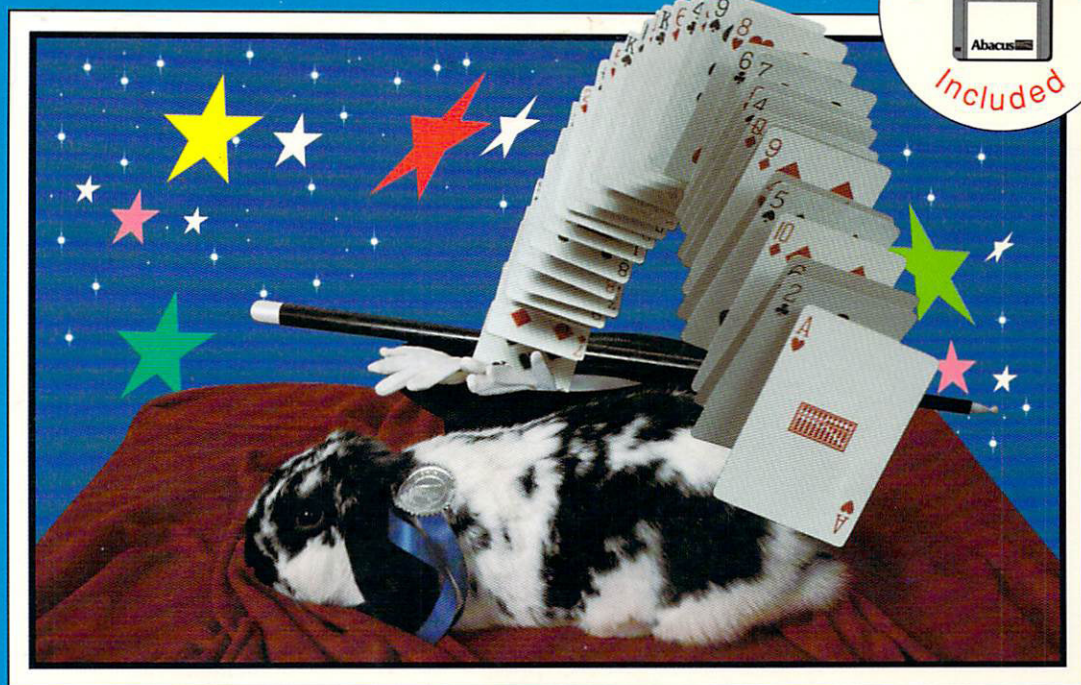


# The <sup>Best</sup> Amiga<sup>®</sup> Tricks & Tips

A valuable collection of useful and productive hints for using your Amiga

by Wolf-Gideon Bleek, Tobias Weltner, and Stefan Maelger



**Abacus**   
A Data Becker Book



# The Best Amiga Tricks & Tips

Wolf-Gideon Bleek  
Tobias Weltner  
Stefan Maelger



A Data Becker  
Published by

**Abacus** 

First Printing, 1990  
Printed in U.S.A.  
Copyright © 1990

Abacus  
5370 52nd Street, SE  
Grand Rapids, MI 49512

Copyright © 1989

Data Becker, GmbH  
Merowingerstrasse 30  
4000 Deusseldorf, West Germany

Editors

Scott Slaughter, Jim D'Haem, Robbin Markley

This book is copyrighted. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior written permission of Abacus Software or Data Becker, GmbH.

Every effort has been made to ensure complete and accurate information concerning the material presented in this book. However, Abacus Software can neither guarantee nor be held legally responsible for any mistakes in printing or faulty instructions contained in this book. The authors always appreciate receiving notice of any errors or misprints.

AmigaBASIC is a trademark or registered trademark of Microsoft Corporation. Amiga 500, Amiga 1000, Amiga 2000, Amiga and C64 are trademarks or registered trademarks of Commodore-Amiga, Inc. Lattice C and Lattice are trademarks or registered trademarks of Lattice Corporation. Aztec C and Aztec are trademarks or registered trademarks of Manx Software Systems. IBM is a trademark or registered trademark of International Business Machines, Inc. Atari ST is a trademark or registered trademark of Atari Corporation.

This book contains trade names and trademarks of many companies and products. Any mention of these names or trademarks in this book is not intended to either convey endorsement or other associations with this book.

Library of Congress Cataloging-in-Publication Data  
Bleek, Wolf-Gideon, 1970-

The best Amiga tricks & tips / Wolf-Gideon Bleek, Tobias Weltner, Stefan Maelger  
p. cm.

Includes index.

ISBN 1-55755-107-3 : \$19.95

1. Amiga (computers)--Programming. I. Weltner, Tobias, 1962- . II. Maelger, Stefan, 1965- . III. Title. IV.

Title: Best Amiga tricks and tips

QA76.8.A177B584 1990

005.265--dc20

90-19378

CIP

# Table of Contents

<b>1.</b>	<b>Introduction.....</b>	<b>1</b>
<b>2.</b>	<b>The AmigaDOS Shell.....</b>	<b>5</b>
2.1	Shell questions and answers.....	8
2.2	AmigaDOS commands.....	23
2.3	New Startup-sequences.....	41
2.4	Using Mount.....	43
2.4.1	Renaming commands.....	45
2.4.2	Less is more.....	46
2.4.3	Printer spooler.....	47
<b>3.</b>	<b>AmigaBASIC.....</b>	<b>49</b>
3.1	Kernel commands.....	52
3.2	AmigaBASIC graphics.....	54
3.2.1	Changing drawing modes.....	54
3.2.2	Changing typestyles.....	58
3.2.3	Move – cursor control.....	60
3.2.4	Faster IFF transfer.....	61
3.2.5	IFF brushes as objects.....	71
3.2.6	Another floodfill.....	78
3.2.7	Window manipulation.....	79
3.2.7.1	Borderless BASIC windows.....	79
3.2.7.2	Gadgets on, gadgets off.....	80
3.2.7.3	DrawBorder.....	81
3.2.7.4	ChangeBorderColor.....	83
3.2.7.5	Monocolor Workbench.....	84
3.2.7.6	PlaneCreator and HAM-Halfbrite.....	85
3.2.7.7	The coordinate problem.....	88
3.3	Fade-in and fade-out.....	89
3.3.1	Basic fading.....	89
3.3.2	Fade-over.....	91
3.3.3	Fading RGB color scales.....	93
3.4	Fast vector graphics.....	96
3.4.1	Model grids.....	96
3.4.2	Moving grid models.....	101
3.4.3	Moving with operating system routines.....	101
3.4.4	3-D graphics for 3-D glasses.....	105
3.5	The Amiga fonts.....	114
3.6	Fast and easy PRINT.....	117
<b>4.</b>	<b>User-friendliness.....</b>	<b>123</b>
4.1	Input Gadgets.....	126
4.1.1	An Intuition window.....	126
4.1.2	Gadgets.....	129
4.1.3	Gadget borders and text.....	135
4.1.4	User-friendly gadgets.....	140
4.1.5	Scrolling tables.....	146

