit, the function will return TRUE, else it will return FALSE.

sec1: (long) Time (seconds) when the button was pressed for the first time.

mic1: (long) Time (micros) when the button was pressed for the first time.

sec2: (long) Current time (seconds).

mic2: (long) Current Time (micros).

Here is a short example on how to check double-click events:

```
struct IntuiMessage *message;
ULONG sec1, mic1, sec2, mic2;

... ... ...

if( message->Class == MOUSEBUTTONS )
{
  /* A mouse button was pressed/released. */
  if( message->Code == SELECTDOWN )
  {
    /* The left mouse button was pressed. */

    /* Save the old time: */
    sec2 = sec1;
    mic2 = mic1;

    /* Get the new time: */
    sec1 = message->Seconds;
    mic1 = message->Micros;

    /* Check if it was a double-click or not: */
    if( DoubleClick( sec2, mic2, sec1, mic1 ) )
    {
      printf("Double-Click!\n");
      /* Reset the values: */
      sec1 = 0;
      mic1 = 0;
    }
  }
}
```

9.6 TIME

You can get the current time by calling the function CurrentTime():

Synopsis: CurrentTime(seconds, micros);

seconds: (long *) Pointer to an ULONG variable which will be initialized with the current seconds stamp.

micros: (long *) Pointer to an ULONG variable which will be initialized with the current micros stamp.

Here is a short example:

```
ULONG seconds, micros;

CurrentTime( &seconds, &micros );
```

9.7 STYLE

Since Intuition is the interface between computer and the user, it is important that there are some standards concerning gadgets, requesters, menus, mouse, etc. If similar programs use the same type of style, the user will much faster learn to handle them. You, as the programmer, have therefore to make sure that your program is easily understandable and foolproof. Make your program easy to learn, and fun to handle!

9.7.1 GADGETS

Remember to make the gadgets easily recognizable, and understandable. Put short words like "CANCEL", "OK" in the gadget, or use graphics symbols. Remember to disable (ghosted) a gadget when it becomes nonfunctional.

9.7.2 REQUESTERS

Requesters have two main functions: They can inform the user about something, and/or ask the user to take some action. It is therefore important that it is easy for the user to understand

what has or will happen, and what he/she has to do. Use colourful graphics if necessary, and remember that if you want some action from the user, always give him/her a way out without affecting anything (CANCEL).

A requester which is meaningless should be taken away. Use the functions EndRequest(), ClearDMRequest().

Place the OK/YES/TRUE gadget (if there exist one) in the bottom left corner, and the CANCEL/NO/FALSE gadget in the bottom right corner. There should always be an escape (CANCEL) gadget in every requester!

9.7.3 MENUS

Menu items/subitems which are nonfunctional should be disabled (ghosted). Use the function OffMenu().

There are two menus, project and edit menu, which many program will use, and should therefore look something like this:

PROJECT

Item	Function
New	Creates a new project.
Open	Opens an old project.
Save	Saves current project.
Save as	Saves current project under a new name.
Print	Prints the whole project.
Print as	Prints parts of a project, and/or with new settings.
Quit	Quits. Remember to ask the user if he/she really wants to quit, and if he/she wants to save the project.

EDIT

Item	Function
Undo	Undoes the last operation.
Cut	Cuts a part away, and stores it in the Clipboard.
Copy	Copies a part of the project into the Clipboard.
Paste	Puts a copy of the Clipboard into the project.

Erase Deletes a part of the project without putting it into the Clipboard.

Remember to enable shortcuts for often used menu-functions. Here are some standard command key styles:

Amiga + key	Function
N	New
O	Open
S	Save
Q	Quit
U	Undo
X	Cut
C	Copy
P	Past

9.7.4 MOUSE

The left mouse button is used for "selection", while the right mouse button is used for "information transfer". The left mouse button is therefore used for selecting gadgets etc, while menu functions and double-menu requesters etc, use the right mouse button.

9.8 FUNCTIONS

AllocMem()

This function allocates memory. You specifies what type and how much you want, and it returns a pointer to the allocated memory, or NULL if there did not exist enough memory.

Synopsis: memory = AllocMem(size, type);

memory: (void *) Pointer to the new allocated memory, or NULL if no memory could be allocated. Remember! Never use memory which you have not successfully allocated.

size: (long) The size (in bytes) of the memory you want. (AllocMem() always allocates memory in multiples of eight bytes. So if you only ask for 9 bytes, Exec would actually give you 16 Bytes (2*8).)

type: (long) You need to choose one of the three following types of memory (see chapter 0

INTRODUCTION for more information about Chip and Fast memory):

MEMF_CHIP Chip memory. This memory can be accessed by both the main processor, as well as the Chips. Graphics/Sound data MUST therefore be placed in Chip memory. If it does not matter what type of memory you get (Fast or Chip), you should try to allocate Fast memory before you allocate Chip memory. (Chip memory is more valuable than Fast memory.)

MEMF_FAST Fast memory. This memory can only be accessed by the main processor. (Graphics and Sound data can NOT be stored in Fast memory, use Chip memory.) This memory is normally a little bit faster than Chip memory, since only the main processor is working with it, and it is not disturbed by the Chips.

MEMF_PUBLIC If it does not matter what type of memory you get (you do not intend to use the memory for Graphics/Sound data), you should use Fast memory. However, all Amigas do not have Fast memory, since you need to by a memory expansion in order to get it. If want to tell Exec that you would like to use Fast memory if there is any, else use Chip memory, you should ask for MEMF_PUBLIC.

If you want the allocated memory to be cleared (initialized to zeros), you should set the flag MEMF_CLEAR.

FreeMem()

This function deallocated previously allocated memory. Remember to deallocate all memory you have taken, and never deallocate memory which you have not taken.

Synopsis: FreeMem(memory, size);

memory (void *) Pointer to some memory which has previously been allocated. Remember! never use memory which has been deallocated.

size (long) The size (in bytes) of the memory you want to deallocate.

AllocRemember()

This function allocates both memory (same as AllocMem), but will also allocate space for a Remember structure which are initialized with the size of the allocated memory, and a pointer to that memory. Each time the program allocates memory with this function, the Remember structures are linked together.

Since the Remember structures contains all necessary information about the memory, and are linked together, all memory can be deallocated with one single function call (FreeRemember()).

Synopsis: memory = AllocRemember(remember, size, type);

memory: (char *) Pointer to the new allocated memory, or NULL if no memory could be allocated. Remember! Never use memory which you have not successfully allocated.

remember: (struct Remember **) Address of a pointer to a Remember structure. Before you call the AllocRemember() function for the first time you should set this pointer to NULL. (Note that it is a pointer to a pointer!)

size: (long) The size (in bytes) of the memory you want. (AllocMem() always allocates memory in multiples of eight bytes. So if you only ask for 9 bytes, Exec would actually give you 16 Bytes (2*8).)

type: (long) You need to choose one of the three following types of memory (see chapter 0 INTRODUCTION for more information about Chip and Fast memory):

MEMF_CHIP Chip memory. This memory can be accessed by both the main processor, as well as the Chips. Graphics/Sound data MUST therefore be placed in Chip memory. If it does not matter what type of memory you get (Fast or Chip), you should try to allocate Fast memory before you allocate Chip memory. (Chip memory is more valuable than Fast memory.)

MEMF_FAST Fast memory. This memory can only be accessed by the main processor. (Graphics and Sound data can NOT be stored in Fast memory, use Chip memory.) This memory is normally a little bit faster than Chip memory, since only the main processor is working with it, and it is not disturbed by the Chips.

MEMF_PUBLIC If it does not matter what type of memory you get (you do not intend to use the memory for Graphics/Sound data), you should use Fast memory. However, all Amigas do not have Fast memory, since you need to by a memory expansion in order to get it. If want to tell Exec that you would like to use Fast memory if there is any, else use Chip memory, you should ask for MEMF_PUBLIC.

If you want the allocated memory to be cleared (initialized to zeros), you should set the flag MEMF_CLEAR.

FreeRemember()

This function deallocates all memory which has been allocated by the AllocRemember() function. Note, you can deallocate all Remember structures only, and deallocate the memory yourself, if you want to.

Synopsis: FreeRemember(remember, everything);

remember: (struct Remember **) Address of a pointer to the first Remember structure (initialized by the AllocRemember() function). (Note that it is a pointer to a pointer!)

everything: (long) A boolean value. If everything is equal to TRUE, all memory (both the allocated memory and the Remember structures) are deallocated. However, if everything is equal to FALSE, only the Remember structures are deallocated, and you have to deallocate the memory yourself.

GetPrefs()

This function makes a copy of the Preferences structure.

Synopsis: pref = GetPrefs(buffer, size);

pref: (struct Preferences *) Pointer to your preferences. Same as your memory pointer (buffer), but is returned so you can check if you got a copy or not. If you could not get a copy of the preferences, the function returns NULL.

buffer: (struct Preferences *) Pointer to the memory buffer which should be used to store a copy of the preferences in.

size: (long) The number of bytes you want to copy to the

buffer. Important, the buffer must be at least as big as the number of bytes you want to copy.

GetDefPrefs()

This function makes a copy of the default Preferences structure.

Synopsis: pref = GetPrefs(buffer, size);

pref: (struct Preferences *) Pointer to the default preferences. If the function could not make a copy of the preferences, the function returns NULL.

buffer: (struct Preferences *) Pointer to the memory buffer which should be used to store a copy of the default preferences in.

size: (long) The number of bytes you want to copy to the buffer. Important, the buffer must be at least as big as the number of bytes you want to copy.

SetPrefs()

This function saves a modified preferences structure. Do NOT change the preferences unless the user really WANTS to!

Synopsis: SetPrefs(pref, size, doit);

pref: (struct Preferences *) Pointer to your modified Preferences structure.

size: (long) The number of bytes you want to change.

doit: (long) Boolean value which if FALSE, changes the preferences, but will not send a NEWPREFS message. If doit is equal to TRUE, the settings will be changed, and a NEWPREFS message will be sent. As long as the user is changing the values, doit should be FALSE, but when the user has finished, set it to TRUE, and all programs will get a NEWPREFS message.

DisplayBeep()

This function flashes the screen's colours. Can be used whenever you want to catch the user's attention.

Synopsis: DisplayBeep(screen);

screen: (struct Screen *) Pointer to the screen, which

colours you want to flash. If you have not opened a screen yourself (you are using the Workbench Screen), you can find a pointer to that screen in the Window structure: (my_window is a pointer to an opened window)

```
DisplayBeep( my_window->WScreen );
```

DoubleClick()

This function checks if the user double-clicked on one of the mouse buttons. You give the function the current as well as the previous time when the button was pressed, and it will check the preferences and return TRUE if the two button events happened within the time limit.

Synopsis: `double = DoubleClick( sec1, mic1, sec2, mic2 );`

double: (long) If the two button events happened within the current time limit, the function will return TRUE, else it will return FALSE.

sec1: (long) Time (seconds) when the button was pressed for the first time.

mic1: (long) Time (micros) when the button was pressed for the first time.

sec2: (long) Current time (seconds).

mic2: (long) Current Time (micros).

CurrentTime()

This function gives the current time.

Synopsis: `CurrentTime( seconds, micros );`

seconds: (long *) Pointer to an ULONG variable which will be initialized with the current seconds stamp.

micros: (long *) Pointer to an ULONG variable which will be initialized with the current micros stamp.

9.9 EXAMPLES

The printed version of these examples can be found in Appendix A (Part IV: Appendices).

Example1

This example shows how to allocate, and deallocate memory.

Example2

This example shows how to allocate and deallocate memory with help of the functions AllocRemember(), and FreeRemember().

Example3

This example shows how to get a copy of the preferences.

Example4

This example shows how to handle double mouse button events.

Example5

This example prints out the current time.

ACM - Amiga C Manual
Anders Bjerin

Part III: Sprites, AmigaDOS, Low Level Graphics, Virtual Sprites, Hints and Tips

http://aminet.net/package/dev/c/ACM

The complete boiled-down C manual for the Amiga which describes how to open and work with Screens, Windows, Graphics, Gadgets, Requesters, Alerts, Menus, IDCMP, Sprites, VSprites, AmigaDOS, Low Level Graphics Routines, Hints and Tips, etc. The manual also explains how to use your C Compiler and gives you important information about how the Amiga works and how your programs should be designed. The manual consists of 15 chapters together with more than 100 fully executable examples with source code.

10 SPRITES

10.1 INTRODUCTION

Sprites are small objects that can be moved around the display without changing the background. The Amiga has eight DMA channels which allows us to use 8 Hardware Sprites. The advantages of using hardware sprites is that they can be moved around, animated etc without interfering with the main processor (CPU). They are therefore extremely fast to move, and also very easy to handle.

Sprites can be used in many situations. If you make a game you can use the sprites as aliens, missiles, explosion etc. Even now, when you are reading this, a sprite is used. Intuition's Pointer is actually a sprite as described in chapter 2 WINDOWS.

10.2 LIMITATIONS

Sprites are wonderful small things but there exist some limitations:

1. Sprites may only be up to 16 pixels wide. (There is no limit on how tall they may be.) (Sprites are always using low-resolution pixels. Even if you have a high-resolution screen, the sprites will only be in low resolution. This shows how independent sprites are.)

2. Each sprite can only use three colours + one "transparent" colour.

3. There are only eight hardware sprites.

These limitations can be avoided:

1. You can place two or more sprites side by side in order to make the object wider than 16 pixels.

2. You can "connect" two sprites and boast the number of available colours from 3 to 15 plus one transparent.

3. It is possible to reuse each hardware sprite to "plop" out several sprites (VSprites) on the display.

10.3 COLOURS

Each sprite may have three different colours plus one transparent colour. Sprite zero and one will use colour register 16-19, sprite two and three will use colour register 20-23 and so on:

Sprite	Colour Register
0 and 1	16 - 19 (16 transparent)
2 and 3	20 - 23 (20 -"-)
4 and 5	24 - 27 (24 -"-)
6 and 7	28 - 31 (28 -"-)

Two important thing to remember:

1. The sprites 0 and 1, 2 and 3, 4 and 5, 6 and 7, use the same colour registers. If you change colour register 17, both sprite zero and one will be affected.

2. If you have a low-resolution screen with a depth of 5 (32 colours), the last 16 colours will be shared between the sprites and the screen. However, if you only use a 16 coloured screen (depth 4) you still use the top sixteen colour registers for the sprites. That means you can have a 16 coloured screen, and still use another 16 colours for the sprites.

Colour register 16, 20, 24 and 28 are "transparent" which means that the background colour of the screen will shine through. Those registers can have any kind of colours since they will not affect the sprites.

10.4 ACCESS HARDWARE SPRITES

If you want to use a hardware sprite you need to:

1. Declare and initialize some sprite graphics data.

2. Declare and initialize a SimpleSprite structure.

3. Call the function GetSprite() with a pointer to your SimpleSprite structure, plus a request for which sprite you want, as parameters.

4. Move the sprite around by calling the function MoveSprite() and animate it by changing the sprite graphics (ChangeSprite()).

5. Return the sprite to the hardware when you do not need it anymore, by calling the function FreeSprite().

10.4.1 SPRITE DATA

We have already described how to create your own sprite data in chapter 2 WINDOWS, but here is a short summary:

[A] The first thing you need to do is to draw on a paper how the sprite should look like. Remember that the sprite may only be 16 pixels wide, but any height. (You can of course put two sprites beside each other if you need a sprite which is wider than 16 pixels). Remember also that you may only use three colours/sprite (described above).

Imagine that you have come up with a suggestion like this:

```
0000000110000000    0: Transparent
0000001111000000    1: Red
0000011111100000    2: Yellow
0000111111110000    3: Green
0001111111111000
0011111111111100
0111111111111110
2222222222222222
2222222222222222
0333333333333330
0033333333333300
0003333333333000
0000333333330000
0000033333300000
0000003333000000
0000000330000000
```

[B] You now need to translate this into Sprite Data. Each line of the graphics will be translated into two words of data. The first word represents the first Bitplane, and the second word the second Bitplane. The idea is that if you want colour 0 both Bitplane zero and one should be 0, if you want colour 1 Bitplane zero should be 1 and Bitplane one 0 and so on:

Colour	Bitplane One	Bitplane Zero	Since
0	0	0	Binary 00 = Colour 0
1	0	1	" 01 = " 1
2	1	0	" 10 = " 2
3	1	1	" 11 = " 3

The data for the pointer would then look like this:

```
Bitplane ZERO      Bitplane ONE

0000000110000000   0000000000000000
0000001111000000   0000000000000000
0000011111100000   0000000000000000
```

```
0000111111110000   0000000000000000
0001111111111000   0000000000000000
0011111111111100   0000000000000000
0111111111111110   0000000000000000
0000000000000000   1111111111111111
0000000000000000   1111111111111111
0111111111111110   0111111111111110
0011111111111100   0011111111111100
0001111111111000   0001111111111000
0000111111110000   0000111111110000
0000011111100000   0000011111100000
0000001111000000   0000001111000000
0000000110000000   0000000110000000
```

[C] The last step is to translate the binary numbers to type UWORD. Group the binary number in four and translate it to Hexadecimal:

```
Binary  Hexadecimal
--------------------
0000 =  0
0001 =  1
0010 =  2
0011 =  3
0100 =  4
0101 =  5
0110 =  6
0111 =  7
1000 =  8
1001 =  9
1010 =  A
1011 =  B
1100 =  C
1101 =  D
1110 =  E
1111 =  F
```

The result will look like this:

```
ONE:    TWO:
------------
0180    0000      0000 0001 1000 0000  0000 0000 0000 0000
03C0    0000      0000 0011 1100 0000  0000 0000 0000 0000
07E0    0000      0000 0111 1110 0000  0000 0000 0000 0000
0FF0    0000      0000 1111 1111 0000  0000 0000 0000 0000
1FF8    0000      0001 1111 1111 1000  0000 0000 0000 0000
3FFC    0000      0011 1111 1111 1100  0000 0000 0000 0000
7FFE    0000      0111 1111 1111 1110  0000 0000 0000 0000
0000    FFFF      0000 0000 0000 0000  1111 1111 1111 1111
0000    FFFF      0000 0000 0000 0000  1111 1111 1111 1111
7FFE    7FFE      0111 1111 1111 1110  0111 1111 1111 1110
3FFC    3FFC      0011 1111 1111 1100  0011 1111 1111 1100
1FF8    1FF8      0001 1111 1111 1000  0001 1111 1111 1000
0FF0    0FF0      0000 1111 1111 0000  0000 1111 1111 0000
07E0    07E0      0000 0111 1110 0000  0000 0111 1110 0000
03C0    03C0      0000 0011 1100 0000  0000 0011 1100 0000
```

```
    0180    0180      0000 0001 1000 0000  0000 0001 1000 0000
```

[D] Since the Amiga need to store the position of the sprite, the size etc, you should also declare two empty words at the top, and to empty words at the bottom of the Sprite data. These words will be initialized and maintained by Intuition, so you do not need to bother about them.

A declaration and initialization of the sprite data would therefore be:

```
UWORD chip my_sprite_data[36]=
{
  0x0000, 0x0000,

  0x0180, 0x0000,
  0x03C0, 0x0000,
  0x07E0, 0x0000,
  0x0FF0, 0x0000,
  0x1FF8, 0x0000,
  0x3FFC, 0x0000,
  0x7FFE, 0x0000,
  0x0000, 0xFFFF,
  0x0000, 0xFFFF,
  0x7FFE, 0x7FFE,
  0x3FFC, 0x3FFC,
  0x1FF8, 0x1FF8,
  0x0FF0, 0x0FF0,
  0x07E0, 0x07E0,
  0x03C0, 0x03C0,
  0x0180, 0x0180,

  0x0000, 0x0000
};
```

IMPORTANT! Remember that all image data must as (always!) be loaded into the Chip memory (the lowest 512k of RAM). If you use Lattice C V5.00 or higher you can use the keyword chip in front of the image name. Check your compiler manual.

10.4.2 SIMPLESPRITE STRUCTURE

The SimpleSprite structure look like this:

```
struct SimpleSprite
{
  UWORD *posctldata;
  UWORD height;
  UWORD x, y;
  UWORD num;
};
```

posctldata: Pointer to an array of UWORDs which is used for the graphics of the sprite.

height: Height of the sprite (lines). (Hardware Sprites are always 16 pixels wide.)

x, y: Position on the display relative to the ViewPort/ View (Display).

num: Sprite number (0-7). When you call the GetSprite() function it will automatically initialize this field with the reserved sprite number, so set it to -1 for the moment.

10.4.3 RESERVE A SPRITE

Before you may use a hardware sprite you must have reserved it. (Since the Amiga is multitasking it can happen that another program is using the sprite.) You reserve a sprite by calling the function GetSprite():

Synopsis: sprite_got = GetSprite(my_sprite, sprite_wanted);

sprite_got: (long) GetSprite() returns the number of the sprite you got (0-7). If it could not get the desired sprite, it returns -1. Remember to check if you got the sprite you wanted! (the SimpleSprite structure's num field will also be initialized automatically.)

my_sprite: (struct SimpleSprite *) Pointer to your SimpleSprite structure.

sprite_wanted: (long) The number of the hardware sprite you want to use (0-7). If it does not matter which sprite you get you can write -1. (The System will then give you any free hardware sprite.)

10.4.4 PLAY WITH THE SPRITE

Once you have reserved a sprite you may start to play around with it. You move the sprite by calling the function MoveSprite():

Synopsis: MoveSprite(view_port, my_sprite, x, y);

view_port: (struct ViewPort *) Pointer to the ViewPort which this sprite is connected to, or 0 if the sprite

should be connected to the current View.

my_sprite: (struct SimpleSprite *) Pointer to your SimpleSprite structure.

x, y: (long) The new position on the display. (Sprites use low-resolution pixels.)

For example, MoveSprite(0, &my_sprite, 30, 50); moves the sprite (my_sprite) to position (30, 50). (Relative to the current View.)

You can also change the image (sprite data) of the sprite in order to animate it. You simply call the function ChangeSprite() with a pointer to a new sprite data as a parameter, and the rest is done for you.

Synopsis: ChangeSprite(view_port, my_sprite, new_data);

view_port: (struct ViewPort *) Pointer to the ViewPort which this sprite is connected to, or 0 if the sprite is connected to the current View.

my_sprite: (struct SimpleSprite *) Pointer to your SimpleSprite structure.

new_data: (short *) Pointer to the new sprite data.

10.4.5 FREE THE SPRITE

When you do not need the sprite any more (when your program quits for example), you need to free the sprite so another task can use it if necessary. Important! You must free all sprites you have allocated, if not no other tasks can use them, and the only way to free the hardware is then to reset the machine. You free a sprite by calling the function FreeSprite():

Synopsis: FreeSprite(sprite_got);

sprite_got: (long) The sprite you want to free (0-7).