4.1.6	Proportional gadgets.....	153
4.2	Rubberbanding.....	159
4.2.1	Rectangles in rubberbanding .....	159
4.2.2	Creating shapes.....	161
4.2.3	Object positioning.....	163
4.3	Status lines & animation .....	166
<b>5.</b>	<b>AmigaBASIC Internals .....</b>	<b>175</b>
5.1	File Monitor.....	178
5.1.1	Using the file monitor.....	191
5.1.2	Patching files with the monitor.....	193
5.1.3	Patching AmigaBASIC.....	194
5.2	AmigaBASIC file structure.....	195
5.2.1	Determining filetype .....	195
5.2.1.1	Checking for a BASIC file .....	196
5.2.1.2	Checking the program header .....	198
5.2.2	ASCII files.....	200
5.2.3	Binary files.....	201
5.2.3.1	Structure of an AmigaBASIC line.....	202
5.3	Utility programs.....	212
5.3.1	DATA generator.....	212
5.3.2	Cross-reference list.....	216
5.3.3	Blank line killer .....	223
5.3.4	REM killer.....	229
5.3.5	Listing variables.....	233
5.3.6	Removing "extra" variables .....	238
5.3.7	Self-modifying programs.....	239
<b>6.</b>	<b>The Workbench.....</b>	<b>243</b>
6.1	Using the Workbench.....	246
6.1.1	Keyboard tricks .....	246
6.1.2	The Trashcan .....	247
6.1.3	Extended selection.....	247
6.2	Information .....	249
6.2.1	The Information screen .....	249
<b>7.</b>	<b>Icons .....</b>	<b>251</b>
7.1	Icon types .....	254
7.2	Icon design.....	255
7.2.1	DiskObject structure.....	255
7.2.2	Drawer structure .....	257
7.2.3	Image structure.....	258
7.2.4	DefaultTool text.....	259
7.2.5	ToolTypes text.....	259
7.2.6	Icon analyzer.....	260
7.3	Making your own icons.....	266
7.3.1	Two graphics, one icon.....	266
7.3.2	Text in graphics .....	266
7.3.3	The icon editor.....	267
7.3.4	Color changes.....	273

<b>8.</b>	<b>Error trapping</b> .....	<b>275</b>
8.1	Errors—and why .....	278
8.1.1	Disk access errors .....	278
8.1.2	User input errors.....	279
8.1.3	Menu errors.....	279
8.2	Trapping errors.....	280
8.2.1	Error checking programs .....	280
8.2.2	Trapping user input errors .....	287
8.3	Correcting errors.....	291
8.3.1	Ghosting menu items.....	291
<b>9.</b>	<b>Machine language</b> .....	<b>297</b>
9.1	Division by zero handler .....	300
9.2	Attention: Virus alert! .....	303
9.2.1	The ultimate virus killer .....	304
9.3	Machine language and BASIC.....	308
9.3.1	Assembler and C programs from BASIC.....	310
9.3.2	BASIC enhancement: ColorCycle.....	313
9.3.3	Putting the mouse to sleep - Zzz .....	315
<b>10.</b>	<b>Input and output</b> .....	<b>317</b>
10.1	Direct disk access.....	321
10.1.1	The trackdisk.device commands .....	328
10.1.2	Multiple disk drive access.....	329
10.1.3	Sector design .....	329
10.2	Memory handling .....	332
10.2.1	Reserving memory through variables.....	332
10.2.2	Allocating memory .....	332
10.3	The Printer Device .....	335
10.3.1	Controlling printer parameters.....	335
10.3.2	Graphic dumps using the printer device.....	340
<b>11.</b>	<b>Hardware hacking</b> .....	<b>347</b>
11.1	Disabling memory expansion.....	351
11.1.1	The 2000A board.....	351
11.1.2	The Amiga 500: printed circuit board.....	352
11.2	Disk drive switching .....	354
11.3	Installing a 68010.....	355
11.4	The roar of the fans .....	358
11.5	New processor information.....	359
11.5.1	The 68010: high power, low price .....	359
11.5.2	The 68012: low cost, high memory .....	361
11.5.3	Monster processors: 68020, 68030, 6888x .....	361
<b>12.</b>	<b>Hints and tips</b> .....	<b>363</b>
12.1	Tips for the Shell .....	365
12.2	Tips for AmigaBASIC.....	370
12.3	Printer tips.....	373
12.4	Amiga Hints.....	375

<b>13.</b>	<b>Devices and the FastFile System .....</b>	<b>377</b>
13.1	The PIPE device.....	380
13.2	The Speak device.....	381
13.3	The NewCon device .....	382
13.4	The FastFileSystem.....	383
13.4.1	FFS and hard disks.....	383
13.4.2	FFS and recoverable RAM disk.....	384
13.5	The new math libraries .....	386
<b>Appendices</b> .....		<b>391</b>
Appendix A	AmigaBASIC tokens.....	393
Appendix B	Other tokens.....	398
Appendix C	Shell Escape sequences .....	399
Appendix D	Printer Escape Sequences.....	400
Appendix E	Program Notes.....	402
Index.....		405



# **1**

# **Introduction**



---

# 1. Introduction

Think back to the first time you sat down at your Amiga. You probably experienced the following reactions: excitement, astonishment, surprise and confusion—probably in that order. Yes, the Amiga really is a super personal computer. But there's so much you can do with an Amiga that you often don't know where to begin. How should you begin to apply the Amiga to your tastes? How do you make the most of the Amiga's many capabilities?


If you are new to the Amiga, you probably have dozens of questions by now. To start you off, Chapter Two of this book describes work with the Command Line Interface (CLI) or Shell.

Part of this book explains methods and programming techniques for getting the most out of Microsoft's AmigaBASIC, with special emphasis on using existing system modules from the software supplied with your Amiga. You'll find handy AmigaBASIC program routines in this book that let you use the various fonts and type styles, use rubberbanding, create borderless windows and even a disk monitor for exploring the machine language code of the disk drive.

Chapter Five describes the handling of AmigaDOS. It shows how to use the AmigaDOS commands from the Shell, and how these commands can be useful to you.

Other subjects covered in Amiga Tricks and Tips include the handling and changing of the Workbench. This includes manipulation and editing of icons for your own purposes.

The Amiga Workbench is an ever expanding and improving system. Programs are changed or added to the Workbench to upgrade and improve the Amiga operating system. This book covers the two latest versions of the Amiga operating system, Workbench 2.0 and Workbench 1.3. The differences between the two versions will be noted whenever possible. For clarity, Workbench 2.0 will be referred to simply as 2.0 and Workbench 1.3 will be referred to as 1.3.

We'll use various symbols to represent the keys on the keyboard. For example,  represents the <Enter> key.

We'll use <Commodore logo> and right <Amiga> keys in this book instead of using a specific key symbol. Therefore when we tell you to press the <Commodore logo> key and you only have a left <Amiga> key, press the left <Amiga> key instead.



# **2**

# **The AmigaDOS**

# **Shell**



---

## 2. The AmigaDOS Shell

The AmigaDOS Shell allows you to access AmigaDOS commands. The original AmigaDOS interface was called the SHELL. SHELL stands for Command Line Interface. This user interface is controlled from the keyboard. Neither the icons nor the mouse can be used in the Shell.

The SHELL works closely with AmigaDOS, the Disk Operating System. Many special SHELL commands make working with diskettes faster and more convenient than performing the same functions from the Workbench. Some disk commands must be called from the SHELL, since they cannot be directly accessed by Intuition. Intuition is the part of the Amiga's operating system that acts as an interface between the user and the window and the mouse technique of handling diskettes, programs and files.

You usually access the Shell from Intuition. However, you can also call AmigaDOS commands from BASIC and C programs.

---

## 2.1 Shell questions and answers

Many new Amiga users ask questions about the Shell. Below are 20 of the most often asked Shell questions and their answers.

**Question 1: How do I get into the Shell?**

**Answer:** The Shell is included on every Workbench diskette. Here's how you can access it:

- a) Accessing Shell with Intuition (the usual method):
  - Boot your system with the Workbench diskette in the drive. You'll see the deep blue Workbench screen.
  - Click the Workbench disk icon. This opens a window named **Workbench**, which contains a number of icons.
  - Click on the Shell icon. This opens a window named **Amiga Shell** (New Shell in 1.3). You can enlarge or reduce the size of this window and in 2.0 you can close it with the close gadget. The Workbench 1.3 Shell doesn't have the close gadget. You now have your own Shell.
- b) Accessing Shell commands through AmigaDOS:
  - AmigaDOS has a command called `execute` which executes AmigaDOS commands in a script file.
  - You can also access AmigaDOS through the system libraries, which is how AmigaBASIC and the C programming language communicate with AmigaDOS.
- c) Interrupting the booting process (the easiest method of calling the Shell):
  - Boot your system as usual. When the Kickstart diskette (Amiga 1000) or Kickstart in ROM (Amiga 500 and 2000) has successfully loaded, the icon requesting a Workbench diskette appears on the screen.
  - Insert the Workbench diskette in the drive. The icon disappears and the system boots up.



- When the **AmigaDOS** window appears, hold down the **Ctrl** key and press the **D** key. The following message appears:

```
SHELL *** BREAK
1>
```

- You are now in the Shell. Enter:
 

```
1> loadwb
```
- You can now access all functions of the Shell.

**Question 2: How do I get out of the Shell?**

**Answer:** In 2.0 simply click on the close gadget in the upper left corner of the Shell window. In 1.3 the Shell window doesn't have a close gadget. You can exit the Shell in 2.0 and 1.3 by typing in the following:

```
1> endshell
```

If you have started programs from Shell, the Shell window remains open while the programs continue running.

**Question 3: I don't have a typewriter, but I have a printer connected to my Amiga. Can I use my Amiga to type?**

**Answer:** Yes. Type in the following Shell command:

```
1> copy * to prt:
```

The asterisk (\*) represents the open Amiga Shell window. The Shell prompt 1> disappears after this entry but the cursor remains on the screen. Now everything you type is sent to the printer after you press the **←** key, similar to a typewriter with one-line correction capability.

Hold down the **Ctrl** key and press the **□** key to exit typewriter mode.

You can also copy text from the Shell window to another window. Type this and press the **←** key to display your text in another window:

```
1> copy * to CON:10/10/300/100/copy_text
```

Re-activate the Shell window by clicking on it. Press and hold the **Ctrl** key and press the **□** key to stop this command.

**Question 4:** I only have one disk drive. Every time I call a `Shell` command, the Amiga wants the Workbench diskette. Can I store the Workbench in memory?

**Answer:** Each `Shell` command is a program stored in directory `c:` of the Workbench diskette. When you call a `Shell` command, the Amiga loads this program from the Workbench diskette. This saves system memory because the `Shell` commands aren't occupying any of that memory. On the other hand, if you only have one disk drive, you may spend too much of your time swapping diskettes.

Buying a second disk drive is one solution to the problem. If you have enough system memory, you can store the AmigaDOS commands that you use frequently in RAM with the `Resident` command. Commands loaded using `resident` are loaded into working memory once. When the command is called from the `Shell` the resident list is searched first and if the command is found, it is executed. To make the `dir` command `resident` enter the following:

```
1> resident c:dir add
```

Enter the following to view a list of AmigaDOS commands which are currently resident in memory:

```
1> resident
```

## AmigaDOS 2.0

In AmigaDOS 2.0 all the AmigaDOS commands were rewritten for compactness and speed. This allows you to make many commands internal commands. Since you can directly execute these commands, it eliminates the need to load them from diskette. The Amiga designers recognized the flexibility of a system that calls commands from diskette. Therefore, they built in an internal command override system, keeping the best of both worlds, internal and external commands. The following are the internal commands of AmigaDOS 2.0:

Alias	INTERNAL
Ask	INTERNAL
CD	INTERNAL
Echo	INTERNAL
Else	INTERNAL
EndCLI	INTERNAL
EndIf	INTERNAL
EndShell	INTERNAL
EndSkip	INTERNAL
Failat	INTERNAL
Fault	INTERNAL
Get	INTERNAL
Getenv	INTERNAL
If	INTERNAL
Lab	INTERNAL
NewCLI	INTERNAL
NewShell	INTERNAL
Path	INTERNAL

Prompt	INTERNAL
Quit	INTERNAL
Resident	INTERNAL
Run	INTERNAL
Set	INTERNAL
Setenv	INTERNAL
Skip	INTERNAL
Stack	INTERNAL
Unalias	INTERNAL
Unset	INTERNAL
Unsetenv	INTERNAL
Why	INTERNAL

**Question 5: How can I stop a Shell command as it executes?**

**Answer:** Press **Ctrl C** to stop any command. **Ctrl D** sends an execute command to stop the program as soon as possible.

**Question 6: Are there wildcard characters on the Amiga like the \* and ? found on the MS-DOS computers?**

**Answer:** The Amiga uses the character combination #? as a wildcard. The asterisk (\*) represents the current Shell window and therefore cannot be used as a wildcard on the Amiga. You can delete all the files on the RAM disk by typing in:

1) delete ram: #?

Try this command:

1) run amig#?

The Amiga can't execute this command because it doesn't know which program to execute. There may be several programs with names beginning with the letters "amig".

**Question 7: How can I determine the syntax of a certain Shell command while working in the Shell?**

**Answer:** Almost all Shell commands have a help template. If you don't remember the exact syntax of a command, enter the command name followed by a space and a question mark. For example:

1> list ?

The 2.0 Shell displays:

```
DIR/M,P=PAT/K,KEYS/S,DATES/S,NODATES/S,TO/K,
SUB/K,SINCE/K,UPTO/K,QUICK/SBLOCK/S,
NOHEAD/S,FILES/S,DIR/S,LFORMAT/K,ALL/S:
```

DIR represents directory. The current directory is listed if DIR is omitted. All other options have a condition, or *argument*, added to the name of the option:

```

/A:   This requires a specific argument
/K:   This argument requires a parameter
/S:   This argument has no parameters

```

The following command displays the programs in df0: with the various starting memory blocks but without dates:

```
1> list df0: keys nodates
```

Type in this command sequence to print the programs in df0: written between October 4, 1989, and today.

```
1> list df0: since 04-Oct-89 upto today
```

**Question 8:** How can I copy a program using one disk drive?

**Answer:** There are three methods of copying programs with one disk drive.

- a) Using the RAM disk:
  - Copy the program you want copied, as well as the `copy` program, from the source diskette into the RAM disk:
 

```
1> copy program to ram:
1> copy c/copy to ram:
```
  - The `copy` program was copied by the second command sequence. This means that you won't have to insert the Workbench diskette during the copying procedure.
  - Remove the source diskette and put the destination diskette in the drive.
  - Type in the following to copy the program onto the destination diskette:
 

```
1> ram: copy ram:program to df0:
```
  - Remove the destination diskette from the drive and insert the Workbench diskette.
  - Enter this line to delete the RAM disk:
 

```
1> delete ram: #?
```