10.4.6 PROGRAM STRUCTURE

A program using sprites would look something like this:

```
/* Since we are using sprites we need to include this file: */
#include <graphics/sprite.h>
```

```
/* 1. Declare and initialize some sprite graphics data: */
UWORD chip my_sprite_data[36]=
{
  0x0000, 0x0000,

  0x0180, 0x0000,
  0x03C0, 0x0000,
  0x07E0, 0x0000,
  0x0FF0, 0x0000,
  0x1FF8, 0x0000,
  0x3FFC, 0x0000,
  0x7FFE, 0x0000,
  0x0000, 0xFFFF,
  0x0000, 0xFFFF,
  0x7FFE, 0x7FFE,
  0x3FFC, 0x3FFC,
  0x1FF8, 0x1FF8,
  0x0FF0, 0x0FF0,
  0x07E0, 0x07E0,
  0x03C0, 0x03C0,
  0x0180, 0x0180,

  0x0000, 0x0000
};


/* 2. Declare and initialize a SimpleSprite structure: */
struct SimpleSprite my_sprite=
{
  my_sprite_data, /* posctldata, pointer to the sprite data. */
  16,             /* height, 16 lines tall. */
  40, 20,         /* x, y, position on the screen. */
  -1,             /* num, this field is automatically   */
                  /* initialized when we call the       */
                  /* GetSprite() function, so we set it */
                  /* to -1 for the moment.              */
};



UWORD chip new_sprite_data[26]=
{
  0x0000, 0x0000,
  and so on...


main()
{
  /* Open the Graphics library etc... */


  /* 3. Call the function GetSprite() with a pointer to */
  /*    your SimpleSprite structure, plus a request for */
  /*    which sprite you want, as parameters. Try to    */
  /*    reserve sprite number 2:                        */
  if( GetSprite( &my_sprite, 2 ) != 2 )
  {
```

```
    /* ERROR! */
    /* Could not reserve sprite number 2. */
  }


  /* 4. Move the sprite around by calling the function */
  /*    MoveSprite() and animate it by changing the    */
  /*    sprite graphics by calling the function        */
  /*    ChangeSprite().                                */

  /* Move the sprite to position (20, 30)
  MoveSprite( 0, &my_sprite, 20, 30 );

  /* Change the sprite data: */
  ChangeSprite( 0, &my_sprite, new_sprite_data );



  /* 5. Return the sprite to the hardware when you do not need it */
  /*    anymore, by calling the function FreeSprite():            */
  FreeSprite( my_sprite.num );

  /* Close other things etc... */
}
```

10.5 TECHNIQUES

Since both moving and changing the sprite is done by special hardware it goes with lightning speed without the main CPU even knowing about it. If you can you should use the sprites since the hardware is specialized in handling them. However, there are, as mentioned before, some limitations which can cause some problems, but by using some special techniques you can manage almost anything. Sprites can also be used to create some interesting special effects, and are perfect to animate.

10.5.1 WIDER SPRITES

If you want a space ship top be 32 pixels wide you can use two sprites side by side in order to get the desired effects. Every time you move the ship you call the MoveSprite() function twice. Since sprites are moved so fast the user will never realize that the ship is built up of two sprites.

```
(30,50)        (30+16, 50)
    *--------------*----------------
    |          ####|####           |
    |         #   #|#  ##           |
    | ##############|############## |
    |###############|###############|
```

```
    | ##############|############## |
    --------------------------------
        Sprite 2       Sprite 3
```

If you use all eight sprites you can get an image which is 128 pixels wide and any height. I do not think you need a bigger ship than that.

10.5.2 MORE COLOURS

Each sprite may only use three colours plus transparent. However, it is possible to connect two sprites and be able to use 15 colours plus transparent. (Since each sprite use two Bitplanes, it means that an attached sprite has four Bitplanes which gives 16 combinations = 15 colours + 1 transparent.)

The sprites may be attached as follows:

```
  1 to 0  (Set the SPRITE_ATTACHED bit in Sprite 1's sprite data)
  3 to 2  (                  - " -              3    - " -     )
  5 to 4  (                  - " -              5    - " -     )
  7 to 6  (                  - " -              7    - " -     )
```

The even sprites (0, 2, 4 and 6) are called Bottom Sprites, while the odd sprites (1, 3, 5 and 7) are called Top Sprites.

When you want to use an attached sprite you need to:

1. Declare and initialize two sprite data.

2. Set the Attach bit (on the odd sprite 1, 3, 5 or 7) in the second word of the sprite data.

3. Once you have reserved both of the sprites, GetSprite(), they can be moved around. It is important that both sprites are moved together so the Attach mode works, otherwise they will become three coloured sprites again.

10.5.2.1 15 COLOURED SPRITE DATA

As you have seen before, the sprite data is built up of two bitplanes. When we attach two sprites we consequently get four Bitplanes, and therefore 15 colours plus transparent.

The Bottom Sprite contributes with Bitplane zero and one, while the Top Sprite contributes with Bitplane two and three.

So if you want the sprite to look like this:

```
0000000000000000     0: Colour register 16 (Transparent)
1111111111111111     1:      - " -      17
```

```
2222222222222222     2:      - " -      18
3333333333333333     3:      - " -      19
4444444444444444     4:      - " -      20
5555555555555555     5:      - " -      21
6666666666666666     6:      - " -      22
7777777777777777     7:      - " -      23
8888888888888888     8:      - " -      24
9999999999999999     9:      - " -      25
AAAAAAAAAAAAAAAA     A:      - " -      26
BBBBBBBBBBBBBBBB     B:      - " -      27
CCCCCCCCCCCCCCCC     C:      - " -      28
DDDDDDDDDDDDDDDD     D:      - " -      29
EEEEEEEEEEEEEEEE     E:      - " -      30
FFFFFFFFFFFFFFFF     F:      - " -      31
```

The four Bitplanes should be like this:

```
Bitplane:  THREE              TWO              ONE              ZERO
-------------------------------------------------------------------------
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 1111111111111111
0000000000000000 0000000000000000 1111111111111111 0000000000000000
0000000000000000 0000000000000000 1111111111111111 1111111111111111
0000000000000000 1111111111111111 0000000000000000 0000000000000000
0000000000000000 1111111111111111 0000000000000000 1111111111111111
0000000000000000 1111111111111111 1111111111111111 0000000000000000
0000000000000000 1111111111111111 1111111111111111 1111111111111111
1111111111111111 0000000000000000 0000000000000000 0000000000000000
1111111111111111 0000000000000000 0000000000000000 1111111111111111
1111111111111111 0000000000000000 1111111111111111 0000000000000000
1111111111111111 0000000000000000 1111111111111111 1111111111111111
1111111111111111 1111111111111111 0000000000000000 0000000000000000
1111111111111111 1111111111111111 0000000000000000 1111111111111111
1111111111111111 1111111111111111 1111111111111111 0000000000000000
1111111111111111 1111111111111111 1111111111111111 1111111111111111
```

Translated into hexadecimal it would be:

```
 3    2    1    0
--------------------
0000 0000 0000 0000
0000 0000 0000 FFFF
0000 0000 FFFF 0000
0000 0000 FFFF FFFF
0000 FFFF 0000 0000
0000 FFFF 0000 FFFF
0000 FFFF FFFF 0000
0000 FFFF FFFF FFFF
FFFF 0000 0000 0000
FFFF 0000 0000 FFFF
FFFF 0000 FFFF 0000
FFFF 0000 FFFF FFFF
FFFF FFFF 0000 0000
FFFF FFFF 0000 FFFF
FFFF FFFF FFFF 0000
FFFF FFFF FFFF FFFF
```

Now we only need to put this into the two sprite data. Remember that the Bottom Sprite (0, 2, 4 or 6) contributes with Bitplanes zero and one, while the Top Sprites (1, 3, 5 or 7) contributes with Bitplanes two and three:

```
/* Sprite Data for the Bottom Sprite: */
UWORD chip bottom_sprite_data[36]=
{
  0x0000, 0x0000,

  /* Bitplane */
  /* ZERO ONE */

  0x0000, 0x0000,
  0xFFFF, 0x0000,
  0x0000, 0xFFFF,
  0xFFFF, 0xFFFF,

  0x0000, 0x0000,
  0xFFFF, 0x0000,
  0x0000, 0xFFFF,
  0xFFFF, 0xFFFF,

  0x0000, 0x0000,
  0xFFFF, 0x0000,
  0x0000, 0xFFFF,
  0xFFFF, 0xFFFF,

  0x0000, 0x0000,
  0xFFFF, 0x0000,
  0x0000, 0xFFFF,
  0xFFFF, 0xFFFF,

  0x0000, 0x0000
};

/* Sprite Data for the Top Sprite: */
UWORD chip top_sprite_data[36]=
{
  0x0000, 0x0000,

  /*  Bitplane  */
  /* TWO  THREE */
  0x0000, 0x0000,
  0x0000, 0x0000,
  0x0000, 0x0000,
  0x0000, 0x0000,

  0xFFFF, 0x0000,
  0xFFFF, 0x0000,
  0xFFFF, 0x0000,
  0xFFFF, 0x0000,

  0x0000, 0xFFFF,
  0x0000, 0xFFFF,
  0x0000, 0xFFFF,
  0x0000, 0xFFFF,
```

```
  0xFFFF, 0xFFFF,
  0xFFFF, 0xFFFF,
  0xFFFF, 0xFFFF,
  0xFFFF, 0xFFFF,

  0x0000, 0x0000
};
```

10.5.2.2 ATTACH SPRITES

To connect two sprites you simply set the SPRITE_ATTACHED flag in the second word of the sprite data of the Top Sprite. So if you want to connect sprite 3 with sprite 2, you should set the SPRITE_ATTACHED flag in sprite 3's sprite data.

```
top_sprite_data[ 1 ] = SPRITE_ATTACHED;
```

or

```
UWORD chip top_sprite_data[36]=
{
  0x0000, SPRITE_ATTACHED,
  and so on...
```

10.5.2.3 MOVE ATTACHED SPRITES

Once two sprites are attached you simply reserve them, and you have got one 15 coloured sprite:

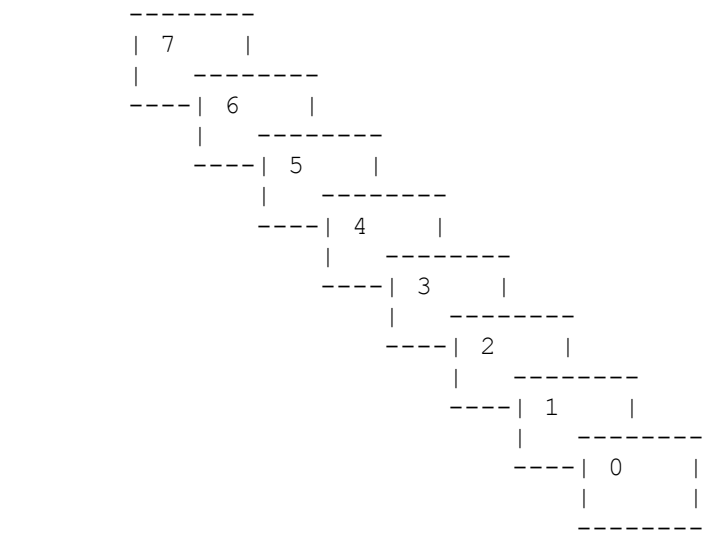
```
  /* Reserve sprite 2 as Bottom Sprite: */
  bottom_sprite_got = GetSprite( &my_bottom_sprite, 2 );

  if( bottom_sprite_got != 2 )
    /* ERROR! */


  /* Reserve sprite 3 as Top Sprite: */
  top_sprite_got = GetSprite( &my_top_sprite, 3 );

  if( top_sprite_got != 3 )
    /* ERROR! */
```

It is important that both of the sprites are moved simultaneously so the Attach function will work. The nice thing is that if you move the Bottom sprite, the Top sprite will also move automatically. If you on the other hand move the Top sprite only, the Bottom sprite will remain unchanged, and the special Attach function will no work. (Back to 3 colours.)

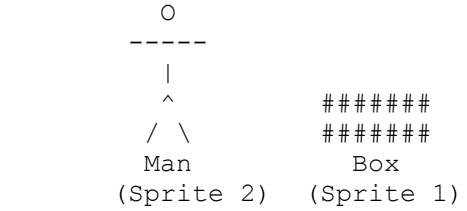
If you want to have some fun you can reserve sprite 1 and set the SPRITE_ATTACHED flag. Since sprite 0 is already used by Intuition as the pointer, sprite 1 would be attached to the pointer, and you could have a 15 coloured pointer! (Whenever Intuition moves the pointer (sprite 0, which is the Bottom sprite), your sprite (sprite 1, Top Sprite) will move automatically.) (You can change Intuition's pointer by calling the function SetPointer. See chapter 2 WINDOWS for more information.)

10.5.3 LEVELS

Each sprite has its own "level" (priority) which means that some sprites will move over, and some under other sprites when they touch each other. The higher sprite number the lower level (priority). That means that sprite 3 will move over sprite 4, while sprite 2 will move over sprite 3:



This can be used in many ways. For example, if you are writing a game with a small man jumping into a box, you can use two sprites. One for the man, and one for the box. The important thing to remember is that the box must have a lower sprite number than the man. If you then would place the man at the same position as the box, the man would appear to go inside the box:



```
    O
  -----
    |
 #######
 #######
 Man (Sprite 2)
 Box (Sprite 1)
```

10.6 FUNCTIONS

GetSprite()

This function reserves a sprite. You must always reserve a sprite before you may use it.

Synopsis: spr_got = GetSprite(my_sprite, spr_wanted);

spr_got: (long) GetSprite() returns the number of the sprite you got (0-7). If it could not get the desired sprite, it returns -1. Remember to check if you got the sprite you wanted! (the SimpleSprite structure's num field will also be initialized automatically.)

my_sprite: (struct SimpleSprite *) Pointer to your SimpleSprite structure.

spr_wanted: (long) The number of the hardware sprite you want to use (0-7). If it does not matter which sprite you get you can write -1. (The System will then give you any free hardware sprite)

MoveSprite()

Use this function to move a sprite.

Synopsis: MoveSprite(view_port, my_sprite, x, y);

view_port: (struct ViewPort *) Pointer to the ViewPort which this sprite is connected to, or 0 if the sprite is connected to the current View.

my_sprite: (struct SimpleSprite *) Pointer to your SimpleSprite structure.

x, y: (long) The new position on the display. (Sprites use low-resolution pixels.)

ChangeSprite()

This function changes the sprite data (image) of a sprite.

Synopsis: ChangeSprite(view_port, my_sprite, new_data);

view_port: (struct ViewPort *) Pointer to the ViewPort which this sprite is connected to, or 0 if the sprite is connected to the current View.

my_sprite: (struct SimpleSprite *) Pointer to your SimpleSprite structure.

new_data: (short *) Pointer to the new sprite data.

FreeSprite()

This function returns an already reserved sprite.

Synopsis: FreeSprite(sprite_got);

sprite_got: (long) The sprite you want to free (0-7).

WaitTOF()

This function waits for the video beam to reach the top of the display. Can be used if you want to slow down the speed a bit, and make the animation smoother.

Synopsis: WaitTOF();

10.7 EXAMPLES

The printed version of these examples can be found in Appendix A (Part IV: Appendices).

Example1

This program shows how to declare and initialize some sprite data and a SimpleSprite structure. It also shows how to reserve a sprite (sprite 2), and how to move it around. The user moves the sprite by pressing the arrow keys.

Example2

This program shows how to declare and initialize some sprite data and a SimpleSprite structure. It also shows how to reserve a sprite (sprite 2), and how to move it around. The user moves the sprite by pressing the arrow keys. In this example we animate the sprite (6 frames, taken from the arcade game Miniblast).

Example3

This program shows how to set up a 15 coloured sprite, and how to move it around.

11 AMIGADOS

11.1 INTRODUCTION

In this chapter are we going to discuss how to read and write files from/to different devices (disk drivers etc). How to protect your data from being corrupted by other tasks working with the same file, and how to attach comments to files, how to rename them etc...

However, before we start it is best to look at how AmigaDOS works, and explain some commonly used words. If you are familiar with AmigaDOS and CLI you can skip the following chapters and immediately start on chapter 11.2 (OPEN AND CLOSE FILES).

11.1.1 PHYSICAL DEVICES

Physical devices are parts of the Amiga to which files can be read from and sent to. The most commonly used physical device is undoubtedly the internal disk drive (DF0:).

Here is the list of the standard AmigaDOS devices:

```
-------------------------------------------------------------
DF0: Diskdrive 0.      File/Directory access
DF1: Diskdrive 1.      File/Directory access
DF2: Diskdrive 2.      File/Directory access
DF3: Diskdrive 3.      File/Directory access
RAM: RAMdisk,          File/Directory access

SER: Serial port,      Input/Output
PAR: Parallel port,    Input/Output
PRT: Printerdevice,    Output to printer (trough Preferences)

CON: Console, (window) Input/Output
RAW: RAW, (window)     Input/Output
NIL: Nothing,          Output to nothing
-------------------------------------------------------------
```

(Note that all device-names end with a colon.)

If you want to copy a file called "program.c" from the internal disk drive (DF0:) to the second disk drive (DF1:) you write:

```
  copy from DF0:program.c to DF1:
```

If you want to copy a file called "program.c" from the internal disk drive (DF0:) to the parallel port (PAR:) you write:

```
  copy from DF0:program.c to PAR:
```

If a printer was connected to the parallel port the file "program.c" would be printed out with the printers default mode. If the printer was connected to the serial port (SER:) you write:

```
copy from DF0:program.c to SER:
```

However, if you want to copy a file to the printer, and you want to use the printer mode you have chosen with Preferences, you use the printer device (PRT:). (With Preferences you can decide which device you want to use (PAR: or SER:), and if the text should be printed with letter quality or in draft mode etc.)

```
copy from DF0:program.c to PRT:
```

AmigaDOS allows you even to use special windows (CON:) for input/output. The only thing you need to remember is to tell AmigaDOS what size and position of the window you want. The syntax is:

```
CON:x/y/width/height/(title)
```

Remember to include all four slashes! (The window title is optional.) To copy the file "program.c" to a special window you write:

```
copy from DF0:program.c to CON:10/20/320/100/Text
```

This will print the file program.c to a window positioned at x position 10, y position 20, width 320, height 100, with the title "Text". If you want to use spaces in the title you need to put quotes around the whole expression:

```
copy from DF0:program.c to "CON:10/20/320/100/Spacy Window"
```

11.1.2 VOLUMES

To access a disk you can either use the physical device name of the drive in which the disk is, or use the volume name of the disk. If you want to copy a file from the RAMdisk called "program.c", to a disk called "DOCUMENTS", and the disk is in the second disk drive, you can either write:

```
copy from RAM:program.c to DF1:
```

or

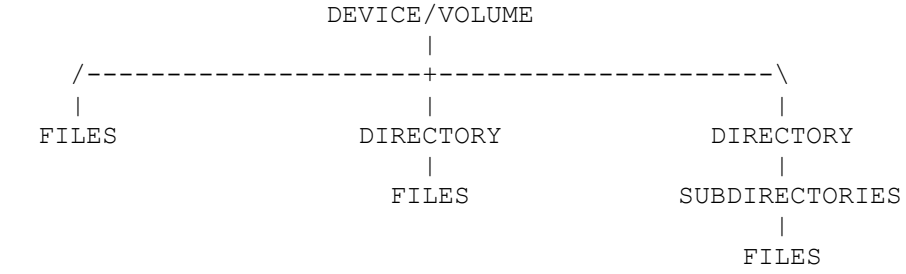
```
copy from RAM:program.c to DOCUMENTS:
```

(Note the colon after the volume name!) The advantage with the

last example is that you do not need to bother in which drive the DOCUMENTS disk is in. Furthermore, if the right disk was not in any drive, AmigaDOS will ask you to insert it, and you do not need to worry about writing to the wrong disk.

11.1.3 DIRECTORIES/SUBDIRECTORIES/FILES

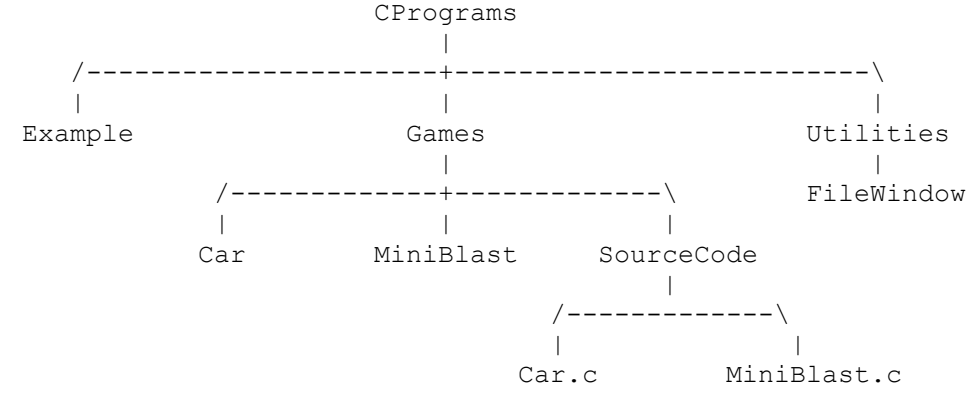
On a disk there exist directories/subdirectories and files. A subdirectory is a directory inside another directory. (There may be subdirectories inside other subdirectories and so on.) See picture:



How to gain access to files:

Left of the colon	Right of the colon	Right of a slash
Device name or Volume name	Directory name or Filename	Subdirectory name or Filename

For example. Here is a disk named "CPrograms" on which there is a program called "Example" and two directories named "Games" and "Utilities". In the Utilities directory there is a program called FileWindow, and in the Games directory there are two games called Car and MiniBlast. There exist also a subdirectory called "SourceCode" which contains two files "Car.c" and "MiniBlast.c". See drawing:



Here is how to access all the files:

```
CPrograms:Example
CPrograms:Utilities/FileWindow
CPrograms:Games/Car
CPrograms:Games/MiniBlast
CPrograms:Games/SourceCode/Car.c
CPrograms:Games/SourceCode/MiniBlast.c
```

11.1.4 LOGICAL DEVICES

Logical devices is a simple way to find files, regardless of where the file actually is. For example: If you have all your C programs on a disk called "Programs", placed i directory named "Examples", and you want to run the program "test1" you need to write:

```
Programs:Examples/test1
```

If you often want to access files in that directory you can assign a logical device to it. You then only need to write the logical device name and the filename, and AmigaDOS will automatically look on the right disk and directory.

You assign logical devices by using the CLI command "Assign" which is called like this:

```
Assign [logical device name]: [device/(directory/subdirectory)]
```

For example:

```
Assign EX: Programs:Examples
```

To gain access to the file "test1" you then only need to write:

```
EX:test1
```

When you boot up the computer it will automatically create some commonly used logical devices. A good example is the logical device "FONTS:". It is automatically assigned to the system disk's "fonts" directory. Here is the list of some of the most commonly used logical devices:

Logical device	Description
SYS:	system disk
C:	CLI commands
FONTS:	fonts
L:	handlers
LIBS:	libraries

```
  S:              sequence library
  DEVS:           devices
```

The logical device "C:" is assigned to the system disk's "c" directory, where all CLI commands are. If you have copied some CLI commands to the RAMdisk, and you want AmigaDOS to look there instead of looking on the system disk's c directory you simply reassign the C: device. For example:

```
  Assign C: RAM:
```

If you call Assign without any parameters, it will list all logical devices, and their assignments. If you have booted up the system with the Lattice C Compiler disk, the following logical devices are assigned:

```
  Automatically assigned:
  ---------------------------------------
  S               Lattice_C_5.0.1:s
  L               Lattice_C_5.0.1:l
  C               Lattice_C_5.0.1:c
  FONTS           Lattice_C_5.0.1:fonts
  DEVS            Lattice_C_5.0.1:devs
  LIBS            Lattice_C_5.0.1:libs
  SYS             Lattice_C_5.0.1:

  Specially assigned: (Lattice C)
  ---------------------------------------
  LIB             Volume: Lattice_C_5.0.2
  INCLUDE         Volume: Lattice_C_5.0.2
  LC              Lattice_C_5.0.1:c
  QUAD            RAM DISK:
  ENV             RAM DISK:env
  CLIPS           RAM DISK:clipboards
```

11.2 OPEN AND CLOSE FILES

Before you can do anything with a file you have to ask the operating system to "open it". This is because AmigaDOS needs to know what you are going to do with the file. Are you going to create a new file or are you going to work with an already opened file. You open a file by calling the function Open():

```
Synopsis:    file_handle = Open( file_name, mode );

file_handle: (BPTR) Actually a pointer to a FileHandle
             structure. If the system could not open the file
             with our requirements Open() returns NULL.

file_name:   (char *) Pointer to a text string which contains
             the file name including any necessary device/
             directory names.
```

mode: (long) When you open a file you need to tell the system what you are going to do with it. This field should therefore contain one of the following flags:

MODE_OLDFILE: Opens an existing file for reading and writing.

MODE_NEWFILE: Opens a new file for writing.