- b) Using the RAM disk and the Resident command:
- Make the copy command resident with:  
`1> resident c:copy add`
  - Next copy the desired file to the RAM drive:  
`1> copy program to ram:`
  - Remove the source diskette and put the destination diskette in the drive.
  - Type in the following to copy the program onto the destination diskette:  
`1> copy ram:program to df0:`
  - Remove the destination diskette from the drive and insert the Workbench diskette.
  - Enter this line to delete the RAM disk:  
`1> delete ram:#?`
  - Enter this line to remove the copy command from the resident list:  
`1> resident copy remove`
- c) Using the Intuition icons:
- Insert the source diskette and click the source diskette's icon.
  - Remove the original diskette as soon as the desired program icon appears. Then insert the destination diskette.
  - Open the destination diskette by clicking its icon. Now you can drag the program icon from the source diskette to the destination diskette's window.
  - Requesters tell you when to exchange diskettes (remember not to remove a diskette from a drive until the disk light turns off).

**Note:**

There are programs on your Workbench diskette which aren't listed in Intuition windows. This is because they have no icons assigned to them. Here's how you can assign icons to these programs.

- Insert the Workbench diskette. Type in the following lines:
 

```
1> copy df0:clock.info to ram:
1> rename ram:clock.info as ram:program.info
1> copy c/copy to ram:
```
- Insert the diskette which contains the original program. Enter:
 

```
1> ram: copy ram:program.info to df0:
```
- Now your program (here just called `program`) has an icon.
- Insert the Workbench diskette and delete the RAM disk:
 

```
1> delete ram:#?
```

**Question 9:** How can I print all the AmigaDOS commands on my printer?

**Answer:** Type in this command sequence to print the complete AmigaDOS command list:

```
1> list quick sys:c to prt:
```

The `quick` option prints the command names only. The file creation date, the time, the protection status and the file size aren't printed. The AmigaDOS commands themselves are in the `c:` subdirectory, on the system disk `sys:.` The list prints out even faster if you use the multitasking capabilities of the Amiga:

```
1> run list quick sys:c to prt:
```

This line opens another task for handling printer output. The Amiga prints the command words in the background, leaving you free to work on other things.

**Question 10:** How can I copy a program using two disk drives?

**Answer:** Enter this line in the Shell to copy the program:

```
1> copy df0:originalprogram to df1:
```

`originalprogram` is the name of your program. It must be in directory `df0:` of the diskette in drive 0 for this command to work correctly.

You can also copy a program by moving the program icon from one disk window to another (see Question 8, part c).

**Question 11: How can I copy an entire diskette?**

**Answer:** Use the `diskcopy` command.

a) If you have one disk drive:

- Insert the Workbench diskette.
- Enter the following Shell command:  

```
1> diskcopy from df0: to df0: name "copy"
```
- Requesters tell you to exchange the source and destination diskettes as needed.

b) If you have two disk drives:

- Insert the Workbench diskette.
- Enter the following Shell command:  

```
1> diskcopy from df0: to df1: name "copy"
```
- Insert the source diskette in drive 0 and the target diskette in drive 1. No diskette swapping is required.

**Note:** Always write-protect the source diskette before you begin copying, so you won't accidentally overwrite the source diskette.

**Question 12: What is a Startup-sequence and what can I do with it?**

**Answer:** The Startup-sequence is a list of AmigaDOS commands executed when the system is first booted up. You can also run the Startup-sequence while in the Shell:

```
1> execute s/startup-sequence
```

Type this command to see what the Startup-sequence contains:

```
1> type s/startup-sequence
```

You can write your own Startup-sequences with the Shell editor Ed. Type this to access Ed and the Startup-sequence:

```
1> ed s/startup-sequence
```

The Startup-sequence for Workbench Version 2.0 looks like this:

```
version >NIL:
Failat 21
SetClock >NIL: load
```

```

copy >NIL: ENVARC: ram:env all quiet noreq
mkdir ram:t ram:clipboards
assign T: ram:t ;set up T: directory for scripts
if exists sys:Monitorslist >t:mon-start
sys:monitors/~#?.info lformat="run >NIL: %s%s"
execute t:mon-start
endif
assign ENV: ram:env
run >NIL: iprefs >NIL:
wait >NIL: 5
addbuffers >NIL: df0: 15
echo "Amiga Workbench Disk. 2.0 Release Version
$Workbench"
BindDrivers
setenv Workbench $Workbench
setenv Kickstart $Kickstart
resident c:Execute pure add
resident c:List pure add
resident c:Assign pure add
assign CLIPS: ram:clipboards
mount speak:
mount aux:
mount pipe:
path ram: c: sys:utilities sys:rexxc sys:system s:
sys:prefs sys:wbstartup add
if exists sys:tools
path sys:tools add
endif
rexxmast >NIL:
if exists s:user-startup
execute s:user-startup
endif
LoadWB
endcli >NIL:

```

Move the cursor to the line you want to change with the cursor keys. Pressing the **(Esc)** key puts you into extended command mode. Pressing **(Esc)** **(D)** **(←)** deletes the current line. Delete the line:

```
endShell > nil:
```

Move the cursor to the line that says `loadwb`. Press **(←)** to move that line down. Move the cursor to that blank line. Enter this:

```
echo"**** This is my Startup-sequence. ****"
```

Press the **(Esc)** key, **(X)** key and **(←)** key to save your Startup-sequence.

Try out the new sequence:

```
1> execute s/startup-sequence
```

As the sequence executes, your message appears on the screen, and the Amiga drops right into the Shell.



**Note:** The `loadwb` command must be present at the end of the Startup-sequence to load `Intuition`. If you exit the Startup-sequence without `loadwb`, you'll get a blank screen without icons.

**Question 13:** Can the Amiga speak while in the Shell?

**Answer:** Yes. The Shell command for speech is `say`. `say` works similar to a `print` command in BASIC. The only differences are that the text is read through the sound system of the Amiga and `say` does not require quotation marks. Type the following to hear `say`:

```
1> say tobi is a real nice guy!
```

You can change the default speech parameters by including a modifier in the text you want spoken. These modifiers are: `-f` (female), `-m` (male), `-r` (robot), `-n` (natural), `-s#` (speed; # is a number ranging from 40 to 400) and `-p#` (pitch; # is a number ranging from 65 to 320). `say` can speak the contents of a file when you add the modifier `-x filename` to the command. The following example recites the Startup-sequence in a woman's voice with a pitch of 180 and a speed of 180:

```
1>say -f -p180 -s180 -x s/startup-sequence
```

You can also use `say` within the Startup-sequence (see Question 12 for editing instructions). Imagine having your Amiga say hello to you every time you turn it on.

**Question 14:** How can I send a C listing to a printer?

**Answer:** Use the Shell `type` command. Say you have a C listing called `test.c` in drive `df1:`. Enter the following:

```
1> run type df1:test.c to prt: opt n
```

`run` uses the multitasking capabilities of the Amiga. While the printer runs, you can work with another program. The `opt n` option inserts line numbers in the C listing. These are helpful when tracking down errors.

**Question 15:** How do I use the multitasking capabilities of the Amiga in everyday work with the Shell?

**Answer:** Normally the Shell processes one command after the other; there is no option for multitasking. Remember that the Shell itself can't perform more than one task at a time. However, the multitasking

operating system of the Amiga allows you to run several single task AmigaDOS commands at once.

For example, you can simultaneously print the directory of the system diskette, edit a document and have the Amiga speak a sentence. The usual command sequence looks like the following:

```
1> list sys: to prt :
1> ed text
1> say hello user
```

This sequence executes faster if you run multiple commands:

```
1> run list sys: to prt :
1> run ed text
1> say hello user
```

The run command passes the command sequence which follows it to a new Shell. Since the original Shell has no tasks to do, it moves to the next task without waiting for the first one to finish.

There is a limitation: Two Shells shouldn't access the same drive (or a drive and the printer) at the same time. In the case of the disk drives, the two Shells share computing time. This takes the entire operation longer than if the two Shells were executed one after the other.

Another way to initiate several tasks at once is by opening multiple Shells with the newshell command. This gives the user another complete input interface. This method works best when you execute several Shell functions over a long period of time instead of executing Shell commands quickly. The following example makes this clear:

```
1> newShell
1> list df0: quick
2> type files opt h
```

Here a new Shell opens and all of the filenames in the df0: directory appear in this window. Then the file contents of the second and new Shell print out. This way you can read filenames in the first Shell window and work in the second window without disturbing the list of names.

The newshell command also offers several options. The user can set the dimensions of the new Shell window. The syntax looks like this:

```
1> newshell "con:0/10/639/100/My Shell"
```

The word con: refers to the console (keyboard and monitor). The first two numbers specify the x and y coordinates of the upper left corner of the window. The last two numbers set the width and height of the

window. You must enclose the expression in quotation marks if you want to have spaces in the window name.

This lets you place new Shell windows so that they don't hide other windows. If you work with multiple Shells, leave the back and front gadget visible for each window. Clicking a front gadget allows you to bring any of the windows to the foreground.

**Question 16:** What options does the Amiga have for text output?

**Answer:** The `copy` command is the simplest method:

```
1> copy * to prt :
```

See Questions 3 and 8 for more information about the `copy` command.

The built in Shell editor `Ed` can be used for writing letters:

```
1> run ed letter
```

The `Ed` window immediately appears and you can enter your letter.

`Ed` runs independently of your original Shell. You can enter as many documents as you wish. When you complete the letter, press the **(Esc)** **(X)** **(←)** key combination to save it to diskette under the name "letter". You can print your saved file from the Shell by typing:

```
1> type letter to prt :
```

One advantage here over the simple typewriter mode from Question 3 is that the text is on diskette. You can print or edit it at any time by typing:

```
1> run ed letter
```

Enter the following if you want to delete the letter:

```
1> delete letter
```

**Question 17:** How can I make the invisible files on my Workbench diskette visible?

**Answer:** A file doesn't appear in an `Intuition` window unless it has a matching info file. This info file contains the icon data for the corresponding file.

There are many files on the Workbench diskette without info files. These files are invisible to 1.3 users. Workbench 2.0 users can simply

select the Window/Show/All file item to display all files on a diskette. Workbench 1.3 users can adapt these files to appear as icons.

Type in the following to load Ed:

```
1> ed S: show
```

Enter the following text in Ed:

```
.key file/a
.bra (
.ket )
if exists sys:Shell.info
  echo "create info file"
  if exists (file)
    copy sys:system/Shell.info to (file).info
  else
    echo "there is no such source file"
  endif
else
  echo "no .info original found"
endif
quit
```

Now press **[Esc]** **[X]** and **[←]** to save the text. This text is saved under the name "show" in the s: directory.

Now you can assign an info file to any file and make the unseen file visible in a window. By entering:

```
1> execute show NameOfTheFile
```

The execute command activates the command sequence show. The .key command uses NameOfTheFile instead of the word file. The /a option indicates that this argument must be entered.

The .bra and .ket commands define the characters which mark the start and end of the argument placeholders in the command sequence.

The command sequence checks for the existence of the info file "Shell.info", since this info file is used as the source info file. If this file is not found in your directory, you must switch the Shell gadget in Preferences to On (see Question 1, part a).

Sometimes new file icons are piled on top of each other, if they are identical. Separate the icons with the mouse (drag them apart), and use the Workbench option Snapshot to keep them in place.

**Question 18:** How can I combine various documents?

**Answer:** A common operation is combining various separate documents into one. These can be parts of a C listing, or a letter heading, text and closing. Ed cannot merge documents like some word processing

programs can. However, AmigaDOS has the `join` command available through the Shell.

Say you have three text files called `header`, `text` and `closing`. You want to create a single document out of these three parts. This is done with `join`:

```
1> join header text closing as letter
```

The three separate components combine in order and save to diskette under the filename "letter".

**Question 19: How can I search for certain text passages in my files?**

**Answer:** The `search` command locates a specific word or sentence in files. C programmers can use this command to search for procedure and variable names in source listings. Here's the syntax of `search`:

```
1> search name search search_text all
```

`name` = name of the file or disk directory being searched

`search_text` = text to search for

`all` = all available directories are searched

This sequence searches all the files on the diskette in drive `df0`: for the word "tobi."

```
1> search df0: search "tobi" all
```

This command sequence checks the file "letter" for the name "Meier".

```
1> search letter search "Meier"
```

This command searches all of the files starting with the letters "docum" in the current directory for the words "Grand Rapids".

```
1> search docum#? search "Grand Rapids"
```

**Question 20: Can I sort the contents of a text file?****Answer:**

Yes, the `sort` command allows text files of up to 200 lines to be sorted alphabetically. This is especially useful for address lists. For example, if the file "addresses" contains the unsorted addresses of your friends, enter the following:

```
1> sort addresses to sorted
```

This line alphabetically sorts the file and saves the sorted list as a new file named "sorted".

If you want to sort more than 200 lines of text, you must increase the size of the stack with the `stack` command.

---

## 2.2 AmigaDOS commands

This section briefly covers the AmigaDOS commands. First the correct 1.3 syntax of the command appears, then a short description of the command, followed by a description of the arguments. We'll describe the command if it supports additional arguments in Version 2.0. The AmigaDOS commands added to Version 1.3 are marked with the identifier (**AmigaDOS 1.3**). Version 2.0 improvements are also marked. All AmigaDOS 2.0 commands were rewritten in C, which has greatly reduced their size and enhanced their execution speed. Many of the commands were made internal AmigaDOS commands in Version 2.0.

The following qualifiers are used in the command descriptions:

- /A (Argument)** This qualifier always requires a certain argument. The command cannot execute if you omit the argument.
- /K (Key)** The qualifier's name must appear as input (e.g., OPT in the DIR example above) and a keyword must appear as well. The parameters allowed and the functions executed depend on the respective Shell command.
- /S (Switch)** This qualifier needs no arguments. It acts as a switch (toggle) for a command. Switches in commands do just what a wall switch does—switch a command on/off or switch the command to another mode.