MODE_READWRITE: Opens an old file with an exclusive lock. (Explained later in this chapter.)

MODE_READONLY: Same as MODE_OLDFILE.

For example, if you want to create a new file called "Highscore.dat" you write:

```
/* Try to open the file: */
file_handle = Open( "Highscore.dat", MODE_NEWFILE );

/* Check if we have opened the file: */
if( file_handle == NULL )
  /* Could NOT open file! */
```

Once you have finished reading/writing the file you need to close it. You do it by calling the function Close():

Synopsis: Close(file_handle);

file_handle: (BPTR) Actually a pointer to a FileHandle structure which has been initialized by a previous Open() call.

Remember to close ALL files you have opened!

11.3 READ AND WRITE FILES

Once you have successfully opened a file you can start to read/write. AmigaDOS consider files to be a stream of bytes, and when you read/write something you need to specify how many bytes you want to read/write.

11.3.1 READ()

You read data by calling the function Read():

```
Synopsis:    bytes_read = Read( file_handle, buffer, size );
```

bytes_read: (long) Number of bytes actually read. Even if you tell AmigaDOS that you want to read x number of bytes, it is not certain that you actually can do it. The file is maybe corrupted, not as big as you thought etc.

file_handle: (BPTR) Actually a pointer to a FileHandle structure which has been initialized by a previous Open() call.

buffer: (char *) Pointer to the data buffer you want to read the data into.

size: (long) Number of bytes you want to read.

For example, if you want to read a file which contains an array (highscore) of ten integers, you write: (the file has already been opened.)

```
bytes_read = Read( file_handle, highscore, sizeof( highscore ) );

if( bytes_read != sizeof( highscore ) )
  /* ERROR while reading! */
```

11.3.2 WRITE()

When you what to write to a file you use the function Write():

```
Synopsis:       bytes_written = Write( file_handle, buffer, size );
```

bytes_writtten: (long) Number of bytes actually written. Even if you tell AmigaDOS that you want to write x number of bytes, it is not certain that you actually can do it. Maybe the disk was full, writeprotected etc.

file_handle: (BPTR) Actually a pointer to a FileHandle structure which has been initialized by a previous Open() call.

buffer: (char *) Pointer to the data buffer which you want to write.

size: (long) Number of bytes you want to write.

For example, if you want to save an array (highscore) of ten integers to a file, you simply write: (the file has

already been opened.)

```
bytes_written = Write( file_handle, highscore, sizeof( highscore ) );

if( bytes_written != sizeof( highscore ) )
  /* ERROR while writing! */
```

11.4 MOVE INSIDE FILES

If you write something to a disk the "file cursor" will be positioned where you stopped writing. So if you start writing again, the new data will be placed after the first data. If you on the other hand wanted to write over the old data you need to move the file cursor to beginning of the data before you start writing. You move the file cursor by calling the function Seek():

```
Synopsis:    old_pos = Seek( file_handle, new_pos, mode );

old_pos:     (long) Previous position in the file, or -1 if
             an error occurred.

file_handle: (BPTR) Actually a pointer to a FileHandle
             structure which has been initialized by a
             previous Open() call.

new_pos:     (long) New position relative to the "mode".

mode:        (long) The new_pos can be relative to:
               OFFSET_BEGINNING: Beginning of the file.
               OFFSET_CURRENT:   Current position.
               OFFSET_END:       The end of the file.
```

So if you want to move the cursor to the beginning of the file you set mode to OFFSET_BEGINNING, and new_pos to 0. (When you open a file by calling the function Open(), the file cursor is automatically placed at the beginning of the file.)

If you on the other hand want to move the cursor to the end of the file you set mode to OFFSET_END, and new_pos still to 0.

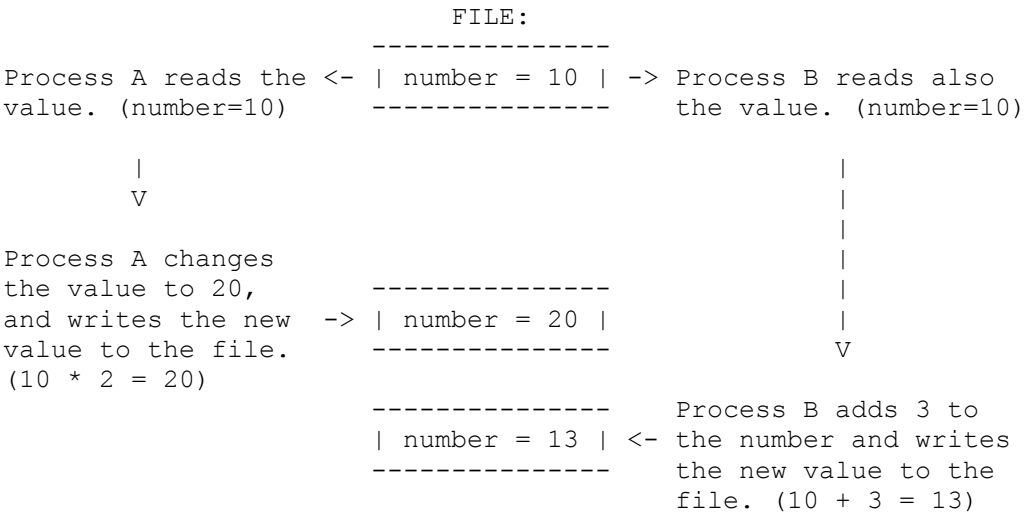
To move 10 bytes forward from the current position you set mode to OFFSET_CURRENT, and new_pos to 10. To move 10 bytes backwards you set new_pos to -10.

11.5 FILES AND MULTITASKING

Since the Amiga can have several tasks running at the same time it can happen that more than one task work with the same file. This can be very dangerous since one process may destroy some

data another process has created.

Imagine two tasks, one will read a file and multiply a value with two, while the other tasks should add 3. Since the two tasks can run at the same time it could happen that they would update the file at the same time:



As you can see, the number that was updated by process A, has been lost. What should have happen was that process A should have "locked" the file so no other tasks could work with it. Process B would then have been forced to wait for Process A to "unlock" the file before it could read the value.

On large mainframe computers there exist different "locks" with different priorities, but on the Amiga there exist only two different types of lock. You can lock a file so other processes may read it but not change it (SHARED_LOCK), or if you do not want any other tasks to even read the file you set an EXCLUSIVE_LOCK. Call the function Lock() to set a lock:

```
Synopsis:  lock = Lock( file_name, mode );

lock:      (BPTR) Actually a pointer to a FileLock structure.

file_name: (char *) Pointer to a text string which contains
           the file name including any necessary devices/
           directories.

mode:      (long) Accessmode:
             SHARED_LOCK:     Other processes may read the file.
             ACCESS_READ:                  - " -
             EXCLUSIVE_LOCK:  No other processes may use this f.
             ACCESS_WRITE:                 - " -
```

Here is an example how to lock a file so no other processes may use the file:

```
struct FileLock *lock;
```

```
lock = Lock( "HighScore.dat", EXCLUSIVE_LOCK );

if( lock == NULL )
  /* ERROR Could NOT create lock! */
```

When you do not need the file any more you must "unlock" it. You do it by calling the function UnLock():

Synopsis: UnLock(lock);

lock: (BPTR) Actually a pointer to FileLock structure which has been initialized by a previous Lock() call.

Remember to unlock ALL files you have locked!

11.6 OTHER USEFUL FUNCTIONS

Here is a list of some other useful functions that you probably want to use. AmigaDOS allows you to rename and delete files, attach comments to them, set protection bits etc. All these things can be done with help of the functions listed below:

11.6.1 CREATE DIRECTORIES

If you want to create a directory you use the function CreateDir(). You only need to give the function a pointer to the name of the directory you want to create, and it will either return a pointer to a Lock (the new directory is automatically given an EXCLUSIVE Lock), or NULL if something went wrong.

Remember to unlock the new directory before your program determinates.

If there already exist a directory with the same name, it is simply locked and your program will not notice any difference.

Synopsis: lock = CreateDir(name);

lock: (BPTR) Actually a pointer to a FileLock structure. If lock is equal to NULL, AmigaDOS have not been able to create the new directory.

name: (char *) Pointer to a string containing the name of the new directory.

Here is a short example:

```
struct FileLock *lock;

lock = CreateDir( "RAM:MyDirectory" );

if( lock == NULL )
  exit( ERROR );

...

UnLock( lock );
```

11.6.2 DELETE FILES AND DIRECTORIES

If you want to delete a file you call the function DeleteFile(). You only need to give it a file/directory name and it will either return TRUE (file was deleted) or FALSE (file could not be deleted). Remember that you can only delete a directory if it is empty.

Synopsis: ok = DeleteFile(name);

ok: (long) Actually a Boolean. It is TRUE if AmigaDOS could delete the file/directory, else FALSE which means something went wrong. (Eg. disk write-protected, directory not empty etc.)

name: (char *) Pointer to a string containing the name of the file/directory you want to delete.

11.6.3 RENAME FILES AND DIRECTORIES

When you want to rename a file/directory you use the function Rename(). You can even move a file between directories by renaming it. For example:

```
  Rename( "df0:Documents/Sale.doc", "df0:Letters/Sale.doc" );
```

will move the file Sale.doc from the directory "Documents" to directory "Letters". Note! You can not rename a file from one volume to another.

Synopsis: ok = Rename(old_name, new_name);

ok: (long) Actually a Boolean. It is TRUE if AmigaDOS could rename the file/directory, else FALSE which means something went wrong. (Eg. disk write-protected.)

old_name: (char *) Pointer to a string containing the old file/directory name.

new_name: (char *) Pointer to a string containing the new file/directory name.

11.6.4 ATTACH COMMENTS TO FILES AND DIRECTORIES

AmigaDOS allows you to set a comment on a file or directory. The comment can give a brief description of what the file is, and is very handy to use when you want to give some extra information about the file etc. The comment can be up to 80 characters long.

A program can set a comment to a file/directory by using the function SetComment().

Synopsis: ok = SetComment(name, comment);

ok: (long) Actually a Boolean. It is TRUE if AmigaDOS could attach the new comment, else FALSE which means something went wrong. (Eg. disk write-protected.)

name: (char *) Pointer to a string containing the name of the file/directory you want to attach the comment to.

comment: (char *) Pointer to a string containing the comment. (A comment may be up to 80 characters long.)

Here is a short example:

```
if( SetComment( "Letter.doc", "Letter to Mr Smith" ) == FALSE )
  printf("ERROR! Could not attach the comment to the file!\n");
```

11.6.5 PROTECT FILES AND DIRECTORIES

You can protect files and directory from being accidentally deleted or changed. You do it by calling the function SetProtection() which will alter the protection bits as desired. Here are a list of the protection bits (flags) you can change:

```
  FIBF_DELETE  : the file/directory can not be deleted.
  FIBF_EXECUTE : the file can not be executed.
  FIBF_WRITE   : you can not write to the file.
  FIBF_READ    : you can not read the file.
```

```
  FIBF_ARCHIVE : Archive bit.
  FIBF_PURE    : Pure bit.
  FIBF_SCRIPT  : Script bit.
```

Note! All of the flags are for the moment not working!

Synopsis: ok = SetProtection(name, mask);

ok: (long) Actually a Boolean. It is TRUE if AmigaDOS could alter the protection bits, else FALSE which means something went wrong. (Eg. disk write-protected.)

name: (char *) Pointer to a string containing the name of the file/directory you want to change the protection bits.

mask: (long) The protection bits. (For example, if you want to make the file/directory not deletable, and that it can not be executed you should set the protection bits: FIBF_DELETE | FIBF_EXECUTE.)

11.7 EXAMINE FILES AND DIRECTORIES

You can examine a file or directory with the function Examine(). You give the function a pointer to a "lock" on the file/directory you want to examine, and it will initializes a FileInfoBlock structure for you. The FileInfoBlock structure contains several interesting fields which you then can look at.

11.7.1 FILEINFOBLOCK AND DATESTAMP STRUCTURE

The FileInfoBlock look like this:

```
struct FileInfoBlock
{
  LONG fib_DiskKey;
  LONG fib_DirEntryType;
  char fib_FileName[108];
  LONG fib_Protection;
  LONG fib_EntryType;
  LONG fib_Size;
  LONG fib_NumBlocks;
  struct DateStamp fib_Date;
  char fib_Comment[80];
  char fib_Reserved[36];
};
```

fib_DiskKey: Key number for the disk. Usually of no interest for us.

fib_DirEntryType: If the number is smaller than zero it is a file. On the other hand, if the number is larger than zero it is a directory.

fib_FileName: Null terminated string containing the file-name. (Do not use longer filenames than 30 characters.)

fib_Protection: Field containing the protection flags:

```
FIBF_DELETE  : the file/directory can not be
               deleted.
FIBF_EXECUTE : the file can not be executed.
FIBF_WRITE   : you can not write to the file.
FIBF_READ    : you can not read the file.
FIBF_ARCHIVE : Archive bit.
FIBF_PURE    : Pure bit.
FIBF_SCRIPT  : Script bit.
```

(Note! All of the flags are for the moment not working!)

fib_EntryType: File/Directory entry type number. Usually of no interest for us.

fib_Size: Size of the file (in bytes).

fib_NumBlocks: Number of blocks in the file.

fib_Date: Structure containing the date when the file was latest updated/created.

fib_Comment: Null terminated string containing the file comment. (Max 80 characters including the NULL sign.)

fib_Reserved: This field is for the moment reserved, and may therefore not be used.

The DateStamp structure look like this:

```
struct DateStamp
{
  LONG ds_Days;
  LONG ds_Minute;
  LONG ds_Tick;
};
```

ds_Days: Number of days since 01-Jan-1978.

ds_Minute: Number of minutes past midnight.

ds_Tick: Number of ticks past the last minute. There are 50 ticks / second. (50 * 60 = 3000 ticks / minute.)

11.7.2 EXAMINE()

The Examine() function is called like this:

Synopsis: ok = Examine(lock, fib_ptr);

ok: (long) Actually a Boolean. It is TRUE if AmigaDOS could get information about the file/directory, else FALSE which means something went wrong.

lock: (BPTR) Actually a pointer to a FileLock structure.

fib_ptr: (struct FileInfoBlock *) Pointer to a FileInfoBlock structure which will be initialized with some information about the file/directory. IMPORTANT! the structure must be on a 4 byte boundary. (See below for more information.)

11.7.3 4 BYTE BOUNDARY

There is one problem left. (Who said it should be easy?) AmigaDOS was not, as everything else on the Amiga, written in C. They used the BCPL programming language which is the predecessor of C. The problem is that BCPL only uses one data type - longword (LONG). ALL data which is handled by AmigaDOS must therefore be on a "4 byte boundary". (Are you with me?) To make sure that a structure starts on a 4 byte boundary you need to allocate the memory by using the function AllocMem(). (Remember to deallocate the memory once you do not need it any more!)

So instead of writing:

```
-----------------------------------------------------------------
struct FileInfoBlock fib; /* Declare a FileInfoBlock called */
                          /* fib. This structure may NOT    */
                          /* necessary start on a 4 byte    */
                          /* boundary!                      */
-----------------------------------------------------------------
```

You write:

```
-----------------------------------------------------------------
struct FileInfoBlock *fib_ptr; /* Declare a FileInfoBlock */
                               /* pointer called fib_ptr. */

/* Allocate enough memory for a FileInfoBlock structure: */
/* This memory WILL be on a 4 byte boundary!             */
fib_ptr = AllocMem( sizeof( struct FileInfoBlock ),
                    MEMF_PUBLIC | MEMF_CLEAR )
```

```
/* Check if we have allocated the memory successfully: */
if( fib_ptr == NULL )
  exit(); /* NOT ENOUGH MEMORY! */


... your program ...


/* Deallocate the memory we have allocated for the fib. struct: */
FreeMem( fib_ptr, sizeof( struct FileInfoBlock ) );
```

11.7.4 EXAMPLE

Here is an example which shows how to use the Examine() function: (Remember to deallocate all allocated memory and to unlock all locked files when your program terminates!)

```
#include <libraries/dos.h>
#include <exec/memory.h>

main()
{
  struct FileLock *lock;
  struct FileInfoBlock *fib_ptr; /* Declare a FileInfoBlock */
                                 /* pointer called fib_ptr. */


  /* 1. Allocate enough memory for a FileInfoBlock structure: */
  fib_ptr = AllocMem( sizeof( struct FileInfoBlock ),
                      MEMF_PUBLIC | MEMF_CLEAR )

  /* Check if we have allocated the memory successfully: */
  if( fib_ptr == NULL )
    exit(); /* NOT ENOUGH MEMORY! */


  /* 2. Try to lock the file: */
  lock = Lock( "highscore.dat", SHARED_LOCK )

  /* Could we lock the file? */
  if( lock == NULL )
  {
    /* Deallocate the memory we have allocated: */
    FreeMem( fib_ptr, sizeof( struct FileInfoBlock ) );

    exit();
  }


  /* 3. Try to get some information about the file: */
```

```
  if( Examine( lock, fib_ptr ) == NULL )
  {
    /* Deallocate the memory we have allocated: */
    FreeMem( fib_ptr, sizeof( struct FileInfoBlock ) );

    /* Unlock the file: */
    UnLock( lock );

    exit();
  }


  /* 4. You may now examine the FileInfoBlock structure! */


  /* 5. Unlock the file: */
  UnLock( lock );


  /* 6. Deallocate the memory we have allocated: */
  FreeMem( fib_ptr, sizeof( struct FileInfoBlock ) );
}
```

11.7.5 EXAMINE FILES/SUBDIRECTORIES IN A DIRECTORY/DEVICE

A directory/device can contain several files plus several (sub)directories. If you want to examine not only a file/directory, but all files/directories in a directory/device the Examine() function is not enough. You need to use the function ExNext(), which works together with the Examine function.

To examine a directory/device you first need to call the function Examine() as normal. If it was a directory/device you start calling the function ExNext(), and continue calling it until you receive an error message. The error message is normally an "end of directory" message, which means that you have examined everything in the directory/device. It can also be some other error message which indicates that something went wrong while reading.

11.7.5.1 EXNEXT()

The ExNext() function is very similar to Examine(). It is called like this:

Synopsis: ok = ExNext(lock, fib_ptr);

ok: (long) Actually a Boolean. It is TRUE if AmigaDOS could get information about a file/directory in the directory/device, else FALSE which means something

went wrong.

lock: (BPTR) Actually a pointer to a FileLock structure.

fib_ptr: (struct FileInfoBlock *) Pointer to a FileInfoBlock structure which will be initialized with some information about the file/directory. IMPORTANT! the structure must be on a 4 byte boundary. (See above for more information.)

Remember that you first need to call the function Examine(), and if it was a directory/device (fib_DirEntryType > 0), you may call the function ExNext() until you receive an error message.

11.7.5.2 ERROR MESSAGES

When you have used an AmigaDOS function, and you have received an error message, you can call the function IoErr() to find out what went wrong. You do not send IoErr() any parameters, it only returns a flag (value) which tells your program what went wrong.

Here is the complete (V1.3) error list: (The header file "libraries/dos.h" contains the definitions.) I do not think I need to explain what they mean.

```
ERROR_NO_FREE_STORE
ERROR_TASK_TABLE_FULL
ERROR_LINE_TOO_LONG
ERROR_FILE_NOT_OBJECT
ERROR_INVALID_RESIDENT_LIBRARY
ERROR_NO_DEFAULT_DIR
ERROR_OBJECT_IN_USE
ERROR_OBJECT_EXISTS
ERROR_DIR_NOT_FOUND
ERROR_OBJECT_NOT_FOUND
ERROR_BAD_STREAM_NAME
ERROR_OBJECT_TOO_LARGE
ERROR_ACTION_NOT_KNOWN
ERROR_INVALID_COMPONENT_NAME
ERROR_INVALID_LOCK
ERROR_OBJECT_WRONG_TYPE
ERROR_DISK_NOT_VALIDATED
ERROR_DISK_WRITE_PROTECTED
ERROR_RENAME_ACROSS_DEVICES
ERROR_DIRECTORY_NOT_EMPTY
ERROR_TOO_MANY_LEVELS
ERROR_DEVICE_NOT_MOUNTED
ERROR_SEEK_ERROR
ERROR_COMMENT_TOO_BIG
ERROR_DISK_FULL
ERROR_DELETE_PROTECTED
```

```
ERROR_WRITE_PROTECTED
ERROR_READ_PROTECTED
ERROR_NOT_A_DOS_DISK
ERROR_NO_DISK
ERROR_NO_MORE_ENTRIES
```

The IoErr() function is called like this:

Synopsis: error = IoErr();

error: (long) This field contains one of the above mentioned flags.

So when you use the function ExNext(), and you receive and error message, you should call the function IoErr to check what went wrong. If it was not ERROR_NO_MORE_ENTRIES (you have checked everything in the directory), something terrible went wrong (error while reading).

11.7.5.3 EXAMPLE

Here is parts of a program that examines everything inside a directory/device, and prints out the filenames:

```
if( Examine( lock, fib_ptr ) )
{
  if( fib_ptr->fib_DirEntryType > 0 )
  {
    printf("Directory name: %s\n", fib_ptr->fib_FileName );

    while( ExNext( lock, fib_ptr ) )
    {
      printf("name: %s\n", fib_ptr->fib_FileName );
    }

    if( IoErr() == ERROR_NO_MORE_ENTRIES )
      printf("End of directory!\n");
    else
      printf("ERROR WHILE READING!!!\n");
  }
  else
    printf("This is a file!\n");
}
else
  printf("Could not examine the directory/device!\n");
```

11.8 FUNCTIONS

Here is a list of commonly used functions:

Open()

This function opens a file. Remember, before you can read/ write files you have to open them.

Synopsis: file_handle = Open(file_name, mode);

file_handle: (BPTR) Actually a pointer to a FileHandle structure. If the system could not open the file with our requirements Open() returns NULL.

file_name: (char *) Pointer to a text string which contains the file name including any necessary devices/ directories.

mode: (long) When you open a file you need to tell the system what you are going to do with it. This field should therefore contain one of the following flags:

MODE_OLDFILE: Opens an existing file for reading and writing.

MODE_NEWFILE: Opens a new file for writing. (If the file already exist it is deleted.)

MODE_READWRITE: Opens an old file with an exclusive lock. (The file is automatically locked with an EXCLUSIVE_LOCK.)

MODE_READONLY: Same as MODE_OLDFILE.

Close()

This function closes an already opened file. Remember to close ALL files you have opened!

Synopsis: Close(file_handle);

file_handle: (BPTR) Actually a pointer to a FileHandle structure which has been initialized by a previous Open() call.

Read()

This function reads a specified number of bytes from a file.

Synopsis: bytes_read = Read(file_handle, buffer, size);

```
  bytes_read:  (long) Number of bytes actually read. Even if
               you tell AmigaDOS that you want to read x
               number of bytes, it is not certain that you
               actually can do it. The file is maybe corrupted,
               not as big as you thought etc.

  file_handle: (BPTR) Actually a pointer to a FileHandle
               structure which has been initialized by a
               previous Open() call.

  buffer:      (char *) Pointer to the data buffer you want to
               read the data into.

  size:        (long) Number of bytes you want to read.
```

Write()

```
  This function writes a specified number of bytes to a file.

  Synopsis:  bytes_wr = Write( file_handle, buffer, size );

  bytes_wr:    (long) Number of bytes actually written. Even if
               you tell AmigaDOS that you want to write x number
               of bytes, it is not certain that you actually
               can do it. Maybe the disk was full, write-
               protected etc.

  file_handle: (BPTR) Actually a pointer to a FileHandle
               structure which has been initialized by a
               previous Open() call.

  buffer:      (char *) Pointer to the data buffer which you
               want to write.

  size:        (long) Number of bytes you want to write.
```

Seek()

```
  This function moves the "file cursor" inside a file:

  Synopsis:    old_pos = Seek( file_handle, new_pos, mode );

  old_pos:     (long) Previous position in the file, or -1 if
               an error occurred.

  file_handle: (BPTR) Actually a pointer to a FileHandle
               structure which has been initialized by a
               previous Open() call.

  new_pos:     (long) New position relative to the "mode".

  mode:        (long) The new_pos can be relative to:
                 OFFSET_BEGINNING: Beginning of the file.
                 OFFSET_CURRENT:   Current position.
```

```
                OFFSET_END:       The end of the file.
```

Lock()

This function "locks" a file so no other processes may alter the contents (SHARED_LOCK). You can even prevent other processes to read the file (EXCLUSIVE_LOCK).

Synopsis: lock = Lock(name, mode);

lock: (BPTR) Actually a pointer to a FileLock structure.

name: (char *) Pointer to a text string which contains the file/directory name.

mode: (long) Accessmode:

```
  SHARED_LOCK:     Other tasks may read the file.
  ACCESS_READ:                 - " -
  EXCLUSIVE_LOCK:  No other tasks may use this f.
  ACCESS_WRITE:                - " -
```

UnLock()

This function unlocks a previously locked file: (Remember to unlock ALL files you have locked!)

Synopsis: Unlock(lock);

lock: (BPTR) Actually a pointer to FileLock structure which has been initialized by a previous Lock() call.

Rename()

This function renames a file or directory. You can even move a file between directories by renaming it. (For example, Rename("df0:Documents/Sale.doc", "df0:Letters/Sale.doc"); will move the file Sale.doc from the directory "Documents" to directory "Letters". Note! You can not rename a file from one volume to another.)

Synopsis: ok = Rename(old_name, new_name);

ok: (long) Actually a Boolean. It is TRUE if AmigaDOS could rename the file/directory, else FALSE which means something went wrong. (Eg. disk write -protected.)

old_name: (char *) Pointer to a string containing the old file/directory name.

new_name: (char *) Pointer to a string containing the new file/directory name.

SetComment

This function attach a comment to a file or directory.

Synopsis: ok = SetComment(name, comment);

ok: (long) Actually a Boolean. It is TRUE if AmigaDOS could attach the new comment, else FALSE which means something went wrong. (Eg. disk write-protected.)

name: (char *) Pointer to a string containing the name of the file/directory you want to attach the comment to.

comment: (char *) Pointer to a string containing the comment. (A comment may be up to 80 characters long.)

SetProtection()

This function alters the protection bits of a file.
You can set following flags:

FIBF_DELETE : the file/directory can not be deleted.
FIBF_EXECUTE : the file can not be executed.
FIBF_WRITE : you can not write to the file.
FIBF_READ : you can not read the file.
FIBF_ARCHIVE : Archive bit.
FIBF_PURE : Pure bit.
FIBF_SCRIPT : Script bit.

(Note! All of the flags are for the moment not working!)

Synopsis: ok = SetProtection(name, mask);

ok: (long) Actually a Boolean. It is TRUE if AmigaDOS could alter the protection bits, else FALSE which means something went wrong. (Eg. disk write-protected.)

name: (char *) Pointer to a string containing the name of the file/directory you want to change the protection bits.

mask: (long) The protection bits. (For example, if you want to make the file/directory not deletable, and that it can not be executed you should set the protection bits: FIBF_DELETE | FIBF_EXECUTE.)

DeleteFile()

This function deletes a file or directory. Remember that a directory must be empty before it can be deleted.

```
  Synopsis: ok = DeleteFile( name );

  ok:       (long) Actually a Boolean. It is TRUE if AmigaDOS
            could delete the file/directory, else FALSE which
            means something went wrong. (Eg. disk write-
            protected, directory not empty etc.)

  name:     (char *) Pointer to a string containing the name
            of the file/directory you want to delete.
```

CreateDir()

```
  This function creates a new directory, AND "locks" is
  automatically. (Remember to unlock the directory later on.)

  Synopsis: lock = CreateDir( name );

  lock:     (BPTR) Actually a pointer to a FileLock structure.
            If lock is equal to NULL, AmigaDOS have not been
            able to create the new directory.

  name:     (char *) Pointer to a string containing the name
            of the new directory.
```

CurrentDir()

```
  This function makes a specified directory "current
  directory". You need to lock the new directory (new_lock)
  before you can make it the current directory. The function
  returns the old current directories lock so you can unlock
  it if necessary.

  Synopsis: old_lock = CurrentDir( new_lock );

  old_lock: (BPTR) Actually a pointer to a FileLock structure.
            It is the old current directory lock.

  new_lock: (BPTR) Actually a pointer to a FileLock structure.
            The new current directory lock.
```

Info()

```
  This function returns information about a specified disk. You
  specify which disk by either lock that disk, or a file/
  directory on that disk.

  Synopsis:  ok = Info( lock, info_data );

  ok:        (long) Actually a Boolean. It is TRUE if AmigaDOS
             could get information about the disk, else FALSE
             which means something went wrong.

  lock:      (BPTR) Actually a pointer to a FileLock structure.
```

info_data: (struct InfoData *) Pointer to an InfoData structure which will be initialized by the Info() function. The problem with this structure is that it must be on a four byte boundary, so you need to use the function AllocMem() to get the right type of memory for the structure. (See Example.)

IoErr()

This function can be used to get more information about an error message. Whenever you have used an AmigaDOS function which did not work properly (you have received an error message), you call this function and it will return an explanation.

Synopsis: error = IoErr();

error: (long) This field contains a flag returned by IoErr() which can be: (I do not think I need to explain what they mean.)

```
ERROR_NO_FREE_STORE
ERROR_TASK_TABLE_FULL
ERROR_LINE_TOO_LONG
ERROR_FILE_NOT_OBJECT
ERROR_INVALID_RESIDENT_LIBRARY
ERROR_NO_DEFAULT_DIR
ERROR_OBJECT_IN_USE
ERROR_OBJECT_EXISTS
ERROR_DIR_NOT_FOUND
ERROR_OBJECT_NOT_FOUND
ERROR_BAD_STREAM_NAME
ERROR_OBJECT_TOO_LARGE
ERROR_ACTION_NOT_KNOWN
ERROR_INVALID_COMPONENT_NAME
ERROR_INVALID_LOCK
ERROR_OBJECT_WRONG_TYPE
ERROR_DISK_NOT_VALIDATED
ERROR_DISK_WRITE_PROTECTED
ERROR_RENAME_ACROSS_DEVICES
ERROR_DIRECTORY_NOT_EMPTY
ERROR_TOO_MANY_LEVELS
ERROR_DEVICE_NOT_MOUNTED
ERROR_SEEK_ERROR
ERROR_COMMENT_TOO_BIG
ERROR_DISK_FULL
ERROR_DELETE_PROTECTED
ERROR_WRITE_PROTECTED
ERROR_READ_PROTECTED
ERROR_NOT_A_DOS_DISK
ERROR_NO_DISK
ERROR_NO_MORE_ENTRIES
```

11.9 EXAMPLES

The printed version of these examples can be found in Appendix A (Part IV: Appendices).

Example1

This program collects ten integer values from the user, and saves them in a file ("HighScore.dat") on the RAM disk. The memory is then cleared, and the file cursor is moved to the beginning of the file. The file is then loaded into the memory again, and printed out.

Example2

This example demonstrates how to create a directory called "MyDirectory" on the RAM disk.

Example3

This example demonstrates how to rename files and directories. It will rename the file Example 1 created (called "HighScore.dat") to "Numbers.dat". It will also rename the directory Example 2 created ("MyDirectory") to "NewDirectory".

Example4

This example demonstrates how to delete files and directories. It will delete the file Example 1 and directory Example 2 created. (The file and directory are supposed to have been renamed by Example 3).

Example5

This example demonstrates how to attach a short comment to a file. A short file called "Letter.doc" will be created, and a short comment will be attached. To see the comment use the CLI command "List".

Example6

This example demonstrates how to protect and unprotect files. The file Example 5 created ("Letter.doc") will be protected, and we will then try to delete it (unsuccessfully). We will then unprotect the file and then try to delete it (successfully).

Example7

This program takes a file/directory/device name as parameter, and prints out some interesting information about it.

Example8

This program takes a directory/device name as parameter, and prints out all the file/directory-names inside it. This example describes how to use Examine() and ExNext().

12 LOW LEVEL GRAPHICS ROUTINES

12.1 INTRODUCTION

In this chapter will we look at Amiga's low level graphics routines. As you saw in chapter 3 (GRAPHICS) the Amiga offers two different levels of graphics support. You can either use the high level graphics routines that use structures in order to give your program flexibility, or you can use the low level graphics routines that give your program total control over the graphics system. We will in this chapter look at the low level graphics routines.

The low level graphics routines can be divided into two sections. The first section is about creating a display. We here describe how to chose the screen resolution, how many colours, interlaced or non interlaced, the size of the viewports etc. In the second section is about all different types of drawing tools the Amiga offers. We will here describe how to draw a single dot, lines, polygons, how to change colours, how to use patterns and masks, and look at different types of filling routines etc.

12.2 CREATE A DISPLAY

We will now take a look at how to create a display. However, before we start with that it is best to give some general information about computers and graphics.

12.2.1 GENERAL INFORMATION

We will here describe how a picture is generated on a display. We will also look at some special display modes such as "Interlaced", high respectively low resolution etc. Finally we will discuss what a pixel is and what Bitplanes are used for.

12.2.1.1 HOW A MONITOR/TV WORK

The video picture that is displayed on a monitor/TV is actually "painted" by a small video beam. The video beam starts at the top left corner of the screen. While it is moved to the right the intensity of the beam is varied so lighter and darker spots

are drawn. Once the beam has reached the right side of the screen, it is moved back to the left side and positioned a bit further down. It is then moved once again to the right side, and so on.

On an American display (NTSC) the beam is moved from the left to the right 262 times (On an European PAL display 312 times). The beam has now reached the bottom of the screen and is moved back to the top left corner where the process is repeated. The screen is painted like this 60 times a second on a NTSC monitor, and 50 times a second on a PAL monitor.

The video picture on an American (NTSC) monitor:

```
    1  ->---->----> |
    2  ->---->----> |
    3  ->---->----> |
    |  and so on... |
  261  ->---->----> |
  262  ->---->----> V
```

As you can see you can have a display that is up to 262 (312) lines tall, but normally you do not use the very first and last lines since they can be hard to see. You are recommended to use a display of maximum 200 (256 PAL) lines. However, if you want to use special video effects, or make the screen to cover the whole display, all 262 (312) lines can be used. This special effect is normally referred as "Overscan".

12.2.1.2 INTERLACED

A full sized normal American (NTSC) display is 200 lines tall (PAL 256 lines). However, the Amiga has a unique feature that can double the amount of lines on the same display to 400 (PAL 512). This special mode is called Interlace.

The first time the video beam is moved over the screen, all odd lines are drawn (1, 3, 5, 7 ... 397, 399), and the second time all even lines are drawn (2, 4, 6, 8 .. 398, 400). This special mode can cause the screen to "flicker" especially if the contrast is high. This is because all odd lines are drawn 30 (PAL 25) times a second, and all even lines 30 (25) times a second. If you have a high phosphor screen you will not have any problems, but on cheap normal monitors the flickering can be very annoying.

```
First time      Second time
     1 -------------
       ------------- 2
     3 -------------
       ------------- 4
     5 -------------
       ------------- 6
     7 -------------
```

```
------------- 8
and so on ...
```

12.2.1.3 HIGH AND LOW RESOLUTION

There exist two different types of horizontal resolution. Low resolution gives 320 pixels across each line, and high resolution doubles it to 640 pixels.

There is an important difference between these two display modes. In low resolution you can have up to 32 colours or even use HAM or Extra Half Bright (explained later), but in high resolution you may only use 16 colours or less.

You may even here use the special display mode "Overscan". The display may then be up to 352 pixels wide in low resolution, and 704 pixels in high resolution.

12.2.1.4 PIXELS

The display is built up of small dots called "Pixels". As described above there may be either 320 or 640 pixels across each line, and 200 or 400 lines (PAL 256 or 512 lines).

When you create a display you allocate rectangular memory areas that are called "Bitplanes". Each bit in that memory area represent one pixel.

A high resolution non interlaced American display, 640 * 200 pixels will need 16,000 bytes of memory for one bitplane.

To create a very small screen (8 * 9 pixels) that will display an "A" the bitplane could look something like this:

```
  Line  Bitplane 1
  ----------------
     1  00000000
     2  00111100
     3  01000010
     4  01000010
     5  01111110
     6  01000010
     7  01000010
     8  01000010
     9  00000000
```

12.2.1.5 COLOURS

If you want to use several colours on the screen you need to use several bitplanes. All bits on the same location in each bitplane will then represent one pixel. With two bitplanes you can have four different combinations for every pixel. Three bitplanes gives eight combinations. Four bitplanes sixteen combinations, and finally five bitplanes thirty two combinations.

Here is an example. We have reserved memory for two bitplanes that will build up a very small display of 10 pixels each line, and four lines.

Line	Bitplane 1	Bitplane 2	Display (Colour)
1	0000000000	0000000000	0000000000
2	1111111111	0000000000	1111111111
3	0000000000	1111111111	2222222222
4	1111111111	1111111111	3333333333

The first line is made up of ten pixels in colour 0 (00[b]=0[d])
The second line is made up of ten pixels in colour 1 (01[b]=1[d])
The third line is made up of ten pixels in colour 2 (10[b]=2[d])
The fourth line is made up of ten pixels in colour 3 (11[b]=3[d])

The more bitplanes you have the more colours you can display. However, since you need more bitplanes you would need to allocate more memory.

Bitplanes	Colours
1	2
2	4
3	8
4	16
5	32

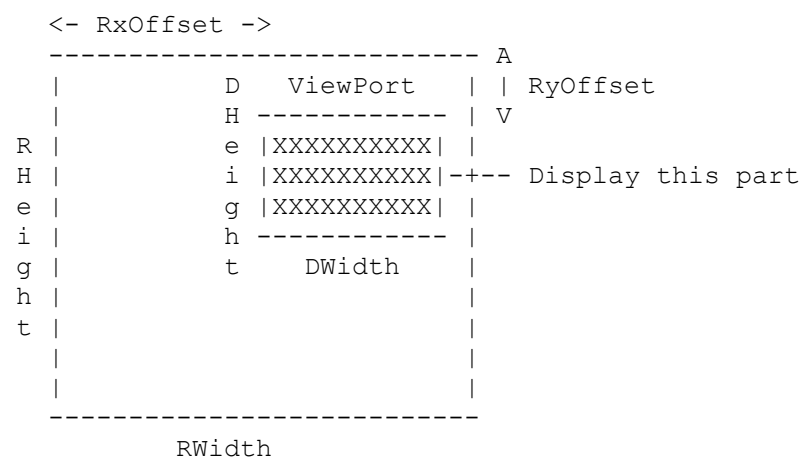
12.2.2 DISPLAY ELEMENTS

12.2.2.1 RASTER

The are which you may draw on is called Raster. It consists of several BitMaps on top of each other. The more Bitmaps the more colours may be used at the same time. The Raster may be up to 1024 pixels wide (RWidth), and 1024 pixels high (RHeight).

The part of the Raster that will be displayed is called ViewPort. A structure called RasInfo contains the RxOffset and RyOffset values which determines what part of the Raster should be displayed in the ViewPort. The ViewPort is DWide pixels wide, and DHeight pixels high. (Simple, isn't it?)

RASTER:

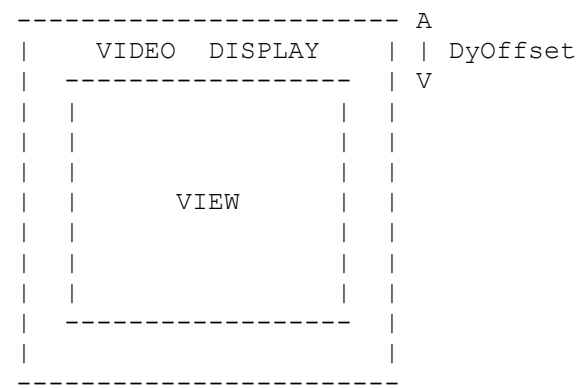


12.2.2.2 VIEW

The video display has an are in which you may put one or more ViewPorts. That area is called View, and the DxOffset and DyOffset values in the View structure determines where on the video display the View should be positioned.

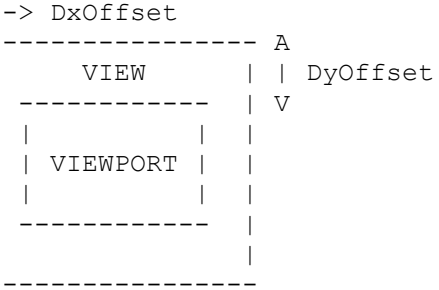
You can set the DxOffset and DyOffset to negative values. The View will then be more to the left and higher up than normal. The top left part of the drawing can be hard to see and this mode is therefore not recommended for business programs. However, if you write games or video programs this special display mode can be very useful.

VIEW:

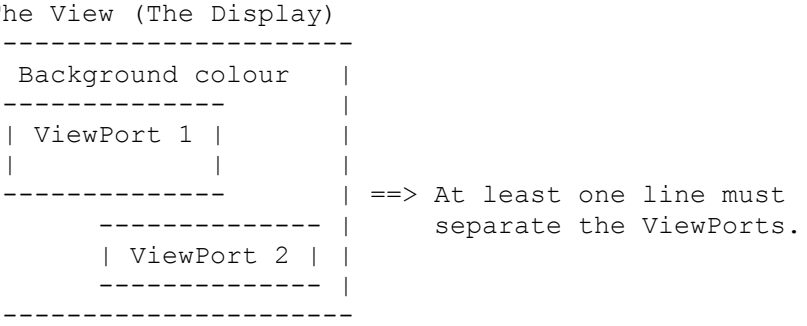


12.2.2.3 VIEWPORT

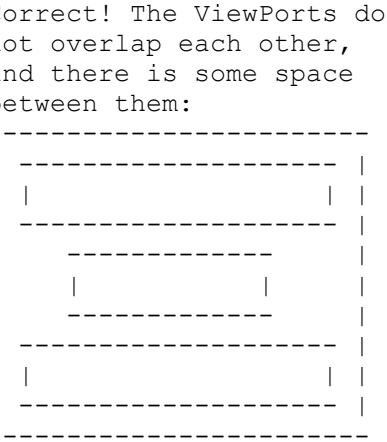
The View may, as said above, contain one or more ViewPorts. The DxOffset and DyOffset values in the ViewPort structure determines where on the View the ViewPort should be positioned.



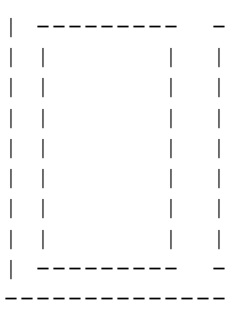
There are some important restriction on how you may place the ViewPorts. The ViewPorts must be placed under each other, with at least one or more lines apart. While the video beam has drawn the last line of one ViewPort, the Amiga needs some time to adjust to the new resolution etc in the next ViewPort. While the Amiga is working with the next screen, the video beam will have travelled down at least one line (maybe more).



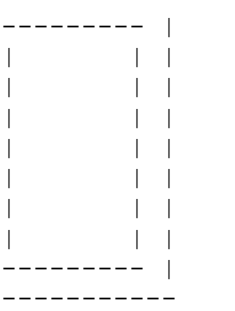
The are around the ViewPorts are filled with the ViewPort's background colour.
Here are three examples:



Incorrect! The ViewPorts must be placed under each other, not beside each other.



Incorrect! There must be some space between ViewPorts. (At least one line, maybe more.)



12.2.2.4 BITMAP

The area on which you may draw on is, as said above, called Raster. The Raster itself is built up of one or more BitMaps. A BitMap is a rectangular memory area. Each bit in that area represent one small dot (pixel). You may use several BitMaps which means that each dot is built up of several bits (one bit for every BitMap), and the binary value of these bits determines what colour should be used.

BitMap 0	BitMap 1	BitMap 2	Colour
0000011111	0000000000	0000000000	0000011111
0000011111	1111111111	0000000000	2222233333
0000011111	0000000000	1111111111	4444455555
0000011111	1111100000	1111111111	6666677777

B I N A R Y T O D E C I M A L

Binary	Decimal	Binary	Decimal
00000	0	10000	16
00001	1	10001	17
00010	2	10010	18
00011	3	10011	19
00100	4	10100	20
00101	5	10101	21
00110	6	10110	22
00111	7	10111	23
01000	8	11000	24
01001	9	11001	25
01010	10	11010	26
01011	11	11011	27

```
|                  |                  |
| 01000        12  |  11000        28 |
| 01101        13  |  11101        29 |
| 01110        14  |  11110        30 |
| 01111        15  |  11111        31 |
---------------------------------------
```

The more BitMaps the more colours can be used. The Amiga allows you to have up to six bitplanes with some restrictions:

BitMaps	Colours	Limitations
1	2	
2	4	
3	8	
4	16	Single playfield
5	32	- " - low resolution
6	64	- " - - " - Half bright
6	4096	-"- - " - HAM

12.2.3 CREATE A DISPLAY

When you create a display you first have to declare and initialize some important structures. These structures are:

1. View structure
2. ViewPort structure
3. ColourMap structure
4. BitMap structure
5. RasInfo structure

Once these structures are initialized you call three functions [MakeVPort(), MrgCop() and LoadView()] and your new display is on.

12.2.3.1 VIEW

A View can consist of one or more ViewPorts. You can also have several Views at the same time in the Amiga's memory, but only one View can be showed at one time. The advantage of several Views is that you can show the user one picture (one View) while the other picture is drawn (in the other View.)

The View itself has some important variables that determines the position of the View, if the view should be interlaced or not, and pointers to the first ViewPort etc.

12.2.3.1.1 VIEW STRUCTURE

The View structure look like this: [Defined in file: "view.h"]

```
struct View
{
  struct ViewPort *ViewPort;
  struct cprlist *LOFCprList;
  struct cprlist *SHFCprList;
  short DyOffset, DxOffset;
  UWORD Modes;
};
```

ViewPort: Pointer to the first ViewPort in the display.

LOFCprList: Pointer to a cprlist structure that is used by the Copper (explained later).

SHFCprList: Pointer to a cprlist structure that is used by the Copper if the structure is interlaced. (If the display is interlaced, both LOFCprList and SHFCprList is used.)

DyOffset: Y position of the whole display.

DxOffset: X position of the whole display.

Modes: If the display should be interlaced set the flag LACE. If the display should be merged together with a external video signal with help of a Genlock set the flag GENLOCK_VIDEO.

12.2.3.1.2 PREPARE A VIEW STRUCTURE

After you have declared a View structure you have to prepare it. To do it you call the InitView() function with a pointer to the View structure as the only parameter:

```
struct View my_view;  /* Declare the View structure. */

InitView( &my_view ); /* Prepare it. */
```

If you want to position you View somewhere on the display you need to set the DxOffset and DyOffset accordingly. For example:

```
my_view.DxOffset = 20; /* 20 pixels out. */
my_view.DyOffset = 50; /* 50 lines down. */
```

If you want to use an interlaced display you should also set the "LACE" flag. For example: my_view.Modes = LACE;

12.2.3.2 VIEWPORTS

As said above, each View can consists of one or more ViewPorts. Each ViewPort can have its own resolution, number of colours and size.

12.2.3.2.1 VIEWPORT STRUCTURE

The ViewPort structure look like this: [Defined in file: "view.h"]

```
struct ViewPort
{
  struct ViewPort *Next;
  struct ColorMap *ColorMap;
  struct CopList *DspIns;
  struct CopList *SprIns;
  struct CopList *ClrIns;
  struct UCopList *UCopIns;
  SHORT DWidth, DHeight;
  SHORT DxOffset, DyOffset;
  UWORD Modes;
  UBYTE SpritePriorities;
  UBYTE reserved;
  struct RasInfo *RasInfo;
};
```

Next:	Pointer to the next ViewPort in the View if there exist more, else NULL.
ColorMap:	Pointer to a ColorMap structure used for this ViewPort. (See below for more information about the ColorMap structure.
DspIns:	Used by the Copper.
SprIns:	Special colour instructions for sprites.
ClrIns:	Special colour instructions for sprites.
UCopIns:	Used by the Copper.
DWidth:	Horizontal size of the ViewPort.
DHeight:	Vertical size of the ViewPort.
DxOffset:	X position of the ViewPort inside the View.
DyOffset:	Y position of the ViewPort inside the View.
Modes:	This ViewPort's display mode. Following flags can be used:

HIRES: Set this flag if you want to use a high resolution ViewPort, else low resolution will be used.