Possible qualifiers that can appear in an argument template only in AmigaDOS 2.0:

- /N (Numeric)** This qualifier indicates that a numeric argument is expected (DOS 2.0 only).
- /M (Multiple)** Multiple arguments can be included. Commas were used in 1.3 to signify multiple arguments. You must separate multiple arguments by spaces. This was updated in DOS 2.0. Also the number of arguments is unlimited in DOS 2.0 (DOS 2.0 only).
- /F (Final)** The argument is the final argument. This allows using strings without enclosing them in quotation marks (DOS 2.0 only).
- ,** (comma) The command takes no arguments (DOS 2.0 only).

**ADDBUFFERS DRIVE/A, BUFFERS/S**

Reserves a buffer on a drive with a certain amount of memory.

*DRIVE*           The drive assigned the buffer.  
*BUFFERS*        The size of the buffer to be allocated.

**ALIAS NAME STRING/F** **(Shell only)**

This command can only be used with the Shell. The command assigns a string to a word (See Chapter 6).

*NAME*            The new command word.  
*STRING*         Contains the command that is called with *NAME*.

**V1 . 3**    Command available in AmigaDOS 1.3 Shell.

**V2 . 0**    Command made an AmigaDOS internal command and correct argument template added.

**Ask PROMPT/A**

Asks a question answered with only (Y)es or (N)o: y returns an error code of 5 and n returns no error code.

*PROMPT*        Contains text displayed on the screen. This is usually in the form of a question.

**V2 . 0**    Command made an AmigaDOS internal command.

**ASSIGN NAME, DIR, LIST/S, EXISTS/S, REMOVE/S**

Assigns a logical device to a directory.

*NAME*            The logical device.  
*DIR*             The directory assigned the logical device.  
*LIST*            Lists the assignments of the logical devices.  
*EXISTS*         Searches for *NAME* in the *ASSIGN* list. The error code 5 is returned if *NAME* is not present.  
*REMOVE*        Removes *Name* from the *ASSIGN* list. It's used for development only.



**V2.0 TARGET/M, DISMOUNT/S, DEFER/S, PATH/S, ADD/S, VOL/S, DIRS/S, DEVICES/S**

- TARGET* The *TARGET/M* argument allows you to make multiple assignments to a single device.
- DISMOUNT* The *DISMOUNT/S* argument allows devices and directories to be removed from the assignment list.
- DEFER* The *DEFER/S* argument creates a late-binding assignment. This assignment only takes effect when the assigned object is accessed.
- PATH* The *PATH/S* argument creates a non-binding assignment. It does not take effect until it is referenced and only remains in effect while it is needed.
- ADD* Adds assignment.
- VOL* The *VOL/S* argument will only display information on the current volume assignments.
- DIRS* The *DIRS/S* argument will only display information on the current directory assignments.
- DEVICES* The *DEVICES/S* argument will only display information on the current device assignments.

**AVAIL CHIP, FAST, TOTAL (AmigaDOS 1.3)**

Displays an overview of the present available memory configuration.

- CHIP* Optional, displays total chip memory.
- FAST* Optional, displays total fast memory.
- TOTAL* Optional, displays total available memory.

**V2.0 FLUSH/S**

*FLUSH* Flushes memory areas.

**BINDDRIVERS**

Binds additional device drivers to the system.

**BREAK PROCESS/A, ALL/S, C/S, D/S, E/S, F/S:**

Stops a task in process.

- PROCESS* Process to be broken off.
- All* Sets the break level at C, D, E and F.
- C,D,E,F* Sets break level.

**V2.0 PROCESS**

*PROCESS/A/N* Specified as numeric.

**CD DIR :**

Changes the directory or displays the current directory.

*DIR:* The drive or the directory which should be accessed.

V2.0 Command made an AmigaDOS internal command.

**CHANGETASKPRI PRI/A, PROCESS/K**

Changes the priority of a process started from the Shell.

*PRI* Priority, shown by *Status* command. Contains the new priority (-128 to 127).

*PROCESS* The new priority is assigned to *PROCESS* number. See the *Status* command.

V2.0 **PRI=PRIORITY/A/N, PROCESS/K/N**

*PRIORITY* Specified as numeric and same as *PRI*.

*PROCESS* Specified as numeric.

**COPY FROM, TO/A, ALL/S, QUIET/S, BUF=BUFFER/K, CLONE/S, DATES/S, NOPRO/S, COM/S :**

Creates a copy of files or a directory.

*FROM* The source file.

*TO* The target file.

*ALL* Copies the entire directory.

*QUIET* Displays no output to the screen.

**BUF-BUFFER**

Uses BUF 512K buffers for copying.

*CLONE* Date, Status bits and comments are also copied.

*DATES* Date is also copied.

*NOPRO* The Status bits are reset when copied.

*COM* The comments are also copied.

V2.0 **COPY FROM/A/M, TO/A, ALL/S, QUIET/S, BUF=BUFFER/K/N, CLONE/S, DATES/S, NOPRO/S, COM/S, NOREQ/S :**

*FROM* Multiple files may be copied.

*BUF* Specified as numeric.

*NOREQ* No requesters will be displayed if an error is encountered.

**DATE DATE, TIME, TO=VER/K**

Input or output of date and/or time.

*DATE*           The date to be input.  
*TIME*            The time to be input.  
*To=VER*         The name of the file into which the date or the time is written.

**V2.0 DAY**

*DAY*            Advances date to next day input. Version 2.0 also allow numeric input into the month field.

**DELETE , , , , , , , , , , ALL/S, Q=QUIET/S :**

Erases files and/or directories.

, , , , , , , , , ,       Ten files or directory names to be deleted.  
*ALL*             The entire directory is deleted.  
*Q=QUIET*        There is no message output to the screen.

**V2.0 FILE/M/A, ALL/S, QUIET/S, FORCE/S :**

*FILE*            Multiple files or directory names to be deleted.  
*FORCE*          Forces deletion, even if file is in use.

**DIR DIR, OPT/K, ALL/S, DIRS/S, FILES/S, INTER/S :**

Displays the directory of a disk.

*DIR*            Name of the disk drive or the directory (pathname).  
*OPT*            Allows input of abbreviations, A=ALL, D=DIRS, F=FILES and I=INTER.  
*ALL*            Shows all files in the directory including its subdirectories and their contents.  
*DIRS*          Displays only directories.  
*FILES*         Displays only files.  
*INTER*         The contents are interactively output. After each file or directory the following entries can be made.  
?                Displays the possible commands.  
B                Back up the directory (directory only).  
E                Enter the displayed directory (directory only).  
T                Type the file (files only).  
Del             The file is deleted.  
Q                Quit the Dir command.

**Note:**         When using these arguments (ALL, DIRS, FILES, INTER) do not include the OPT argument.

**DISKCHANGE DEVICE /A**

Tells AmigaDOS that a disk has been changed.

*DEVICE* Which drive has experienced a disk change.

**DISKCOPY** [**FROM**] <disk> **TO** <disk> [**NOVERIFY**] [**MULTI**]  
[**NAME** <name>]

Creates a copy of a disk.

*FROM* <disk>

The source drive.

*TO* <disk> The destination drive.

*NOVERIFY* No verification performed during the copy.