LACE: Set this flag if you want the ViewPort to be interlaced, else it will be non interlaced.

SPRITES: Set this flag if you will use sprites in this ViewPort.

HAM: Set this flag if you want to use the special display mode "Hold And Modify". The display must be in low resolution, and use six bitplanes. Allows all 4096 colours to be displayed at the same time.

EXTRA_HALFBRITE: Set this flag if you want to use the special display mode "Extra Half Bright". The display must be in low resolution, and use six bitplanes. Allows you to use 64 colours.

DUALPF: Set this flag if you want to use dual playfields.

PFBA: Set this flag if you want that playfield 1 should be behind playfield 2. If this flag is not set, the priorities will be reversed.

GENLOCK_VIDEO: Set this flag if the display should be mixed together with a video signal with help of a Genlock.

Note! You can combine several of these modes if you like. Just remember to put a "|" between each flag. For example: To create a high resolution interlaced ViewPort you set the mode to "HIRES|LACE".

SpritePriorities: The priority of this ViewPort's sprites.

reserved: Which sprites are reserved, and can not be used. (See chapter SPRITES and VSPRITES for more information.)

RasInfo: Pointer to a RasInfo structure.

12.2.3.2.2 PREPARE A VIEWPORT STRUCTURE

After you have declared a ViewPort structure you have to prepare it. (Same as with all View structures.) To do it you call the InitVPort() function with a pointer to the ViewPort structure as the only parameter:

```
/* Declare the View structure: */
struct ViewPort my_view_port;

/* Prepare it: */
InitVPort( &my_view_port );
```

After you have prepared it you set the desired values: (An example)

```
my_view_port.Next = NULL;            If we have several ViewPorts
                                     in the same View we link
                                     them together, else NULL.

my_view_port.DWidth = WIDTH;         Set the desired width.
my_view_port.DHeight = HEIGHT;       Set the desired height.

my_view_port.DxOffset = 0;           X and Y position of the
my_view_port.DyOffset = 0;           ViewPort in the View.

my_view_port.Modes = HIRES|LACE;     Set desired flags. (Eg
                                     high resolution interlace)

my_view_port.RasInfo = &my_ras_info; Pointer to this ViewPort's
                                     RasInfo.
```

12.2.3.3 COLORMAP

Each ViewPort has a its own lists of colours. They are stored in a ColorMap structure which contains information like: how many colours this display use, what colours, and each colours individual RGB values. (Red + Green + Blue intensity)

12.2.3.3.1 COLORMAP STRUCTURE

The ColorMap structure look like this: [Defined in file: "view.h"]

```
struct ColorMap
{
  UBYTE Flags;
  UBYTE Type;
  UWORD Count;
  APTR ColorTable;
};
```

12.2.3.3.2 DECLARE AND INITIALIZE A COLORMAP STRUCTURE

To declare and initialize a ColorMap structure you simply call the GetColorMap() function. It will take care of everything, and the only thing you need to tell the function is how many colours you want to use.

Get a colour map, and link it to the ViewPort:

```
my_view_port.ColorMap = GetColorMap( 32 );
```

You should also remember to check if you have got the the colour map or not. For example:

```
if( my_view_port.ColorMap == NULL )
  clean_up(); /* leave */
```

12.2.3.3.3 SET THE RGB VALUES

Once you have got your ColorMap structure, you need to set the RGB (Red, Green, Blue) values for each colour. Each colour can have a mixture of red, green and blue, on a scale from 0 to 15. Hexadecimal that would be from 0x0 to 0xF.

Here are some examples:

RGB value	Colour
0xF00	Red
0x0F0	Green
0x00F	Blue
0x600	Dark red
0x060	Dark green
0x006	Dark blue
0xFF0	Yellow
0x000	Black
0x444	Dark grey
0x888	Light grey
0xFFF	White

Since each RGB values can have sixteen intensities, you can define up to 16 * 16 * 16 = 4096 different colours.

If you prepare a ColorMap structure to use four colours,
you need to give it a colour table of RGB values.

```
/* 1. Declare a RGB colour table: */
UWORD my_color_table[] =
{
  0x000, /* Black */
  0xF00, /* Red */
  0x0F0, /* Green */
  0x00F, /* Blue */
};

/* 2. Declare a pointer: */
UWORD *pointer;

/* 3. Set the pointer so it points to the colour table: */
pointer = (UWORD *) my_view_port.ColorMap->ColorTable;

/* 4. Set the colours: */
for( loop = 0; loop < 4; loop++ )
  *pointer++ = my_color_table[ loop ];
```

12.2.3.3.4 DEALLOCATE THE COLOURMAP

When your program closes a ViewPort you must remember to
deallocate the memory used for the ColourMap. You deallocate
a ColourMap with help of the FreeColorMap() function:

```
FreeColorMap( my_view_port.ColorMap );
```

12.2.3.4 BITMAP

The BitMap is, as said above, a rectangular memory are were
each bit represents one pixel. By combining two or more BitMaps
each pixel is represented by a combination of two or more bits
which determines what colour should be used. A pointer to each
BitMap, information about the size of the BitMaps etc are all
stored in the BitMap structure.

12.2.3.4.1 BITMAP STRUCTURE

The BitMap structure look like this: [Defined in file: "gfx.h"]

```
struct BitMap
{
  UWORD BytesPerRow;
  UWORD Rows;
```

```
  UBYTE Flags;
  UBYTE Depth;
  UWORD pad;
  PLANEPTR Planes[ 8 ];
};
```

BytesPerRow: How many bytes are used on every row.

Rows: How many rows there are.

Flags: Special information about the BitMaps (for the moment not used.)

Depth: How many BitPlanes.

pad: Extra space.

Planes: A list of eight pointers. (PLANEPTR is a pointer to a BitMap.)

12.2.3.4.2 DECLARE AND INITIALIZE A BITMAP STRUCTURE

You declare the structure as normal, and initialize it with help of the function InitBitMap(). You need to give the function a pointer to your BitMap structure and information about the size and depth (how many BitPlanes used).

Synopsis: InitBitMap(bitmap, depth, width, height);

bitmap: (struct BitMap *) Pointer to the BitMap.

depth: (long) How many BitPlanes used.

width: (long) The width of the raster.

height: (long) The height of the raster.

Information about the size and depth of a BitMap etc is used in many places in a program and. I recommend you therefore to define some constants at the top of the program and use them rather than writing the values directly. This makes it much easier to change anything since you only need to change the code at one place, and many writing errors are avoided.

Example:

```
#define WIDTH  320 /* 320 pixels wide.              */
#define HEIGHT 200 /* 200 lines tall.               */
#define DEPTH    5 /* 5 BitPlanes gives 32 colours. */

struct BitMap my_bit_map;

InitBitMap( &my_bit_map, DEPTH, WIDTH, HEIGHT );
```

12.2.3.4.3 ALLOCATE RASTER

Once the BitMap structure has been initialized it is time to reserve memory for each BitPlane. You reserve display memory with help of the AllocRaster() function:

Synopsis: pointer = AllocRaster(width, height);

pointer (PLANEPTR) Pointer to the allocated memory or NULL if enough memory could not be reserved.

width: (long) The width of the BitMap.

height: (long) The height of the BitMap.

Since you may need to reserve several BitPlanes it is best to include the AllocRaster() function in a loop. Remember to check that you have got the memory you asked for! Finally, the memory you get is probably filled with a lot of trash, and it is best to clear it before you start to use it. The fastest way to clear large rectangular memory areas is to use the Blitter, and you do it by calling the function BltClear():

Synopsis: BltClear(pointer, bytes, flags);

pointer (char *) Pointer to the memory.

bytes: (long) The lower 16 bits tells the blitter how many bytes per row, and the upper 16 bits how many rows. This value is automatically calculated for you with help of the macro RASSIZE(). Just give RASSIZE() the correct width and height and it will return the correct value. [RASSIZE() is defined in file "gfx.h".]

flags: (long) Set bit 0 to force the function to wait until the Blitter has finished with your request.

Here is an example on how to allocate display memory, check it and clear it: (Remember that you muse deallocate ALL memory you have allocated! See below for more information.)

```
for( loop = 0; loop < DEPTH; loop++ )
{
  my_bit_map.Planes[ loop ] = AllocRaster( WIDTH, HEIGHT );

  if( my_bit_map.Planes[ loop ] == NULL )
    clean_up(); /* Leave */

  BltClear( my_bit_map.Planes[ loop ], RASSIZE( WIDTH, HEIGHT ), 0 );
}
```

12.2.3.5 RASINFO

The RasInfo structure contains information like were the BitMap structure is, and the Raster's x/y offset. If you use the special display mode "Dual Playfields" it will also contain a pointer to the second Playfield.

12.2.3.5.1 RASINFO STRUCTURE

The RasInfo structure look like this: [Defined in file: "view.h"]

```
struct RasInfo
{
  struct RasInfo *Next;
  struct BitMap *BitMap;
  SHORT RxOffset, RyOffset;
};
```

Next: If you use the special display mode "Dual Playfields" this pointer will contain the address to the second playfield. (Playfield one RasInfo's structure will have a pointer to the second Playfield's RasInfo structure which pointer will point to NULL.)

BitMap: Pointer to the BitMap.

RxOffset: X offset of the Raster. (By changing the RxOffset and RyOffset values you can scroll the whole Raster very fast.)

RyOffset: Y offset of the Raster.

12.2.3.5.2 DECLARE AND INITIALIZE A RASINFO STRUCTURE

Here is an example on how to declare and initialize a RasInfo structure:

```
/* Declare a RasInfo structure: */
struct RasInfo my_ras_info;

/* Prepare the RasInfo structure: */
my_ras_info.BitMap = &my_bit_map; /* Pointer to the BitMap   */
                                  /* structure.              */
my_ras_info.RxOffset = 0;         /* The top left corner of  */
my_ras_info.RyOffset = 0;         /* the Raster should be at */
                                  /* the top left corner of  */
                                  /* the display.            */
my_ras_info.Next = NULL;          /* Single playfield - only */
```

```
                    /* one RasInfo structure   */
                    /* is necessary.           */
```

12.2.3.6 MAKEVPORT()

Once you have initialized all necessary structures you need to prepare the Amiga's hardware (especially the Copper) so they can create the display you have asked for. Each ViewPort can have its own resolution, size, colour list etc and you must therefore prepare every ViewPort by calling the MakeVPort() function.

Synopsis: MakeVPort(view, viewport);

view: (struct View *) Pointer to the ViewPort's View.

viewport: (struct ViewPort *) Pointer to the ViewPort.

NOTE! You have to prepare EVERY ViewPort you are going to use!

12.2.3.7 MRGCOP()

All ViewPort's special display instructions together with other display instructions used by the Sprite hardware etc must be combined before you can show the display. All these instructions are combined with one single function call: MrgCop(). MrgCop() will merge all coprocessors' display instructions into one list.

Synopsis: MrgCop(view);

view: (struct View *) Pointer to the View.

12.2.3.8 LOADVIEW()

You can now at last show the display. You do it by calling the LoadView() function:

Synopsis: LoadView(view);

view: (struct View *) Pointer to the View.

You have to remember that you have now created your own display and when your program terminates you MUST return the old display! You must therefore store a pointer to the old display before you turn on your own. You can find a pointer to the current View in the GfxBase structure.

```
struct View *old_view;

old_view = GfxBase->ActiView;
```

12.2.4 CLOSE A DISPLAY

When you close your display you have to remember to restore the old view by calling the LoadView() structure with a pointer to the old View structure: LoadView(old_view);

You must also return some memory that was automatically reserved when you called the MakeVPort() and MrgCop() functions. You should therefore call the FreeVPortCopLists() function for every ViewPort you have created.

Synopsis: FreeVPortCopLists(viewport);

view: (struct ViewPort *) Pointer to the ViewPort.

You must also call the FreeCprList() function for every View you have created. You give the function a pointer to the cprlist structure (LOFCprList). If you have created an interlaced View you must also free a special cprlist structure (SHFCprList) which is only used for interlaced Views.

Synopsis: FreeCprList(cprlist);

cprlist: (struct cprlist *) Pointer to the View's cprlist (LOFCprList) structure. If the View was interlaced you must also call the FreeCprList function with a pointer to the SHFCprList.

The display memory you have allocated for the Raster must also be deallocated. You deallocate the Raster by calling the FreeRaster() function with a pointer to one Bitplane. You must call this function for every Bitplane you have allocated!

Synopsis: FreeRaster(bitplane, width, height);

bitplane: (PLANEPTR) Pointer to a Bitplane.

width: (long) The Bitplane's width.

height: (long) The Bitplane's height.

Finally you have to deallocate the ColorMap structure that was automatically allocated and initialized by the GetColorMap() function. You deallocate a colour map by calling the FreeColorMap() function.

```
Synopsis: FreeColorMap( colormap );

colormap: (struct ColorMap *) Pointer to the ColorMap structure.
```

Here is an example on how to close a display:

```
/* 1. Restore the old View: */
LoadView( old_view );

/* 2. Deallocate the ViewPort's display instructions: */
FreeVPortCopLists( &my_view_port );

/* 3. Deallocate the View's display instructions: */
FreeCprList( my_view.LOFCprList );

/* 4. Deallocate the display memory, BitPlane for BitPlane: */
for( loop = 0; loop < DEPTH; loop++ )
  FreeRaster( my_bit_map.Planes[ loop ], WIDTH, HEIGHT );

/* 5. Deallocate the ColorMap: */
FreeColorMap( my_view_port.ColorMap );
```

12.2.5 EXAMPLE

Here is an example on how to open a high resolution NTSC (American) display. This is what has to be done:

```
 1. Prepare the View.
 2.     - " -   ViewPort.
 3.     - " -   ColorMap.
 4.     - " -   BitMap.
 5.     - " -   RasInfo.
 6. Prepare the Amiga with MakeVPort() and MrgCop().
 8. Show the new View.
 9. The View is on and it is now up to you to do what you want.
10. Restore the old View.
11. Free all allocated resources and leave.
```

```
#include <intuition/intuition.h>
#include <graphics/gfxbase.h>


#define WIDTH  640 /* 640 pixels wide (high resolution)    */
#define HEIGHT 200 /* 200 lines high (non interlaced NTSC) */
#define DEPTH    3 /* 3 BitPlanes, gives eight colours.    */
#define COLOURS  8 /* 2^3 = 8                              */


struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
```

```
struct View my_view;
struct View *my_old_view;
struct ViewPort my_view_port;
struct RasInfo my_ras_info;
struct BitMap my_bit_map;
struct RastPort my_rast_port;


UWORD my_color_table[] =
{
  0x000, /* Colour 0, Black       */
  0x800, /* Colour 1, Red         */
  0xF00, /* Colour 2, Light red   */
  0x080, /* Colour 3, Green       */
  0x0F0, /* Colour 4, Light green */
  0x008, /* Colour 5, Blue        */
  0x00F, /* Colour 6, Light Blue  */
  0xFFF, /* Colour 7, White       */
};


void clean_up();
void main();


void main()
{
  UWORD *pointer;
  int loop;


  /* Open the Intuition library: */
  IntuitionBase = (struct IntuitionBase *)
    OpenLibrary( "intuition.library", 0 );
  if( !IntuitionBase )
    clean_up( "Could NOT open the Intuition library!" );

  /* Open the Graphics library: */
  GfxBase = (struct GfxBase *)
    OpenLibrary( "graphics.library", 0 );
  if( !GfxBase )
    clean_up( "Could NOT open the Graphics library!" );


  /* Save the current View, so we can restore it later: */
  my_old_view = GfxBase->ActiView;


  /* 1. Prepare the View structure, and give it a pointer to */
  /*    the first ViewPort:                                  */
  InitView( &my_view );
  my_view.ViewPort = &my_view_port;


  /* 2. Prepare the ViewPort structure, and set some important values: */
  InitVPort( &my_view_port );
```

```
my_view_port.DWidth = WIDTH;         /* Set the width.                */
my_view_port.DHeight = HEIGHT;       /* Set the height.               */
my_view_port.RasInfo = &my_ras_info; /* Give it a pointer to RasInfo. */
my_view_port.Modes = HIRES;          /* High resolution.              */


/* 3. Get a colour map, link it to the ViewPort, and prepare it: */
my_view_port.ColorMap = (struct ColorMap *) GetColorMap( COLOURS );
if( my_view_port.ColorMap == NULL )
  clean_up( "Could NOT get a ColorMap!" );

/* Get a pointer to the colour map: */
pointer = (UWORD *) my_view_port.ColorMap->ColorTable;

/* Set the colours: */
for( loop = 0; loop < COLOURS; loop++ )
  *pointer++ = my_color_table[ loop ];


/* 4. Prepare the BitMap: */
InitBitMap( &my_bit_map, DEPTH, WIDTH, HEIGHT );

/* Allocate memory for the Raster: */
for( loop = 0; loop < DEPTH; loop++ )
{
  my_bit_map.Planes[ loop ] = (PLANEPTR) AllocRaster( WIDTH, HEIGHT );
  if( my_bit_map.Planes[ loop ] == NULL )
    clean_up( "Could NOT allocate enough memory for the raster!" );

  /* Clear the display memory with help of the Blitter: */
  BltClear( my_bit_map.Planes[ loop ], RASSIZE( WIDTH, HEIGHT ), 0 );
}


/* 5. Prepare the RasInfo structure: */
my_ras_info.BitMap = &my_bit_map; /* Pointer to the BitMap structure.  */
my_ras_info.RxOffset = 0;         /* The top left corner of the Raster */
my_ras_info.RyOffset = 0;         /* should be at the top left corner  */
                                  /* of the display.                   */
my_ras_info.Next = NULL;          /* Single playfield - only one       */
                                  /* RasInfo structure is necessary.   */

/* 6. Create the display: */
MakeVPort( &my_view, &my_view_port );
MrgCop( &my_view );


/* 7. Prepare the RastPort, and give it a pointer to the BitMap. */
InitRastPort( &my_rast_port );
my_rast_port.BitMap = &my_bit_map;


/* 8. Show the new View: */
LoadView( &my_view );


/* 9. Your View is displayed and you can  */
```

```
  /*    now do whatever you want with it.   */


  /* 10. Restore the old View: */
  LoadView( my_old_view );


  /* 11. Free all allocated resources and leave. */
  clean_up( "THE END" );
}


/* Returns all allocated resources: */
void clean_up( message )
STRPTR message;
{
  int loop;

  /* Free automatically allocated display structures: */
  FreeVPortCopLists( &my_view_port );
  FreeCprList( my_view.LOFCprList );

  /* Deallocate the display memory, BitPlane for BitPlane: */
  for( loop = 0; loop < DEPTH; loop++ )
    if( my_bit_map.Planes[ loop ] )
      FreeRaster( my_bit_map.Planes[ loop ], WIDTH, HEIGHT );

  /* Deallocate the ColorMap: */
  if( my_view_port.ColorMap ) FreeColorMap( my_view_port.ColorMap );

  /* Close the Graphics library: */
  if( GfxBase ) CloseLibrary( GfxBase );

  /* Close the Intuition library: */
  if( IntuitionBase ) CloseLibrary( IntuitionBase );

  /* Print the message and leave: */
  printf( "%s\n", message );
  exit();
}
```

12.3 DRAW

We have described how to create and open a display, and we will now look at all different drawing routines the Amiga offers. The Amiga allows you to draw single dots and lines as well as complex figures. These routines can be combined with masks and multicoloured patterns to great effect. The Amiga offers also three different ways of drawing filled objects. You can even scroll, copy and logically combine rectangular areas with help of the fast Blitter.

12.3.1 RASTPORT

Before you can start using the drawing routines you still need to declare and initialize one more structure. ("Not one more!" I can hear you say.) The reason why we need one more structure is that most of the drawing routines need to know what pen colour, pattern and mask (described later in this chapter) you have chosen. This kind of information is therefore stored in a structure called RastPort.

12.3.1.1 RASTPORT STRUCTURE

The RastPort structure look like this: [Defined in file: "rastport.h"] (Luckily you normally do not need to bother too much about this structure.)

```
struct RastPort
{
  struct Layer *Layer;
  struct BitMap *BitMap;
  USHORT *AreaPtrn;
  struct TmpRas *TmpRas;
  struct AreaInfo *AreaInfo;
  struct GelsInfo *GelsInfo;
  UBYTE Mask;
  BYTE FgPen;
  BYTE BgPen;
  BYTE AOlPen;
  BYTE DrawMode;
  BYTE AreaPtSz;
  BYTE linpatcnt;
  BYTE dummy;
  USHORT Flags;
  USHORT LinePtrn;
  SHORT cp_x, cp_y;
  UBYTE minterms[8];
  SHORT PenWidth;
  SHORT PenHeight;
  struct TextFont *Font;
  UBYTE AlgoStyle;
  UBYTE TxFlags;
  UWORD TxHeight;
  UWORD TxWidth;
  UWORD TxBaseline;
  WORD TxSpacing;
  APTR *RP_User;
  ULONG longreserved[2];
  UWORD wordreserved[7];
  UBYTE reserved[8];
};
```

Layer: Pointer to a Layer structure.

BitMap: Pointer to this RastPort's BitMap structure.

AreaPtrn: Pointer to an array of USHORTs that is used to create the area fill pattern.

TmpRas: Pointer to a TmpRas structure. Used by the AreaMove(), AreaDraw() and AreaEnd() functions. (See below for more information.)

AreaInfo: Pointer to an AreaInfo structure. Also used by the AreaMove(), AreaDraw() and AreaEnd() functions.

GelsInfo: Pointer to a GelsInfo structure. Used by animation routines (VSprites and BOSs).

Mask: This mask allows you determine which BitPlanes should be affected by the drawing routines. The first bit represents BitPlane zero, the second bit represents BitPlane one and so on. If the bit is on (1) that BitPlane will be affected. If the bit is off (0) that BitPlane will not be affected by the drawing routines.

For example, if the mask value is set to 0xFF (0xFF hexadecimal = 11111111 binary) all BitPlanes will be affected by the drawing routines. If the mask value is set to 0xF6 (0xF6 hexadecimal = 11110110 binary) BitPlane zero and three will not be affected.

FgPen: Foreground pen's colour.

BgPen: Backgrounds pen's colour.

AOlPen: Areaoutline pen's colour.

DrawMode: There exist four different types of drawing modes:

JAM1 Where you write the foreground pen will be used to replace the old colour, and where you do not write the background is unchanged. For example, the FgPen is set to colour five and draw mode to JAM1. If you then write a pixel that pixel would be of colour five, if you write a character that character would be of colour five and the background unchanged. (One colour jammed into the Raster.)

JAM2 Where you write the foreground pen will be used to replace the old colour, and where you do not write the back-

ground pen will be used. For example, the FgPen is set to colour five BgPen to colour six and draw mode to JAM2. If you then write a pixel that pixel would be of colour five, if you write a character that character would be of colour five and the background colour would be six. (Two colour jammed into the raster.)

COMPLEMENT Each pixel you draw will be drawn with the binary complement colour. Where you write 1's the corresponding bit in the Raster will be reversed. The advantage with this drawing mode is that if you write twice on the same spot the old picture will be restored.

INVERSVID This mode is only used for text and is either combined with JAM1 or JAM2.
INVERSVID|JAM1 Only the background is drawn with the FgPen.
INVERSVID|JAM2 The background is drawn with the FgPen, and the characters with the BgPen.