*MULTI* Multiple copies on a single master may be made.

*NAME Name* Names the copy Name.

**DISKDOCTOR DRIVE /A**

Repairs errors on a disk. Damaged files may or may not be removed.

*DRIVE* The drive the program will attempt to recover.

**ECHO** , **NOLINE** / **S** , **FIRST** / **K** , **LEN** / **K** :

Sends a text to the current output path, usually the screen.

, Text that is output to the current output path.

*NoLines* After the output of the given strings, the output doesn't jump to a new line.

*First n* The starting position of the text to be output.

*Len n* The length of the text to be output.

**V 2 . 0** Command made an AmigaDOS internal command and **FIRST** and **LEN** were specified as numeric.

**ED / EDIT**

Used to edit text files. See Section 2.4 for details and Sections 9.1 and 9.2 for the **ED** and **EDIT** quick reference sections.

**ELSE**

Allows alternative conditions in script files (see **IF**).

**V 2 . 0** Command made an AmigaDOS internal command.

**ENDCLI/ENDSHELL**

Exits Shell or Shell window.

**V2.0** Command made an AmigaDOS internal command.

**ENDIF**

Ends an IF/ENDIF construct in a script file (see IF).

**V2.0** Command made an AmigaDOS internal command.

**ENDSKIP**

Script file resumes execution at line following this command during a Skip.

**V2.0** Command made an AmigaDOS internal command.

**EVAL VALUE1/A,OP,VALUE2,TO,LFORMAT/K:**

Evaluates simple expressions.

<i>Value1</i>	Decimal, hex or octal value
<i>OP</i>	math operator: +, -, *, /, mod, &, l, ~,<<, >>,xor,eqv
<i>Value2</i>	Decimal, hex or octal value
<i>TO</i>	Optional
<i>LFORMAT</i>	Specifies output format:
	%Xn hex (n is number of digits)
	%On octal (n is number of digits)
	%N decimal
	%C character

**EXECUTE NAME TEXT**

Executes a script file.

<i>NAME</i>	The name of the script file to execute.
<i>TEXT</i>	The arguments passed to the file.

**FAILAT RCKLIM/N**

Sets the return error code limit or returns the current return error code limit.

<i>RShellM</i>	Contains the size of the new Return error Code LIMit.
----------------	---

**V2.0** Command made an AmigaDOS internal command.

**FAULT** /N, /N, /N, /N, /N, /N, /N, /N, /N, /N: (AmigaDOS 2.0)

Prints information about a specific error.

*N* The valid error number.

V2.0 Command made an AmigaDOS internal command.

**FF -0, -N** (AmigaDOS 1.3)

This command accelerates the text output on the screen. FF was written by C. Heath, used by permission of Microsmiths, Inc®.

-0 FastFont text output is turned on.

-N FastFont text output is turned off (Note: you should enter -N, not a number for N).

V2.0 Implemented internally in AmigaDOS 2.0.

**FILENOTE FILE /A COMMENT /A**

Inserts a comment into a file.

*FILE* Which file will receive the comment.

*COMMENT* The comment of the file.

V2.0 **ALL/S, QUIET/S:**

*ALL* All files will receive the comment.

*QUIET* No text is displayed during command operation.

**FORMAT DRIVE <disk> NAME <Name> [FFS][NOICONS] [QUICK]**

Formats a disk and gives it a name.

*DRIVE* Required to specify drive.

<disk> Location of the drive containing the disk to format.

*NAME* Required to specify Name.

<name> The formatted disk receives the name "Name."

*FFS* The FastFileSystem is used to format.

*NOICONS* Optional (the disk will not have an icon if this option is used).

*QUICK* Only formats root and boot blocks.

**GET/GETENV NAME** (AmigaDOS 1.3)

This command reads the contents of an environment variable.

*NAME* The label of the variable whose contents should be read.

V2.0 Command made an AmigaDOS internal command.

**ICONX**

(AmigaDOS 1.3)

Assigns icon and data to a script file. This lets you access the script file from the Workbench using the mouse (see Chapter 6).

**IF NOT/S, WARN/S, ERROR/S, FAIL/S, EQ/K, GT/K, GE/K, VAL/S, EXISTS/K:**

This allows choices to be made in script files, based upon conditions.

<i>NOT</i>	Logical reversal of a condition.
<i>WARN</i>	Condition is fulfilled when error code is larger than or equal to 5.
<i>ERROR</i>	Condition is fulfilled when error code is larger than or equal to 10.
<i>FAIL</i>	Condition is fulfilled when error code is larger than or equal to 20.
<i>Text1 EQ Text2</i>	Condition fulfilled when Text1 equals Text2.
<i>GT/Val</i>	Greater than and greater than or equal to. Val used for numeric calculations.
<i>Exists Name</i>	Condition fulfilled when file Name is accessible.

**V2.0** Command made an AmigaDOS internal command.

**INFO DEVICE**

Displays information on the screen about connected disk drives.

*Device* Specifies a device.

**INSTALL DRIVE/A, NOBOOT/S, CHECK/S**

Converts a blank formatted disk into a boot disk.

<i>DRIVE</i>	The drive which contains the disk to be installed.
<i>NOBOOT</i>	Makes the disk a non-bootable DOS disk.
<i>CHECK</i>	Checks to see if the disk is bootable and if the standard Amiga boot code is present.

**V2.0 FFS/S**

*FFS* Use the FastFileSystem.

**JOIN .....AS=TO/K**

Joins two or more files together.

..... First of the two files to be joined together.

..... Second of the two files to be joined together.

**AS** The file to which the joined files are written.

**V2.0 FILES/M**

**FILES** Multiple files may be specified.

**LAB Text**

Defines a string as the branch label for a script file.

**Text** The string to be defined as a label.

**V2.0** Command made an AmigaDOS internal command.

**LIST DIR, P=PAT/K, KEYS/S, DATES/S, NODATES/S, TO/K, SUB/K, SINCE/K, UPTO/K, QUICK/S, BLOCK/S, NOHEAD/S, FILES/S, DIRS/S, LFORMAT/K:**

Lists data about files.

**DIR** Displays only information about the file in **DIR**.  
**P=PAT** Displays only the files specified in **Pattern**.  
**KEYS** Displays the number of header blocks of the file or directory.  
**DATES** Displays the date.  
**NODATES** Suppresses the date.  
**TO** Sends the output to the file **Name**.  
**SUB** Displays information about the file whose name is contained in **Text**.  
**SINCE** Displays only the files created since **Date**.  
**UPTO** Displays only the files created before **Date**.  
**QUICK** Displays the filename only.  
**BLOCK** The file size is given in blocks.  
**NOHEAD** The information is suppressed.  
**FILES** Lists only the files.  
**DIRS** Lists only the directories.  
**LFORMAT="Text"**

The option causes the text in **Text** to be displayed. Entering **%s** serves as a place holder for the actual filename. Entering a second **%s** causes the filename to be displayed a second time. Entering three **%s** causes the first one to display the path description of the current file. The next two contain the filename. Entering four **%s** produces the path description for the first and third ones and the filename for the second and fourth.

**V2.0 ALL/S**

**All** Lists ALL files.



**LoadWB -Debug**

Loads the Workbench from the Shell.