AreaPtSz: The height of a pattern (see below) must be a power of two, and that value (2^x) is stored here. If the AreaPtSz is set to 4, the pattern is 2^4 = 16 lines tall. (If you use multicoloured patterns the AreaPtSz value should be negative. Set the AreaPtSz to -3 if you will use an eight lines tall multicoloured pattern. 2^3 = 8)

linpatcnt: Used to create line pattern.

dummy: A trash are were dummy values are stored.

Flags: If the AREAOUTLINE flag is set all "AreaFill routines" will draw a thin line around all areas that are filled with help of the AOlPen.

If the RastPort is double-buffered the DBUFFER flag is set.

LinePtrn: 16 bits used for the line pattern. A value of 0x9669 (hexadecimal) will create a pattern like this: 1001011001101001.

cp_x: Current pen's X position.

cp_y: Current pen's Y position.

minterms: Extra memory space.

PenWidth: The width of the pen (not used by the drawing

routines).

PenHeight: The height of the pen (not used by the drawing routines).

Font: Pointer to a TextFont structure.

AlgoStyle: The style that was algorithmically generated. (If the font does not support for example Underline, Bold or Italic the Amiga can generate them itself with help of some algorythms.

TxFlags: Special text flags.

TxHeight: The height of the characters.

TxWidth: The nominal width of the characters.

TxBaseline: The position of the text's baseline.

TxSpacing: Spacing between each character.

RP_User: Special variables and memory areas that is of no use for us. Used only by the Amiga and in future versions of the Graphics system.

longreserved: - " -

wordreserved: - " -

reserved: - " -

NOTE! You very rarely need to change or examine these values directly. You normally use (and should use) functions which are described below in order to change any values.

12.3.1.2 PREPARE A RASTPORT

To prepare a RastPort you need to declare a RastPort structure and initialize it by calling the InitRastPort() function with a pointer to the RastPort as only parameter.

Synopsis: InitRastPort(rast_port);

rast_port: (RastPort *) Pointer to the RastPort that should be Initialized.

The last thing you have to do is to give your RastPort a pointer to your BitMap structure. Here is a short example:

```
/* 1. Declare a RastPort structure: */
```

```
struct RastPort my_rast_port;

/* 2. Initialize the RastPort with help of the */
/* InitRastPort() function:                    */
InitRastPort( &my_rast_port );

/* 3. Give the RastPort a pointer to your BitMap structure: */
/* [We assume that the BitMap structure have been correctly */
/* declared and initialized as described above.]            */
my_rast_port.BitMap = &my_bit_map;
```

12.3.2 DRAWING PENS

The low level graphics drawing routines use three different types of drawing pens:

FgPen: The foreground pen is the most commonly used pen.

BgPen: The background pen is normally used to draw the background with, for example if you are writing text.

AOlPen: The Areaoutline pen is used to outline areas that are filled with any of the AreaFill routines.

NOTE! The drawing mode will determine which pens will used.

You use the SetAPen() function to change the FgPen's colour:

Synopsis: SetAPen(rast_port, new_colour);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

new_colour: (long) A new colour value.

You use the SetBPen() function to change the BgPen's colour:

Synopsis: SetBPen(rast_port, new_colour);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

new_colour: (long) A new colour value.

You use the SetOPen() macro to change the AOlPen's colour:
Note! This is not a function. It is actually a macro that is defined in the header file "gfxmacros.h". If you want to use this function you have to remember to include this file.

Synopsis: SetOPen(rast_port, new_colour);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

new_colour: (long) A new colour value.

12.3.3 DRAWING MODES

Five different drawingmodes are supported by the Amiga:

JAM1 The FgPen will be used, the background unchanged. (One colour jammed into a Raster.)

JAM2 The FgPen will be used as foreground pen while the background (when you are writing text for example) will be filled with the BgPen's colour. (Two colours are jammed into a Raster.)

COMPLEMENT Each pixel affected will be drawn with the binary complement colour. Where you write 1's the corresponding bit in the Raster will be reversed.

INVERSVID|JAM1 This mode is only use together with text. Only the background of the text will be drawn with the FgPen.

INVERSVID|JAM2 This mode is only use together with text. The background of the text will be drawn with the FgPen, and the characters itself with the BgPen.

To set the drawing mode you use the SetDrMd() function:

Synopsis: SetDrMd(rast_port, new_mode);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

new_mode: (long) The new drawing mode. Set one of the following: JAM1, JAM2, COMPLEMENT, INVERSVID|JAM1 or INVERSVID|JAM2.

12.3.4 PATTERNS

Amiga's drawing routines allow you to use patterns. You may use both line pattern as well as area patterns which may be two or multicoloured. Line patterns are perfect when you are drawing diagrams, borders etc. Area pattern can be used to fill any type of object on the screen with help of the filling routines. Many colours and different patterns give you great flexibility

to create interesting and good looking displays.

12.3.4.1 LINE PATTERNS

Line patterns are 16 bits wide and each bit represents one dot. The line pattern is initially set to 0xFFFF (1111111111111111) which generates solid lines. To create a pattern of two dots then two spaces then two dots again and so on you would set the line pattern to 0xCCCC (1100110011001100).

You change the line pattern value with help of the SetDrPt() macro. Note! This is not a function. It is actually a macro that is defined in the header file "gfxmacros.h". If you want to use this function you have to remember to include this file.

Synopsis: SetDrPt(rast_port, line_pattern);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

line_pattern: (UWORD) The pattern. Each bit represents one dot. To generate solid lines you set the pattern value to 0xFFFF [hex] (1111111111111111 [bin]).

Here are some commonly used pattern and the corresponding pattern values:

Description	Pattern [bin]	Pattern value [hex]
solid line	1111111111111111	0xFFFF
large dotty line	1111000011110000	0xF0F0
medium dotty line	1100110011001100	0xCCCC
small dotty line	1000100010001000	0x8888
grey line	1010101010101010	0xAAAA

12.3.4.2 AREA PATTERNS

Area patterns are 16 bits wide and a power of two (1, 2, 4, 8, 16, 32, ...) lines tall. To generate an area pattern you must therefore declare and initialize an array of UWORDS. Each bit in the array represents one dot. To create a pattern with a lot of small squares you would declare and initialize the array something like this:

```
UWORD area_pattern =
{
  0xF0F0, /* 1111000011110000 */
  0xF0F0, /* 1111000011110000 */
  0xF0F0, /* 1111000011110000 */
```

```
  0xF0F0, /* 1111000011110000 */
  0x0F0F, /* 0000111100001111 */
  0x0F0F, /* 0000111100001111 */
  0x0F0F, /* 0000111100001111 */
  0x0F0F, /* 0000111100001111 */
};
```

To set the pattern you call the SetAfPt() macro with a pointer to your RastPort as well as you new pattern as parameters. Note! This is not a function. It is actually a macro that is defined in the header file "gfxmacros.h". If you want to use this function you have to remember to include this file.

Synopsis: SetAfPt(rast_port, area_pattern, pow2);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

area_pattern: (UWORD) Pointer to an array of UWORDS that generate the pattern. Each bit in the array represents one dot.

pow2: (BYTE) The pattern must be two to the power of pow2 lines tall. If the pattern is one line tall pow2 should be set to 0, if the pattern is two lines tall pow2 should be set to 1, if the pattern is four lines tall pow2 should be set to 2, and so on. (If you use multicoloured patterns the pow2 should be negative. A sixteen lines tall multicoloured pattern should therefore have the pow2 value set to -4 [2^4 = 16].)

12.3.4.3 MULTICOLOURED PATTERNS

You may even produce multicoloured patterns. You then need to declare an array with a depth of two or more. The larger depth the array have the more colours can be used. Each level represents one BitPlane.

Here is an area pattern array that generates eight different coloured boxes. To get eight colours you need an array with a depth of three. The pattern will look like this: (The numbers are the colour registers that will be used.)

```
  0123
  0123
  4567
  4567

UWORD area_pattern =
{
  {         /* Plane 0          */
    0x0F0F, /* 0000111100001111 */
```

```
    0x0F0F, /* 0000111100001111 */
    0x0F0F, /* 0000111100001111 */
    0x0F0F  /* 0000111100001111 */
  },
  {         /* Plane 1          */
    0x00FF, /* 0000000011111111 */
    0x00FF, /* 0000000011111111 */
    0x00FF, /* 0000000011111111 */
    0x00FF  /* 0000000011111111 */
  },
  {         /* Plane 2          */
    0x0000, /* 0000000000000000 */
    0x0000, /* 0000000000000000 */
    0xFFFF, /* 1111111111111111 */
    0xFFFF  /* 1111111111111111 */
  }
};
```

When you use multicoloured patterns the pow2 value in the SetAfPt() macro should be negative. Simply put a minus in front of the value and the drawing routines will understand that you use multicoloured patterns.

When using multicoloured patterns the drawing mode should be set to JAM2, FgPen to colour 255 and BgPen to colour 0:

```
/* Prepare to draw with multicoloured pattern: */
SetDrMd( &my_rast_port, JAM2 );
SetAPen( &my_rast_port, 255 );
SetBPen( &my_rast_port,   0 );

/* Set pattern: (4 lines tall -> pow2 = 2 [2^2], multicol. -2)
SetAfPt( &my_rast_port, area_pattern, -2 );
```

12.3.5 BITPLANE MASK

To create special effects you can turn off BitPlanes in your Raster so they will not be affected by the drawing routines. The first bit in the mask represents the first BitPlane, the second bit the second BitPlane and so on. A 1 means that the BitPlane may be affected, while a 0 means that the BitPlane will not be affected. To turn off BitPlane zero and two the mask value should be set to 0xFA [hex] = 11111010 [bin].

The mask variable "Mask" can be found in the RastPort structure: my_rast_port.Mask = 0xFF (All BitPlanes may be altered.)

12.3.6 DRAW SINGLE PIXELS

You draw single pixels with help of the WritePixel() function.

Synopsis: WritePixel(rast_port, x, y);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

x: (long) X position of the pixel.

y: (long) Y position of the pixel.

12.3.7 READ SINGLE PIXELS

You use the ReadPixel() function in order to determine what colour a specific pixel has. Just give it a pointer to your RastPort and the coordinates of the pixel and the function will return what colour value that pixel was made of.

Synopsis: colour = ReadPixel(rast_port, x, y);

colour: (long) ReadPixel returns the colour value of the specified pixel (colour 0 - 255) or -1 if the coordinates were outside the Raster.

rast_port: (struct RastPort *) Pointer to the RastPort which contain the pixel you want to examine.

x: (long) X position of the pixel.

y: (long) Y position of the pixel.

12.3.8 POSITION THE CURSOR

Before you can start drawing lines or writing text you need to position the cursor (the position from where the line/text will be drawn from). The cursor is nothing the user can see, it is just the current coordinates for the drawing pen. The cursor position can be found in the RastPort's cp_x and cp_y variables.

The cursor position is altered by calling the Move() function.

Synopsis: Move(rast_port, x, y);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

x: (long) The new X position.

y: (long) The new Y position.

12.3.9 TEXT

In this chapter we will not go into deep details on how to write characters/text. All information about changing style and fonts will be described in a special TEXT chapter. However, I will at least describe how to print normal plain characters on the screen.

To print text you use the Text() function. It needs a pointer to the RastPort that should be affected, a pointer to a text string and finally a value that tells Text() how many characters of the string should be printed. Only one line each call may be printed, no formatting, word-wrapping etc is done.

Synopsis: Text(rast_port, string, nr_of_chr);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

string: (char *) Pointer to a text string that will be printed.

nr_of_chr: (long) The number of characters that should be printed.

This example prints "Hello" on the display:

```
Text( &my_rast_port, "HELLO", 5 );
```

12.3.10 DRAW SINGLE LINES

You draw single lines with help of the Draw() function. It will draw a line from the current position to a new position anywhere on the Raster.

Synopsis: Draw(rast_port, x, y);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

x: (long) The new X position.

y: (long) The new Y position.

Here is an example that will draw a triangle on the screen:

```
Move( &my_rast_port, 100, 100 ); /* Move to start position. */
Draw( &my_rast_port, 300, 100 ); /* Draw the three lines.   */
Draw( &my_rast_port, 200, 200 );
Draw( &my_rast_port, 100, 100 );
```

12.3.11 DRAW MULTIPLE LINES

To draw multiple lines we use the PolyDraw() function. It needs a pointer to an array of coordinates, a value that tells PolyDraw() how many pairs of coordinates (x,y) will be used and of course a pointer to the RastPort that should be affected. A line is drawn from the current position to the first coordinate. There a second line is drawn to the second coordinate and so on.

Synopsis: PolyDraw(rast_port, number, coordinates);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

number: (long) The number of coordinates (x,y) defined in the array.

coordinates: (short *) Pointer to an array of coordinates.

Here is an example that also will draw a triangle on the screen:

```
/* Three coordinates: */
WORD coordinates[] =
{
  300, 100,
  200, 200,
  100, 100
};

/* Move to the start position: */
Move( &my_rast_port, 100, 100 );

/* Draw some lines: */
PolyDraw( &my_rast_port, 3, coordinates );
```

12.3.12 DRAW FILLED RECTANGLES

To draw filled rectangles you use the RectFill function. (Note that the area pattern will have effect here.)

Synopsis: RectFill(rast_port, minx, miny, maxx, maxy);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

minx: (long) Left position of the rectangle.

miny: (long) Top - " -

maxx: (long) Right - " -

maxy: (long) Bottom - " -

12.3.13 FLOOD FILL

Flood fill is used when a complicated object should be filled. There exist two different types of flood fill, Outline mode and Colour mode. In outline mode everything will be filled with the new colour, and the flood is only stopped when it has reached an edge of the same colour as AOlPen. In colour mode all pixels with the same colour and are side by side with each other will be affected. The fill will stop first when the flood has reached a different colour.

You flood fill objects with help of the Flood() function.

Synopsis: Flood(rast_port, mode, x, y);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

mode: (long) Which mode should be used. If you want to use the Colour mode set the mode variable to 1, to get the Outline mode set the mode variable to 0.

x: (long) X position where the flood fill should start.

y: (long) Y position where the flood fill should start.

Here is an example that will draw a triangle and then fill it. We set the AOlPen to the same colour as the FgPen so the flood fill will stop when it reaches the edges, outline mode.

```
/* Set the AOlPen to the same colour as FgPen: */
SetOPen( &my_rast_port, my_rast_port.FgPen );

/* Draw a triangle: */
Move( &my_rast_port, 100, 100 );
Draw( &my_rast_port, 300, 100 );
Draw( &my_rast_port, 200, 200 );
Draw( &my_rast_port, 100, 100 );

/* Fill it: (Use the Outline mode) */
Flood( &my_rast_port, 0, 200, 150 );
```

12.3.14 DRAW FILLED AREAS

The Amiga allows you to draw areas that are directly filled. You move to a location and then starts to draw your objects. The user will however not notice anything since no lines are drawn. The lines are instead stored in a large vector list, and will first be drawn and filled when you finish (close) the object.

12.3.14.1 AREAINFO AND TMPRAS STRUCTURES

To manage this you need to do some things:

1. A buffer in which a list of all your vertices can be stored in. Five bytes per vertex is needed, and to create a buffer for 100 lines you need 500 bytes. However, the buffer must be word aligned, and you should therefore declare the buffer as a collection of words instead of bytes. Since there goes two bytes on every word, 500 bytes means 250 words.
   Example: UWORD my_buffer[250];

2. Declare an AreaInfo structure. (Declared in file "rastport.h".)
   Example: struct AreaInfo my_area_info;

3. Combine the buffer and the AreaInfo structure with help of the InitArea() function. It needs a pointer to an AreaInfo structure, a pointer to a memory buffer and a value that tells it how many vertices may be stored in the buffer.
   Example: InitArea(&my_area_info, my_buffer, 100);

4. Declare a TmpRas structure. (Also declared in the file "rastport.h".)
   Example: struct TmpRas my_tmp_ras;

5. Reserve a BitPlane large enough to store all your objects in. Normally you make the BitPlane equal large as the display. Example:
   ```
   PLANEPTR my_bit_plane;
   my_bit_plane = AllocRaster( 320, 200 );
   if( my_bit_plane == NULL )
     exit();
   ```

6. Initialize and combine the TmpRas structure and the extra memory by calling the InitTmpRas() function. It needs a pointer to the TmpRas structure, a pointer to some extra display memory, and finally a value telling it how large the extra memory area is, calculated with help of the RASSIZE() macro. Example:
   ```
   InitTmpRas( &my_tmp_ras, my_bit_plane, RASSIZE( 320, 200 ) );
   ```

7. Tell the RastPort were it can find the AreaInfo and TmpRas structures. Example:

```
   my_rast_port.AreaInfo = &my_area_info;
   my_rast_port.TmpRas = &my_tmp_ras;
```

8. Before your program terminates it must deallocate the display memory it has allocated. Use the function FreeRaster(). Example:
   FreeRaster(my_bit_plane, 320, 200);

12.3.14.2 AREAMOVE(), AREADRAW() AND AREAEND()

Once the TmpRas and AreaInfo structures have been initialized, and the RastPort know where to find them, you may start to use the AreaFill functions. The first thing you probably would do is to move to a new position were you want to start drawing. You do it by calling the AreaMove() function.

Synopsis: AreaMove(rast_port, x, y);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

x: (long) Start X position.

y: (long) Start Y position.

You can now start to draw with help of the AreaDraw() function. NOTE! The user will not see anything, it is first when you close this polygon it will be drawn and filled.

Synopsis: AreaDraw(rast_port, x, y);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

x: (long) New X position.

y: (long) New Y position.

Be careful not to draw outside the Raster, it may crash the system!

Once you are ready with your object you call the function AreaEnd() and the polygon is closed and filled. You do not need to close the polygon itself, the last line will if necessary been drawn automatically. (If you call AreaMove() before you have closed a polygon, it will be closed automatically before the new polygon starts.)

Synopsis: AreaEnd(rast_port);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

12.3.14.3 TURN OFF THE OUTLINE FUNCTION

The areas that are filled with help of the AreaFill functions will be outlined with the AOlPen. You can however turn this outline function off if you do not want it. You simply call the macro (declared in file "gfxmacros.h") BNDROFF() with a pointer to your RastPort as only parameter.

Synopsis: BNDROFF(rast_port);

rast_port: Pointer to the RastPort which outlinefunction should be turned off.

12.3.14.4 EXAMPLE

Here is an example:

```
#define WIDTH  320
#define HEIGHT 200

/* Declare an AreaInfo and TmpRas structure: */
struct AreaInfo my_area_info;
struct TmpRas my_tmp_ras;

/* Vertex buffer: (100 lines, 5 bytes each gives 500 bytes */
/* which is 250 words. Must be word-aligned.)               */
UWORD my_buffer[ 250 ];

/* Pointer to some display memory that will be allocated: */
PLANEPTR my_bit_plane;

/* Initialize the AreaInfo structure. */
/* Prepare for 100 vertices:          */
InitArea( &my_area_info, my_buffer, 100);

/* Allocate some display memory: */
my_bit_plane = AllocRaster( WIDTH, HEIGHT );

/* Have we allocated it successfully? */
if( my_bit_plane == NULL )
  exit();

/* Prepare the TmpRas structure: */
InitTmpRas( &my_tmp_ras, my_bit_plane, RASSIZE( WIDTH, HEIGHT ) );

/* Tell RastPort were it can find the structures: */
my_rast_port.AreaInfo = &my_area_info;
my_rast_port.TmpRas = &my_tmp_ras;
```

```
/* Draw a filled square: */
AreaMove( my_rast_port,   0,   0 );
AreaDraw( my_rast_port, 100,   0 );
AreaDraw( my_rast_port, 100, 100 );
AreaDraw( my_rast_port,   0, 100 );
AreaEnd( my_rast_port );


/* Deallocate the extra memory: */
FreeRaster( my_bit_plane, WIDTH, HEIGHT );
```

12.3.15 SET THE RASTER TO A SPECIFIC COLOUR

You set a whole Raster to a specific colour with help of the SetRast() function.

Synopsis: SetRast(rast_port, colour);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

colour: (long) The colour reg. you want to fill the whole raster with.

12.3.16 BLITTER

The Blitter is a chip inside the Amiga which is specialized in moving and logically combining large rectangular memory areas. It moves and works with enormous amount of data and can therefore do a lot of hard work which would take much more time for the main processor to do. The main processor can instead concentrate itself on more important tasks.

The Blitter, also called Fat Agnus (the chip is very fat), is the hero behind the Amigas fashinating graphics capabilities. It can be used to clear, scroll and copy large memory areas. Unlike other blitters it can also logically combine memory areas (explained later) and can therefore do a lot more interesting things than just moving data.

12.3.16.1 CLEAR RECTANGULAR MEMORY AREAS

You can clear memory with help of the BltClear() function. The advantage of using this function is that it works together with the blitter which is the fastest chip to clear memory.

Synopsis: BltClear(memory, count, flags);

memory: (char *) Pointer to the memory that should be cleared.

count: (long) Number of bytes that should be cleared. If you clear display memory you can use the macro RASSIZE() to calculate how many bytes the display use.

flags: (long) Write 0 to allow the computer to continue while the blitter is still working, or write 1 which will force the program to wait for the blitter.

12.3.16.2 SCROLL A RECTANGULAR AREA

You can scroll an area of a raster with help of the ScrollRaster() function. It works together with the blitter and is therefore very fast. The area can be moved both vertically as well as horizontally. The area which is moved out of the raster is lost, and the new area is filled with the BgPen.

Synopsis: ScrollRaster(rp, dx, dy, minx, miny, maxx, maxy);

rp: (struct RastPort *) Pointer to the RastPort that should be affected.

dx: (long) Delta X movement. (A positive number moves the area to the right, a negative number to the left.)

dy: (long) Delta Y movement. (A positive number moves the area down, a negative number up.)

minx: (long) Left edge of the rectangle.

miny: (long) Top edge of the rectangle.

maxx: (long) Right edge of the rectangle.

maxy: (long) Bottom edge of the rectangle.

12.3.16.3 COPY RECTANGULAR AREAS

When you are going to copy rectangular memory areas you can either use BltBitMap() or ClipBlit(). BltBitMap() should be used when you simply want to copy data and do not bother about overlapping layers (windows) etc. The ClipBlit() on the other

hand should be used when you want to look out for overlapping layers etc.

BltBitMap() copies BitMaps directly without worrying about overlapping layers.

Synopsis: BltBitMap(sbm, sx, sy, dbm, dx, dy, w, h, flag, m, t);

sbm: (struct BitMap *) Pointer to the "source" BitMap.

sx: (long) X offset, source.

sy: (long) Y offset, source.

dbm: (struct BitMap *) Pointer to the "destination" BitMap.

dx: (long) X offset, destination.

dy: (long) Y offset, destination.

w: (long) The width of the memory area that should be copied.

h: (long) The height of the memory area that should be copied.

flag: (long) This value tells the blitter what kind of logically operations should be done. See below for more information.

m: (long) You can here define a BitMap mask, and tell the blitter which BitPlanes should be used, and which should not. The first bit represents the first BitPlane, the second bit the second BitPlane and so on. If the bit is on (1) the corresponding BitPlane will be used, else (0) the BitPlane will not be used. To turn off BitPlane zero and two, set the mask value to 0xFA (11111010). To use all BitPlanes set the mask value to 0xFF (11111111).

t: (char *) If the copy overlaps and this pointer points to some chip-memory, the memory will be used to store the temporary area in. However, normally you do not need to bother about this value.

ClipBlit() copies BitMaps with help of Rastports and will therefore care about overlapping layers, and should be used if you have windows on your display.

Synopsis: ClipBlit(srp, sx, sy, drp, dx, dy, w, h, flag);

srp: (struct RastPort *) Pointer to the "source" RastPort.

sx: (long) X offset, source.

sy: (long) Y offset, source.

drp: (struct RastPort *) Pointer to the "destination" RastPort.

dx: (long) X offset, destination.

dy: (long) Y offset, destination.

w: (long) The width of the memory area that should be copied.

h: (long) The height of the memory area that should be copied.

flag: (long) This value tells the blitter what kind of logically operations should be done. See below for more information.

As said above the blitter can logically combine data from the source and destination areas. The first four bits to the left in the flag field determines what calculations should be done.

If the leftmost bit is set (value 0x80) the result will be a combination of the source and destination. If the second leftmost bit is set (value 0x40) the result will be a combination of the source and inverted destination. If the third leftmost bit is set (value 0x20) the result will be a combination of the inverted source and destination. If the fourth leftmost bit is set (value 0x10) the result will be the source together with destination and all inverted. See table below:

Bit	Value	Log
1000 0000	0x80	SD
0100 0000	0x40	$S\overline{D}$
0010 0000	0x20	$\overline{S}D$
0001 0000	0x10	$\overline{SD}$

To make a normal copy the result should be only S. We can get it by setting the flag value to 0xC0 [hex] (11000000 [bin]). The mathematically formula will then be:

$$SD + S\overline{D} = S(D + \overline{D}) = S$$

If you want to get the source inverted ($\overline{S}$), set the flag value to 0x30 [hex] (00110000 [bin]). The mathematically formula will then be:

$$\overline{S}D + \overline{SD} = \overline{S}(D + \overline{D}) = \overline{S}$$

To get the destination inverted (D), set the flag value to 0x50 [hex] (01010000 [bin]). The mathematically formula will then be:

$$\overline{S}D + \overline{SD} = \overline{D}(S + \overline{S}) = \overline{D}$$

Here is an example that will copy a rectangular area from a RastPort called source_rp to another RastPort called destination_rp. The rectangular area is WIDTH pixels wide and HEIGHT lines tall.

```
ClipBlit( &source_rp,      /* Source RastPort.              */
          0, 0,            /* Top left corner, source.      */
          &destination_rp, /* Destination RastPort.         */
          0, 0,            /* Top left corner, destination. */
          WIDTH, HEIGHT,   /* Size of the memory area.      */
          0xC0 );          /* Normal copy.                  */
```

12.4 FUNCTIONS

InitView()

This function will initialize a View structure.

Synopsis: InitView(view);

view: (struct View *) Pointer to the View that should be initialized.

InitVPort()

This function will initialize a ViewPort structure.

Synopsis: InitVPort(view_port);

view_port: (struct ViewPort *) Pointer to the ViewPort that should be initialized.

GetColorMap()

This function allocates and initializes a ColorMap structure.

Synopsis: colormap = GetColorMap(colours);

colormap: (struct ColorMap *) GetColorMap returns a pointer to the ColorMap structure it has allocated and initialized, or NULL if not enough memory.

colours: (long) A value specifying how many colours you want that the ColorMap structure should store.

(1, 2, 4, 8, 16, 32)

FreeColorMap()

This function deallocates the memory that was allocated by the GetColorMap() function. Remember to deallocate all memory that you allocate. For every GetColorMap() function there should be one FreeColorMap() function.

FreeColorMap(colormap);

colormap: (struct ColorMap *) Pointer to a ColorMap structure that GetColorMap() returned and you now want to deallocate.

InitBitMap()

This function initializes a BitMap structure.

Synopsis: InitBitMap(bitmap, depth, width, height);

bitmap: (struct BitMap *) Pointer to the BitMap.

depth: (long) How many BitPlanes used.

width: (long) The width of the raster.

height: (long) The height of the raster.

AllocRaster()

This function reserves display memory (one BitPlane).

Synopsis: pointer = AllocRaster(width, height);

pointer (PLANEPTR) Pointer to the allocated memory or NULL if enough memory could not be reserved.

width: (long) The width of the BitMap.

height: (long) The height of the BitMap.

BltClear()

This function clears large rectangular memory areas. This function work together with the blitter and is therefore very fast.

Synopsis: BltClear(pointer, bytes, flags);

pointer (char *) Pointer to the memory.

bytes: (long) The lower 16 bits tells the blitter how many

bytes per row, and the upper 16 bits how many rows. This value is automatically calculated for you with help of the macro RASSIZE(). Just give RASSIZE() the correct width and height and it will return the correct value. [RASSIZE() is defined in file "gfx.h".]

flags: (long) Set bit 0 to force the function to wait until the Blitter has finished with your request.

MakeVPort()

This function prepares the Amiga's hardware (especially the Copper) to display a ViewPort. NOTE! You have to prepare EVERY ViewPort you are going to use!

Synopsis: MakeVPort(view, viewport);

view: (struct View *) Pointer to the ViewPort's View.

viewport: (struct ViewPort *) Pointer to the ViewPort.

MrgCop()

This function puts together all displayinstructions and prepares the view to be showed.

Synopsis: MrgCop(view);

view: (struct View *) Pointer to the View.

LoadView()

This function will start showing a View. Remember that when you close your View you must switch back to the old view. (See examples for more details.)

Synopsis: LoadView(view);

view: (struct View *) Pointer to the View.

FreeVPortCopLists()

This function will return all memory that was automatically allocated by the MakeVPort() function. Remember to call FreeVPortCopLists() for every ViewPort you have created!

Synopsis: FreeVPortCopLists(viewport);

view: (struct ViewPort *) Pointer to the ViewPort.

FreeCprList()

This function will return all memory that was automatically allocated by the MrgCop() function.

Synopsis: FreeCprList(cprlist);

cprlist: (struct cprlist *) Pointer to the View's cprlist (LOFCprList) structure. If the View was interlaced you must also call the FreeCprList function with a pointer to the SHFCprList.

FreeRaster()

This function will deallocate display memory (BitPlane). Remember to deallocate all BitPlanes!

Synopsis: FreeRaster(bitplane, width, height);

bitplane: (PLANEPTR) Pointer to a Bitplane.

width: (long) The Bitplane's width.

height: (long) The Bitplane's height.

FreeColorMap()

This function deallocates the memory that was allocated by the GetColorMap() function. Remember to deallocate all memory that you allocate. For every GetColorMap() function there should be one FreeColorMap() function.

FreeColorMap(colormap);

colormap: (struct ColorMap *) Pointer to a ColorMap structure that GetColorMap() returned and you now want to deallocate.

InitRastPort()

This function initializes a RastPort.

Synopsis: InitRastPort(rast_port);

rast_port: (RastPort *) Pointer to the RastPort that should be Initialized.

SetAPen()

This function will change the FgPen's colour.

Synopsis: SetAPen(rast_port, new_colour);

rast_port: (struct RastPort *) Pointer to the RastPort that

should be affected.

new_colour: (long) A new colour value.

SetBPen()

This function will change the BgPen's colour.

Synopsis: SetBPen(rast_port, new_colour);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

new_colour: (long) A new colour value.

SetOPen()

This macro will change the AOlPen's colour. Note! This is not a function. It is actually a macro that is defined in the header file "gfxmacros.h". If you want to use this function you have to remember to include this file.

Synopsis: SetOPen(rast_port, new_colour);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

new_colour: (long) A new colour value.

SetDrMd()

This function will change the drawing mode.

Synopsis: SetDrMd(rast_port, new_mode);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

new_mode: (long) The new drawing mode. Set one of the following: JAM1, JAM2, COMPLEMENT, INVERSVID|JAM1 or INVERSVID|JAM2.

JAM1	The FgPen will be used, the background unchanged. (One colour jammed into a Raster.)
JAM2	The FgPen will be used as foreground pen while the background (when you are writing text for example) will be filled with the BgPen's colour. (Two colours are jammed into a Raster.)
COMPLEMENT	Each pixel affected will be drawn

with the binary complement colour. Where you write 1's the corresponding bit in the Raster will be reversed.

INVERSVID|JAM1 This mode is only use together with text. Only the background of the text will be drawn with the FgPen.

INVERSVID|JAM2 This mode is only use together with text. The background of the text will be drawn with the FgPen, and the characters itself with the BgPen.

SetDrPt()

This function will set the line pattern.

Synopsis: SetDrPt(rast_port, line_pattern);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

line_pattern: (UWORD) The pattern. Each bit represents one dot. To generate solid lines you set the pattern value to 0xFFFF [hex] (1111111111111111 [bin]).

SetAfPt()

This function will set the area pattern:

Synopsis: SetAfPt(rast_port, area_pattern, pow2);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

area_pattern: (UWORD) Pointer to an array of UWORDS that generate the pattern. Each bit in the array represents one dot.

pow2: (BYTE) The pattern must be two to the power of pow2 lines tall. If the pattern is one line tall pow2 should be set to 0, if the pattern is two lines tall pow2 should be set to 1, if the pattern is four lines tall pow2 should be set to 2, and so on. (If you use multicoloured patterns the pow2 should be negative. A sixteen lines tall multicoloured pattern should therefore have the pow2 value set to -4 [2^4 = 16].)

WritePixel()

This function will draw a single pixel.

Synopsis: WritePixel(rast_port, x, y);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

x: (long) X position of the pixel.

y: (long) Y position of the pixel.

ReadPixel()

This function reads the colour value of a pixel.

Synopsis: colour = ReadPixel(rast_port, x, y);

colour: (long) ReadPixel returns the colour value of the specified pixel (colour 0 - 255) or -1 if the coordinates were outside the Raster.

rast_port: (struct RastPort *) Pointer to the RastPort which contain the pixel you want to examine.

x: (long) X position of the pixel.

y: (long) Y position of the pixel.

Move()

This function moves the cursor.

Synopsis: Move(rast_port, x, y);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

x: (long) The new X position.

y: (long) The new Y position.

Text()

This function prints text into a Raster.

Synopsis: Text(rast_port, string, nr_of_chr);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

string: (char *) Pointer to a text string that will be printed.

nr_of_chr: (long) The number of characters that should be

printed.

Draw()

This function draws single lines from the current position to the new specified position.

Synopsis: Draw(rast_port, x, y);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

x: (long) The new X position.

y: (long) The new Y position.

PolyDraw()

This function will draw multiple lines.

Synopsis: PolyDraw(rast_port, number, coordinates);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

number: (long) The number of coordinates (x,y) defined in the array.

coordinates: (short *) Pointer to an array of coordinates.

RectFill()

This function will draw filled rectangles.

Synopsis: RectFill(rast_port, minx, miny, maxx, maxy);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

minx: (long) Left position of the rectangle.

miny: (long) Top - " -

maxx: (long) Right - " -

maxy: (long) Bottom - " -

Flood()

This function will flood fill complicated objects.

Synopsis: Flood(rast_port, mode, x, y);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

mode: (long) Which mode should be used. If you want to use the Colour mode set the mode variable to 1, to get the Outline mode set the mode variable to 0.

x: (long) X position where the flood fill should start.

y: (long) Y position where the flood fill should start.

AreaMove()

This function will start a new polygon.

Synopsis: AreaMove(rast_port, x, y);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

x: (long) Start X position.

y: (long) Start Y position.

AreaDraw()

This function will add a new vertex to the vector list.

Synopsis: AreaDraw(rast_port, x, y);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

x: (long) New X position.

y: (long) New Y position.

AreaEnd()

This function will close, draw and fill the polygon.

Synopsis: AreaEnd(rast_port);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

BNDROFF()

This macro (declared in file "gfxmacro.h") will turn off the outline mode.

Synopsis: BNDROFF(rast_port);

rast_port: Pointer to the RastPort which outlinefunction should be turned off.

SetRast()

This function sets a whole Raster to a specific colour.

Synopsis: SetRast(rast_port, colour);

rast_port: (struct RastPort *) Pointer to the RastPort that should be affected.

colour: (long) The colour reg. you want to fill the whole raster with.

ScrollRaster()

This function will scroll a rectangular area of a raster.

Synopsis: ScrollRaster(rp, dx, dy, minx, miny, maxx, maxy);

rp: (struct RastPort *) Pointer to the RastPort that should be affected.

dx: (long) Delta X movement. (A positive number moves the area to the right, a negative number to the left.)

dy: (long) Delta Y movement. (A positive number moves the area down, a negative number up.)

minx: (long) Left edge of the rectangle.

miny: (long) Top edge of the rectangle.

maxx: (long) Right edge of the rectangle.

maxy: (long) Bottom edge of the rectangle.

BltBitMap()

This function copies parts of BitMaps directly without worrying about overlapping layers.

Synopsis: BltBitMap(sb, sx, sy, db, dx, dy, w, h, fl, m, t);

sb: (struct BitMap *) Pointer to the "source" BitMap.

sx: (long) X offset, source.

sy: (long) Y offset, source.

db: (struct BitMap *) Pointer to the "destination" BitMap.

dx: (long) X offset, destination.

dy: (long) Y offset, destination.

w: (long) The width of the memory area that should be copied.

h: (long) The height of the memory area that should be copied.

fl: (long) The four leftmost bits tells the blitter what kind of logically operations should be done.

m: (long) You can here define a BitMap mask, and tell the blitter which BitPlanes should be used, and which should not. The first bit represents the first BitPlane, the second bit the second BitPlane and so on. If the bit is on (1) the corresponding BitPlane will be used, else (0) the BitPlane will not be used. To turn off BitPlane zero and two, set the mask value to 0xFA (11111010). To use all BitPlanes set the mask value to 0xFF (11111111).

t: (char *) If the copy overlaps and this pointer points to some chip-memory, the memory will be used to store the temporary area in. However, normally you do not need to bother about this value.

ClipBlit()

This function copies parts of BitMaps with help of Rastports and will therefore care about overlapping layers, and should be used if you have windows on your display.

Synopsis: ClipBlit(srp, sx, sy, drp, dx, dy, w, h, flag);

srp: (struct RastPort *) Pointer to the "source" RastPort.

sx: (long) X offset, source.

sy: (long) Y offset, source.

drp: (struct RastPort *) Pointer to the "destination" RastPort.

dx: (long) X offset, destination.

dy: (long) Y offset, destination.

w: (long) The width of the memory area that should be copied.

h: (long) The height of the memory area that should be copied.

flag: (long) This value tells the blitter what kind of logically operations should be done. See below for more information.

12.5 EXAMPLES

The printed version of these examples can be found in Appendix A (Part IV: Appendices).

Example1

This example shows how to create your own display, and fill it with a lot of pixels in seven different colours.

Example2

This example shows how to create a large Raster and a smaller display. We fill the Raster with a lot of pixels in seven different colours and by altering the RxOffset and RyOffset values in the RasInfo structure, the Raster is scrolled in all directions. This method to scroll a large drawing in full speed is used in many games and was even used in my own racing game "Car".

Example3

This example shows how to create a display that covers the entire display. This method is called "Overscan", and is primarily used in video and graphics programs, but can also be used in games etc to make the display more interesting.

Example4

This example demonstrates how to open two different ViewPorts on the same display. The first ViewPort is in low resolution and use 32 colours, while the second ViewPort is in high resolution and only use 2 colours.

Example5

This example demonstrates how to open a ViewPort in interlace mode.

Example6

This example demonstrates how to create a ViewPort in dual playfield mode. Playfield 1 use four colours and is placed behind playfield 2 which only use two colours (transparent and grey). Playfield 1 is filled with a lot of dots and is scrolled around while playfield 2 is is not moved and is filled with only five grey rectangles.

Example7

This example demonstrates how to create a ViewPort with the special display mode "Hold and Modify".

Example8

This example shows how to use the functions: SetAPen(), SetBPen(), SetOPen(), SetDrMd(), SetDrPt(), WritePixel(), ReadPixel(), Move(), Draw(), Text() and finally PolyDraw().

Example9

This example shows how to flood fill a figure, and how to draw filled rectangles (both solid as well as filled with single and multi coloured patterns).

Example10

This example demonstrate how to use the Area Fill functions. [AreaMove(), AreaDraw() and AreaEnd().]

Example11

This example demonstrate how to copy rectangular memory areas with help of the blitter.

13 VSPRITES

13.1 INTRODUCTION

VSprites, or Virtual Sprites as their whole name is, are very similar to Hardware Sprites. VSprites have the same limitations such as a maximum width of 16 pixels and can only use three colours (actually four colours, but the first colour will act as transparent.) However, VSprites are superior to Hardware Sprites because you may use much more than eight VSprites at the same time, and each VSprite may use its own three colours out of a palette of 4096 colours. The disadvantage with VSprites compared to Hardware Sprites is that Hardware Sprites are totally controlled by the hardware and thus very fast, while VSprites are partly controlled by the software and because of that a bit slower.

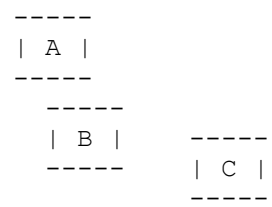
13.2 HOW VSPRITES WORK

The reason why VSprites are so similar to Hardware Sprites is that VSprites are actually displayed with help of Hardware Sprites. If you want a VSprite to be at the top of the display and one further down, the system will position a Hardware Sprite at the top and change its image so it correspond to the VSprites image. The colours will also automatically be changed by altering the Copper List. (See chapter "Hacks" for more information about the Copper list.)

When the video beam has passed the first VSprite, the Hardware Sprite is not needed there any more and can therefore be used again. The system can therefore use the same Hardware Sprite to display another VSprite, and so on. Because of this you can have much more than eight sprites at the same time on the display.

If two or more VSprites would be on the same height, more Hardware Sprites are needed. The system has eight Hardware Sprites as its disposal, and can therefore display up to eight VSprites on the same line.

Here is an example. You want to display three VSprites positioned as below. In this case two Hardware Sprites are needed. Hardware Sprite one is used to display VSprite A and B. A and B are not on the same height, and only one Hardware Sprite is therefore needed. However, VSprite C is on the same lines as B, and a second Hardware Sprite is therefore needed.



13.2.1 LIMITATIONS

There are some important limitations that you need to remember:

1. If you have reserved some Hardware Sprites for your own use, the system can not use these sprites and this will limit the maximum numbers of VSprites on the same height.

2. Hardware Sprites 0 and 1, 2 and 3, 4 and 5, 6 and 7 share the same colours. If sprite 0 is used to display one VSprite, the computer can not use sprite 1 to display another VSprite on the same height if it has different colours. Because of that there is a limitation of a maximum of four VSprites with different colours on the same height.

3. If you display the VSprites on a screen with a depth of 5 or more (32 colours or more) you will see strange colour fluctuations on the lines where the VSprites are. The reason is that VSprites are now using the same colour register as the display, and as we said before, the system is all the time changing these colours to match the VSprites' colours.

If the system can not find a Hardware Sprite to display a VSprite because of the limitations above, the VSprite will simply not be drawn.

13.2.2 HOW TO AVOID THE LIMITATIONS

There are several guidelines to follow in order to avoid the limitations:

1. Use as few VSprites as possible on the same lines.
2. Try to use VSprites with the same colours.
3. If you are using a display with more than 16 colours, you should only use colour 16, 20, 24, and 28. (These are the transparent colours, and will therefore not be changed.)

13.3 CREATE VSPRITES

```
IF YOU WANT TO USE VSPRITES YOU HAVE TO
---------------------------------------------------------------
 1. Declare and initialize some sprite data for each VSprite.
 2. Declare a VSprite structure for each VSprite plus two extra
    structures for two dummy VSprites.
 3. Decide each VSprite's colours.
 4. Declare a GelsInfo structure.
 5. Open the Graphics Library.
 6. Initialize the GelsInfo structure.
 7. Initialize the VSprite structure.
 8. Add the VSprites to the VSprite list.
 9. Sort the Gels list. SortGList()
10. Draw the Gels list. DrawGList()
11. Set the Copper and redraw the display.
    [ MrgCop() and LoadView() or        ]
    [ MakeScreen() and RethinkDisplay() ]
12. Play around with the VSprites.
13. Remove the VSprites. RemVSprite()

(Easy, isn't it?)
```

13.3.1 VSPRITE DATA

Sprite data for VSprites is declared and initialized as normal sprite data. Just remember that all sprite data must be placed in Chip memory.

Example:

```
  UWORD chip vsprite_data[]=
  {
    0x0FF0, 0x0FF0,
    0x0FF0, 0x0FF0,
    0x0FF0, 0x0FF0,
    0x0FF0, 0x0FF0
  };
```

13.3.2 VSPRITE STRUCTURE

You need to declare a VSprite structure for each VSprite you are going to use, and two extra for two "dummy VSprites". The two dummy VSprites are used by the gel system (GEL stands for Graphic ELements). (One of the two dummy sprites is placed first in the gel-list, and the other one is placed last. The gel system can then sort the list much faster, and speed is

essential here.)

Example:

```
  struct VSprite head, tail, vsprite;
```

The VSprite structure looks like this: [Declared in the header file "graphics/gels.h".]

```
struct VSprite
{
  struct VSprite *NextVSprite;
  struct VSprite *PrevVSprite;
  struct VSprite *DrawPath;
  struct VSprite *ClearPath;

  WORD OldY, OldX;
  WORD Flags;
  WORD Y, X;
  WORD Height;
  WORD Width;
  WORD Depth;
  WORD MeMask;
  WORD HitMask;
  WORD *ImageData;
  WORD *BorderLine;
  WORD *CollMask;
  WORD *SprColors;
  struct Bob *VSBob;
  BYTE PlanePick;
  BYTE PlaneOnOff;
  VUserStuff VUserExt;
};
```

NextVSprite: Pointer to the next VSprite in the list.

PrevVSprite: Pointer to the previous VSprite in the list.

DrawPath: Used by BOBs.

ClearPath: Used by BOBs.

OldY: Previous Y position.

OldX: Previous X position.

Flags: Following flags may be set by the user:

SUSERFLAGS	Mask of the user settable flags.
VSPRITE	Set this flag if you are using the structure for a VSprite.
MUSTDRAW	Set this flag if this VSprite must be drawn.
	Following flags are set by the system:
GELGONE	This flag is set if the VSprite is outside the

VSOVERFLOW Too many VSprites on the same lines. If the flag MUSTDRAW is set, the system will try to draw it, but it could look strange.

Y: Y position.

X: X position.

Height: The height of the VSprite.

Width: Number of words used for each line. For the moment a VSprite can only be 16 pixels wide, which means two words.

Depth: Number of planes. A VSprite can for the moment only use two planes, 3 colours and one transparent.

MeMask: What can collide with this VSprite. (Will be explained in a future version of the manual.)

HitMask: What this VSprite can collide with. (Will be explained in a future version of the manual.)

ImageData: Pointer to the sprite data.

BorderLine: (Will be explained in a future version of the manual.)

CollMask: (Will be explained in a future version of the manual.)

SprColors: Pointer to a colour table for this VSprite.

VSBob: Used if it is a BOB. (Will be explained in a future version of the manual.)

PlanePick: Used if it is a BOB. (Will be explained in a future version of the manual.)

PlaneOnOff: Used if it is a BOB. (Will be explained in a future version of the manual.)

VUserExt: You may use this as you please. Here you can add extra information etc. (Is actually declared as a SHORT variable.)

13.3.3 COLOUR TABLE

Each VSprite can have its own three colours out of a palette of 4096. The three desired colours are placed in an array of WORDS.

Example:

```
  WORD colour_table[] = { 0x000F, 0x00F0, 0x0F00 };
```

13.3.4 GELSINFO STRUCTURE

If you want to use VSprites or BOBs you have to declare and initialize a GelsInfo structure.

Example:

```
  struct GelsInfo ginfo;
```

The GelsInfo structure looks like this: [Declared in the header file "graphics/rastport.h".]

```
struct GelsInfo
{
  BYTE sprRsrvd;
  UBYTE Flags;
  struct VSprite *gelHead, gelTail;
  WORD *nextLine;
  WORD **lastColor;
  struct collTable *collHandler;
  short leftmost, rightmost, topmost, bottommost;
  APTR firstBlissObj, lastBlissObj;
};
```

sprRsrvd: Which hardware sprites should be reserved to be used as VSprites. Bit zero represents the first sprite, bit one the second sprite and so on. If the bit is set, the sprite may be used. Set the mask to 0xFF if all sprites should be used as VSprites.

Flags: Used by the system.

gelHead: Pointer to the first dummy VSprite.

gelTail: Pointer to the second dummy VSprite.

nextLine: Which sprites are available on the next line.

lastColor: Pointer to an array of colours which were last used.

collHandler: Pointer to the collision routines.

leftmost: Used by the system.

rightmost: Used by the system.

topmost: Used by the system.

bottommost: Used by the system.

firstBlissObj: Used by the system.

lastBlissObj: Used by the system.

13.3.5 INITIALIZE THE GELSINFO STRUCTURE

After you have declared the GelsInfo structure you need to initialize some important fields. First you need to tell the system which sprites it should be allowed to use as VSprites. If any hardware sprite should be allowed to be used, set the sprRsrvd field to 0xFF. If any sprite except the first two should be allowed to be used, set the mask to 0xFC.

You must also give the nextLine field a pointer to an array of eight WORDS, and the lastColor field a pointer to an array of eight WORD pointers.

Example:

```
  WORD nextline[8];
  WORD *lastcolor[8];

  /* All sprites except the first two may be used as VSprites: */
  ginfo.sprRsrvd = 0xFC;

  ginfo.nextLine = nextline;
  ginfo.lastColor = lastcolor;
```

Finally you give the GelsInfo structure to the system by calling the InitGels() function, and give the Rastport's GelsInfo field a pointer to your GelsInfo structure.

Synopsis: InitGels(head, tail, ginfo);

head: (struct VSprite *) Pointer to the first "dummy" VSprite structure.

tail: (struct VSprite *) Pointer to the second "dummy" VSprite structure.

ginfo: (struct GelsInfo *) Pointer to an initialized GelsInfo structure.

Example:

```
  /* Give the GelsInfo structure to the system: */
  InitGels( &head, &tail, &ginfo );
```

```
  /* Give the Rastport a pointer to the GelsInfo structure: */
  my_window->RPort->GelsInfo = &ginfo;
```

13.3.6 INITIALIZE THE VSPRITE STRUCTURE

The VSprite structure must be initialized. You need to position the VSprite, set the height, width and depth. The width can for the moment only be 2 words wide (16 pixels), and have a depth of 2 (4 colours). You must also tell the structure that you want a VSprite. You do it by setting the Flags field to "VSPRITE". Finally you must give the structure a pointer to this VSprite's colour table and sprite data.

Example:

```
vsprite.X = x;                    /* Set the X and Y position.   */
vsprite.Y = y;
vsprite.Height = 16;              /* Set the height to 16 lines. */
vsprite.Width = 2;                /* Set the width to 2 words.   */
vsprite.Depth = 2;                /* Set the depth to 2 planes.  */
vsprite.Flags = VSPRITE;          /* We want a VSprite.          */
vsprite.SprColors = colour_table; /* Pointer to the colour table */
vsprite.ImageData = vsprite_data; /* Pointer to the sprite data. */
```

13.3.7 ADD THE VSPRITE TO THE VSPRITE LIST

When all structures are initialized we add the new VSprite to the list. We do it by calling the AddVSprite() function with a pointer to our VSprite structure and a pointer to the Rastport as parameters.

Synopsis: AddVSprite(vsprite, rp);

vsprite: (struct VSprite *) Pointer to an initialized VSprite structure.

rp: (struct RastPort *) Pointer to the RastPort.

Example:

```
  AddVSprite( &vsprite, my_window->RPort );
```

13.3.8 PREPARE THE GEL SYSTEM

The last thing we have to do to see the new VSprite is to sort and draw the GEL list, change the copper list and finally

redraw the screen. All this is done with help of four functions; SortGList(), DrawGList(), MrgCop() and LoadView().

If your program is running under Intuition you should use the functions MakeScreen() and RethinkDisplay() instead of MrgCop() and LoadView().

SortGList() will reorganize the VSprite list so that the further down on the display the sprites are positioned the later they will appear in the list.

Synopsis: SortGList(rp);

rp: (struct RastPort *) Pointer to the RastPort.

DrawGList() will draw the VSprites into the specified Rastport.

Synopsis: DrawGList(rp, vp);

rp: (struct RastPort *) Pointer to the RastPort.

vp: (struct ViewPort *) Pointer to the ViewPort.

MrgCop() reorganizes the Copper list. This is why each VSprite can have its own individual colour values.

Synopsis: MrgCop(view);

view: (struct View *) Pointer to the View structure which copper list should be changed.

LoadView() will create and show the new display.

Synopsis: LoadView(view);

view: (struct View *) Pointer to the View structure which should be used to show the display.

MakeScreen()
Synopsis: MakeScreen(screen);

screen: (struct Screen *) Pointer to the screen which should be affected.

RethinkDisplay() will reorganize the complete display. Note that this function will take quite a long time to execute, so use it only when absolutely needed.

Synopsis: RethinkDisplay();

Example1: (You have created your own display.)

```
SortGList( my_rastport );
DrawGList( my_rastport, my_viewport );
MrgCop( my_view );
LoadView( my_view );
```

Example2: (Your program is running under Intuition.)

```
SortGList( my_window->RPort );
DrawGList( my_window->RPort, &(my_screen->ViewPort) );
MakeScreen( my_screen );
RethinkDisplay();
```

13.3.9 CHANGE THE VSPRITE

Once you have your VSprite you can start to play around with it. You can:

1. Move it around by changing the X and Y fields in the VSprite structure.
2. Change the image of the VSprite by giving the VSprite structure a new pointer to another sprite data.
3. Change the VSprite colours by giving the VSprite structure a new pointer to another colour table.

However, whatever you do with the VSprite you must always call the following functions to be able to see the changes:

```
SortGList();
DrawGList();
MrgCop();
LoadView();
```

or

```
SortGList();
DrawGList();
MakeScreen();
RethinkDisplay();
```

13.3.10 REMOVE VSPRITES

When you want to remove a VSprite from the list you should call the RemVSprite() function.

Synopsis: RemVSprite(vsprite);

vsprite: (struct VSprite *) Pointer to the VSprite you want

to remove.

Example:

```
  RemVSprite( &vsprite );
```

13.4 A COMPLETE EXAMPLE

Here is a complete example on how to use VSprites:

```
/* Example1                                                              */
/* This example demonstrates how to get and use a VSprite. The VSprite */
/* can be moved around by the user by pressing the arrow keys.          */


/* Since we use Intuition, include this file: */
#include <intuition/intuition.h>

/* Include this file since you are using sprites: */
#include <graphics/gels.h>


/* Declare the functions we are going to use: */
void main();
void clean_up();


struct IntuitionBase *IntuitionBase = NULL;
/* We need to open the Graphics library since we are using sprites: */
struct GfxBase *GfxBase = NULL;


/* Declare a pointer to a Screen structure: */
struct Screen *my_screen;

/* Declare and initialize your NewScreen structure: */
struct NewScreen my_new_screen=
{
  0,            /* LeftEdge  Should always be 0. */
  0,            /* TopEdge   Top of the display.*/
  640,          /* Width     We are using a high-resolution screen. */
  200,          /* Height    Non-Interlaced NTSC (American) display. */
  2,            /* Depth     4 colours. */
  0,            /* DetailPen Text should be drawn with colour reg. 0 */
  1,            /* BlockPen  Blocks should be drawn with colour reg. 1 */
  HIRES|SPRITES,/* ViewModes High resolution, sprites will be used. */
  CUSTOMSCREEN, /* Type      Your own customized screen. */
  NULL,         /* Font      Default font. */
  "VSprites!",  /* Title     The screen's title. */
  NULL,         /* Gadget    Must for the moment be NULL. */
  NULL          /* BitMap    No special CustomBitMap. */
};
```

```
/* Declare a pointer to a Window structure: */
struct Window *my_window = NULL;

/* Declare and initialize your NewWindow structure: */
struct NewWindow my_new_window=
{
  0,             /* LeftEdge    x position of the window. */
  0,             /* TopEdge     y position of the window. */
  640,           /* Width       640 pixels wide. */
  200,           /* Height      200 lines high. */
  0,             /* DetailPen   Text should be drawn with colour reg. 0 */
  1,             /* BlockPen    Blocks should be drawn with colour reg. 1 */
  CLOSEWINDOW|   /* IDCMPFlags  The window will give us a message if the */
  RAWKEY,        /*             user has selected the Close window gad, */
                 /*             or if the user has pressed a key. */
  SMART_REFRESH| /* Flags       Intuition should refresh the window. */
  WINDOWCLOSE|   /*             Close Gadget. */
  WINDOWDRAG|    /*             Drag gadget. */
  WINDOWDEPTH|   /*             Depth arrange Gadgets. */
  WINDOWSIZING|  /*             Sizing Gadget. */
  ACTIVATE,      /*             The window should be Active when opened. */
  NULL,          /* FirstGadget No Custom gadgets. */
  NULL,          /* CheckMark   Use Intuition's default CheckMark. */
  "Use the arrow keys to move the VSprite!", /* Title */
  NULL,          /* Screen      Will later be connected to a custom scr. */
  NULL,          /* BitMap      No Custom BitMap. */
  80,            /* MinWidth    We will not allow the window to become */
  30,            /* MinHeight   smaller than 80 x 30, and not bigger */
  640,           /* MaxWidth    than 640 x 200. */
  200,           /* MaxHeight */
  CUSTOMSCREEN   /* Type        Connected to the Workbench Screen. */
};


/* 1. Declare and initialize some sprite */
/*    data for each VSprite:             */
UWORD chip vsprite_data[]=
{
  0x0180, 0x0000,
  0x03C0, 0x0000,
  0x07E0, 0x0000,
  0x0FF0, 0x0000,
  0x1FF8, 0x0000,
  0x3FFC, 0x0000,
  0x7FFE, 0x0000,
  0x0000, 0xFFFF,
  0x0000, 0xFFFF,
  0x7FFE, 0x7FFE,
  0x3FFC, 0x3FFC,
  0x1FF8, 0x1FF8,
  0x0FF0, 0x0FF0,
  0x07E0, 0x07E0,
  0x03C0, 0x03C0,
  0x0180, 0x0180,
};
```

```
/* 2. Declare three VSprite structures. One will be used, */
/*    the other two are "dummies":                        */
struct VSprite head, tail, vsprite;


/* 3. Decide the VSprite's colours:           */
/*                          RGB     RGB     RGB */
WORD colour_table[] = { 0x000F, 0x00F0, 0x0F00 };


/* 4. Declare a GelsInfo structure: */
struct GelsInfo ginfo;


/* This boolean variable will tell us if the VSprite is in */
/* the list or not:                                        */
BOOL vsprite_on = FALSE;


/* This program will not open any console window if run from */
/* Workbench, but we must therefore not print anything.      */
/* Functions like printf() must therefore not be used.       */
void _main()
{
 /* The GelsInfo structure needs the following arrays: */
  WORD nextline[ 8 ];
  WORD *lastcolor[ 8 ];

  /* Sprite position: */
  WORD x = 40;
  WORD y = 40;

  /* Direction of the sprite: */
  WORD x_direction = 0;
  WORD y_direction = 0;

  /* Boolean variable used for the while loop: */
  BOOL close_me = FALSE;

  ULONG class; /* IDCMP */
  USHORT code; /* Code */

  /* Declare a pointer to an IntuiMessage structure: */
  struct IntuiMessage *my_message;


  /* Open the Intuition Library: */
  IntuitionBase = (struct IntuitionBase *)
    OpenLibrary( "intuition.library", 0 );

  if( IntuitionBase == NULL )
    clean_up(); /* Could NOT open the Intuition Library! */


  /* 5. Open the Graphics Library:                          */
```

```
/* Since we are using sprites we need to open the Graphics Library: */
/* Open the Graphics Library: */
GfxBase = (struct GfxBase *)
  OpenLibrary( "graphics.library", 0);

if( GfxBase == NULL )
  clean_up(); /* Could NOT open the Graphics Library! */


/* We will now try to open the screen: */
my_screen = (struct Screen *) OpenScreen( &my_new_screen );

/* Have we opened the screen succesfully? */
if(my_screen == NULL)
  clean_up();


my_new_window.Screen = my_screen;


/* We will now try to open the window: */
my_window = (struct Window *) OpenWindow( &my_new_window );

/* Have we opened the window succesfully? */
if(my_window == NULL)
  clean_up(); /* Could NOT open the Window! */


/* 6. Initialize the GelsInfo structure: */

/* All sprites except the first two may be used to draw */
/* the VSprites: ( 11111100 = 0xFC )                    */
ginfo.sprRsrvd = 0xFC;
/* If we do not exclude the first two sprites, the mouse */
/* pointer's colours may be affected.                     */


/* Give the GelsInfo structure some memory: */
ginfo.nextLine = nextline;
ginfo.lastColor = lastcolor;


/* Give the Rastport a pointer to the GelsInfo structure: */
my_window->RPort->GelsInfo = &ginfo;


/* Give the GelsInfo structure to the system: */
InitGels( &head, &tail, &ginfo );


/* 7. Initialize the VSprite structure: */

vsprite.Flags = VSPRITE; /* It is a VSprite.           */
vsprite.X = x;           /* X position.                */
vsprite.Y = y;           /* Y position.                */
vsprite.Height = 16;     /* 16 lines tall.             */
vsprite.Width = 2;       /* Two bytes (16 pixels) wide. */
```

```
vsprite.Depth = 2;       /* Two bitplanes, 4 colours.   */


/* Pointer to the sprite data: */
vsprite.ImageData = vsprite_data;

/* Pointer to the colour table: */
vsprite.SprColors = colour_table;


/* 8. Add the VSprites to the VSprite list: */
AddVSprite( &vsprite, my_window->RPort );

/* The VSprite is in the list. */
vsprite_on = TRUE;


/* Stay in the while loop until the user has selected the Close window */
/* gadget: */
while( close_me == FALSE )
{
  /* Stay in the while loop as long as we can collect messages */
  /* successfully: */
  while(my_message = (struct IntuiMessage *) GetMsg(my_window->UserPort))
  {
    /* After we have collected the message we can read it, and save any */
    /* important values which we maybe want to check later: */
    class = my_message->Class;
    code  = my_message->Code;

    /* After we have read it we reply as fast as possible: */
    /* REMEMBER! Do never try to read a message after you have replied! */
    /* Some other process has maybe changed it. */
    ReplyMsg( my_message );

    /* Check which IDCMP flag was sent: */
    switch( class )
    {
      case CLOSEWINDOW:     /* Quit! */
             close_me=TRUE;
             break;

      case RAWKEY:          /* A key was pressed! */
             /* Check which key was pressed: */
             switch( code )
             {
               /* Up Arrow: */
               case 0x4C:      y_direction = -1; break; /* Pre */
               case 0x4C+0x80: y_direction = 0;  break; /* Rel */

               /* Down Arrow: */
               case 0x4D:      y_direction = 1; break; /* Pre */
               case 0x4D+0x80: y_direction = 0; break; /* Rel */

               /* Right Arrow: */
               case 0x4E:      x_direction = 2; break; /* Pre */
               case 0x4E+0x80: x_direction = 0; break; /* Rel */
```

```
                /* Left Arrow: */
                case 0x4F:      x_direction = -2; break; /* Pre */
                case 0x4F+0x80: x_direction = 0;  break; /* Rel */
              }
              break;
      }
    }


    /* 12. Play around with the VSprite: */

    /* Change the x/y position: */
    x += x_direction;
    y += y_direction;

    /* Check that the sprite does not move outside the screen: */
    if(x > 320)
      x = 320;
    if(x < 0)
      x = 0;
    if(y > 200)
      y = 200;
    if(y < 0)
      y = 0;

    vsprite.X = x;
    vsprite.Y = y;


    /* 9. Sort the Gels list: */
    SortGList( my_window->RPort );

    /* 10. Draw the Gels list: */
    DrawGList( my_window->RPort, &(my_screen->ViewPort) );

    /* 11. Set the Copper and redraw the display: */
    MakeScreen( my_screen );
    RethinkDisplay();
  }


  /* Free all allocated memory: (Close the window, libraries etc) */
  clean_up();

  /* THE END */
}


/* This function frees all allocated memory. */
void clean_up()
{
 /* 13. Remove the VSprites: */
  if( vsprite_on )
    RemVSprite( &vsprite );

  if( my_window )
```

```
    CloseWindow( my_window );

  if(my_screen )
    CloseScreen( my_screen );

  if( GfxBase )
    CloseLibrary( GfxBase );

  if( IntuitionBase )
    CloseLibrary( IntuitionBase );

  exit();
}
```

13.5 FUNCTIONS

InitGels()

This function "gives" an already prepared GelsInfo structure to the system.

Synopsis: InitGels(head, tail, ginfo);

head: (struct VSprite *) Pointer to the first "dummy" VSprite structure.

tail: (struct VSprite *) Pointer to the second "dummy" VSprite structure.

ginfo: (struct GelsInfo *) Pointer to an initialized GelsInfo structure.

AddVSprite()

This function will add a VSprite to the VSprite list.

Synopsis: AddVSprite(vsprite, rp);

vsprite: (struct VSprite *) Pointer to an initialized VSprite structure.

rp: (struct RastPort *) Pointer to the RastPort.

SortGList()

This function will reorganize the VSprite list so that the further down on the display the sprites are positioned the later they will appear in the list.

Synopsis: SortGList(rp);

rp: (struct RastPort *) Pointer to the RastPort.

DrawGList()

This function will draw the VSprites into the specified Rastport.

Synopsis: DrawGList(rp, vp);

rp: (struct RastPort *) Pointer to the RastPort.

vp: (struct ViewPort *) Pointer to the ViewPort.

MrgCop()

This function reorganizes the Copper list. This is why each VSprite can have its own individual colour values.

Synopsis: MrgCop(view);

view: (struct View *) Pointer to the View structure which copper list should be changed.

LoadView()

This function will create and show the new display.

Synopsis: LoadView(view);

view: (struct View *) Pointer to the View structure which should be used to show the display.

MakeScreen()

This function will recalculate the screen display values.

Synopsis: MakeScreen(screen);

screen: (struct Screen *) Pointer to the screen which should be affected.

RethinkDisplay()

This function will reorganize the complete display. Note that this function will take quite a long time to execute, so use it only when absolutely needed.

Synopsis: RethinkDisplay();

13.6 EXAMPLES

The printed version of these examples can be found in Appendix A (Part IV: Appendices).

Example1

This example demonstrates how to get and use a VSprite. The VSprite can be moved around by the user by pressing the arrow keys.

Example2

This example demonstrates how to use several VSprites each with its own colour table.

Example3

This program demonstrates how to animate several (!) VSprites.

14 HINTS AND TIPS

14.1 INTRODUCTION

This chapter contains a mixture of useful hints and tips.

14.2 NTSC VERSUS PAL

If you have an American Amiga you can open screens that are up to 200 lines high. (400 with interlace) This system is called NTSC. However, an European Amiga can open screens that are up to 256 lines high. (512 with interlace) This system is called PAL.

14.2.1 HOW TO WRITE PROGRAMS THAT WILL FIT BOTH SYSTEMS

If you want to write programs that will fit both systems you must do one of the following things:

1. Do not open screens that are taller than 200 lines. The program can then be run on both systems. However, this is not always the best solution since the PAL user will be annoyed when he can not use the last 56 lines of the display.

2. Open the Graphics library and see what type of system your program is running under. If you discover that it is running on a NTSC system, use only the first 200 lines. If the program is running on a PAL system you may use all 256 lines.

If you decide that your program should only use screens up to 200 lines tall you should still check what system the program is running under. If you discover that it is an European Amiga you can position the display 28 (56/2) lines down, and the large unused are at the bottom of the PAL displays will visually disappear.

14.2.2 NTSC OR PAL?

To check which system, NTSC or PAL, your program is running on you simply open the graphics library and check the DisplayFlags.

Here is a short demonstration program:

```
/* Example 1                                                  */
/* This example tell you if you have an American (NTSC) or */
/* European (PAL) system.                                     */


/* Declares commonly used data types, such as UWORD etc: */
#include <exec/types.h>

/* This header file declares the GfxBase structure: */
#include <graphics/gfxbase.h>


/* Pointer to the GfxBase structure. NOTE! This pointer must */
/* always be called "GfxBase"!                                */
struct GfxBase *GfxBase;


main()
{
  /* Open the Graphics Library: (any version) */
  GfxBase = (struct GfxBase *)
    OpenLibrary( "graphics.library", 0 );

  if( !GfxBase )
    exit(); /* ERROR! Could not open the Graphics Library! */


  if( GfxBase->DisplayFlags & NTSC )
    printf( "You have an American (NTSC) Amiga.\n" );

  if( GfxBase->DisplayFlags & PAL )
    printf( "You have an European (PAL) Amiga.\n" );


  /* Close the Graphics Library: */
  CloseLibrary( GfxBase );
}
```

14.3 PROGRAMS RUNNING UNDER WORKBENCH

A program can either be run from the CLI or Workbench. If a program is run from workbench a special "console" window is automatically opened. All text will then be outputted in that window. If the program was run from CLI no special "console" window is opened since the text can be printed in the CLI window.

The special window is, as said above, automatically opened when the program is run from workbench. However, sometimes you do not want this window. If you will not print anything the window is unnecessary, and somehow annoying.

To get rid of the window you simply call your main() function _main(). (Note the special symbol "_".) The program will then not open the console window if run from workbench, but the disadvantage is of course that you can not print any text with the printf() function etc. If you would try to print text without having any window to print it in, the system will crash. (Not to be recommended.)

Here are two examples. The first one is a normal C program that simply prints "Hello!". If the program is run from the CLI, the text will be printed in the CLI window. If the program is run from workbench, a console window will be opened and the text printed in it. The second example will not open any console window if run from workbench, but it can then of course not print anything.

```
/* Example 2                                                    */
/* This program will print "Hello!" in the CLI window if        */
/* started from CLI, or the text will be printed in a special */
/* window that is automatically opened if run from workbench. */

void main();

void main()
{
  int loop;

  printf( "Hello!\n" );

  /* Wait for a while: */
  for( loop = 0; loop < 500000; loop++ )
    ;
}


/* Example 3 */
/* This program will not open any console window if run from */
/* workbench. The disadvantage is of course that you can not */
/* use any "console functions" such as printf().             */

void _main();

void _main() /* Note the special character in front of main()! */
{
  int loop;

  /* Wait for a while: */
  for( loop = 0; loop < 500000; loop++ )
    ;
}
```

The reason why no console window is opened in Example 3 is because we have called our main() function _main(). The first function that is processed by C is actually not the main() function, it is the _main() function which then calls main().

The _main() function is like a preprocessor that initializes the argc and argv variables. It also checks if the program is running under workbench, and will then open a console window.

The _main() function can be found on the fourth Lattice C disk, in the source directory called "umain.c".

If you call your own main() function _main(), the prewritten _main() function will not be used. No window will be opened, but no text can be printed, and you have to preprocess the argc and argv variables yourself.

14.4 CHECK IF THE PROGRAM WAS STARTED FROM CLI OR WORKBENCH

It sometimes happen that you would like to know if the program was run from Workbench or a CLI window. It is actually very simple to check. The "argc" value will always be equal to one ore more if the program was run from a CLI window, while it will be equal to zero if the program was run from Workbench.

Here is a short program that tells you if it was run from a CLI window or from Workbench.

```
/* Example 4                                                  */
/* This program tells you if it was run from workbench or */
/* from a CLI window.                                         */

void main();

void main( argc, argv )
int argc;
char *argv[];
{
  int loop;


  if( argc )
    printf( "The program was run from a CLI window!\n" );
  else
    printf( "The program was run from Workbench!\n" );


  /* Wait for a while: */
  for( loop = 0; loop < 500000; loop++ )
    ;
}
```

14.5 EXAMPLES

The printed version of these examples can be found in Appendix A (Part IV: Appendices).

Example1

This example tell you if you have an American (NTSC) or European (PAL) system.

Example2

This program will print "Hello!" in the CLI window if started from CLI, or the text will be printed in a special window that is automatically opened if run from workbench.

Example3

This program will not open any console window if run from workbench. The disadvantage is of course that you can not use any "console functions" such as printf().