*-Debug* AmigaDOS 1.3 adds a hidden menu with the debugging commands *Debug* and *FlushLibs*.

**V2.0 Delay**

*DELAY* The *DELAY* option waits three seconds before continuing.  
*-Debug* was removed as an option.

**LOCK DRIVE/A, ON/S, OFF/S, PASKEY:**

Prevents or allows access to a hard drive partition.

*DRIVE* Contains the protected hard disk partition.  
*ON* Prevents access to the hard drive partition. Access is restored after entering the password (max. 4 characters).  
*OFF* Removes an existing password. This command functions only with Kickstart 1.3.  
*PASKEY* Four character password required for access.

**MAKEDIR DIR/A**

Creates a new directory with the name *Name*.

*DIR* The name of the new directory.

**V2.0 DIR/M**

*DIR* Multiple directories can be created.

**MAKELINK FROM/A, TO/A, HARD/S: (AmigaDOS 2.0)**

Creates a file that points to another file. When the first file is specified, the linked file is called.

*FROM* The name of the original file.  
*TO* The name of the linked file.  
*HARD* Files will not be linked across volumes.

**MOUNT DEVICE/A, FROM/K**

Mounts a device.

*Device* A new device name.  
*From Name* Removes parameters from the file *Name* instead of the *Devs/Mount-list* file.



**PROTECT FILE/A, FLAGS, ADD/S, SUB/S**

Determines the protection bits a file should have.

<i>FILE</i>	The name of the file to protect.
<i>FLAGS</i>	Sets the protection status.
R	The file can be read.
W	The file can be written to.
D	The file is deletable.
E	The file is executable.
	In V1.3 the Hidden (H), Script (S), Pure (P) and Archive (A) bits can be set or reset.
H	Hidden file.
S	The file can be started without execute (script files only).
P	The file can be placed in the Resident list.
A	The file is archived.
	The H and A bits function only with Kickstart 1.3.
+, <i>ADD</i>	Sets the status of the given Status bit.
-, <i>SUB</i>	Removes the status of the status bit.

**V2.0 ALL/S, QUIET/S**

<i>ALL</i>	Multiple files may now be protected.
<i>QUIET</i>	No messages are displayed.

**QUIT RC**

Stops execution of a script file and returns an error code.

RC	Return error Code.
----	--------------------

V2.0 Command made an AmigaDOS internal command and RC specified as numeric.

**RELABEL DRIVE/A, NAME/A**

Changes the name of a disk.

<i>DRIVE</i>	The drive containing the disk to be renamed.
<i>NAME</i>	The new name of the disk.

**REMRAD**

(AmigaDOS 1.3)

This command erases all files from the reset-resistant RAM disk. The ramdrive.device is also removed after the next boot.

**RENAME FROM/A, TO=AS/A**

Renames files.

*FROM* Name of the data which is to be renamed.

*TO=AS* The new name.

**V2.0 FROM/A/M, QUIET/S**

*FROM* Multiple files may now be protected.

*QUIET* No messages are displayed.

**RESIDENT NAME, FILE, REMOVE/S, ADD/S, REPLACE/S, PURE/S, SYSTEM/S: (AmigaDOS 1.3)**

This command erases, replaces or includes a new command in the list of resident commands.

*NAME* The resident name.

*FILE* Contains the command that should be activated in the Resident list.

*REMOVE* Deletes the command from the list.

*ADD* The command is included in the list.

*REPLACE* Replaces an existing command of the same name in the list with the new version of the command.

*PURE* Checks Pure bit of the command to see if it is set.

*SYSTEM* Files added to the system portion of the resident list cannot be removed.

**V2.0** Command made an AmigaDOS internal command and **FORCE** can be used instead of **PURE**.

**RUN COMMAND**

Runs a program as a background process.

*COMMAND*

An AmigaDOS command to run as a background process.

**V2.0** Command made an AmigaDOS internal command.

**SEARCH FROM/A, SEARCH, ALL/S, NONUM/S, QUIET/S, QUICK/S, FILE/S:**

Searches data for a string.

*FROM* The file to be searched.

*SEARCH Text*

The string to be searched for.

*ALL* Searches all directories and subdirectories.  
*NONUM* Displays no line numbers if string is found.  
*QUIET* No output is displayed.  
*QUICK* The output format is more compact.  
*FILE* Searches for the specified file then the character string.

**V2.0 FILE/A/M, QUIET/S**

*FILE* Multiple files may now be searched.  
*QUIET* No messages are displayed.

**SETCLOCK LOAD/S, SAVE/S, RESET/S**

Transfer the system date and time to and from the clock.

*Load* Loads date and time from the internal clock.  
*Save* Saves system date and time to the internal clock.

**V2.0 RESET/S**

*RESET* Resets clock completely.

**SETDATE FILE/A, DATE/A, TIME:**

Inserts a date or time into data.

*FILE* File into which the date and time are inserted.

*DATE* The date assigned to the file.

*TIME* The time assigned to the file.

**V2.0 ALL/S**

*ALL* Multiple files can have their dates set.

**SET/SETENV NAME, STRING/F: (AmigaDOS 1.3)**

Assigns a string to an environment variable.

*NAME* The label of the variable.

*STRING* The character string to be assigned to the variable.

**V2.0 SET**

*SET* Command also accessed by SET.

Command made an AmigaDOS internal command.

**SETPATCH**

Patches ROM in Kickstart, enhancing system software.

**SKIP LABEL, BACK/S :**

Jumps within a script file to a defined label.

*LABEL* Contains the string defined as a label.  
*BACK* Jumps to the start of the script file before searching for the label.

V2.0 Command made an AmigaDOS internal command.

**SORT FROM/A, TO/A, COLSTART/K :**

Alphabetically sorts a file and saves it to another file.

*FROM* The source filename.  
*TO* The new file the sorted data is written to.  
*COLSTART* The line after which the text is sorted.

V2.0 **CASE/S, NUMERIC/S**

*CASE* The sort is case sensitive, uppercase first.  
*NUMERIC* The sort is numeric sensitive, letters first.

**STACK SIZE :**

Changes the stack size or returns the current size.

*SIZE* The stack size in bytes.

V2.0 Command made an AmigaDOS internal command and *SIZE* parameter specified as numeric.

**STATUS**

**PROCESS, FULL/S, TCB/S, Shell=ALL/S, COM=COMMAND/K :**

Outputs information about Shell processes.

*PROCESS* Selects the task number which should be displayed.  
*FULL* Combines the TCB and Shell options.  
*TCB* Displays priority, stack size and global vector size.  
*Shell=ALL* Displays the status of the current command process.  
*Com=COMMAND* Searches for the Shell command COMMAND.

**V2.0** Command made an AmigaDOS internal command and PROCESS parameter specified as numeric.

**TYPE FROM/A, TO/S, OPT/K, HEX/S, NUMBER/S :**

Displays the contents of a file.

*FROM* The source file.  
*TO* The destination file to which Name1 is copied. If a name isn't given the file appears on the screen.  
*OPT* Allows using H and N abbreviation for Hex and Number.  
*NUMBER* The lines are displayed with line numbers.  
*HEX* The characters are displayed in hex and ASCII characters.

**V2.0** Multiple files may be input.

**UNALIAS NAME (AmigaDOS 2.0)**

Removes an alias from the alias list.

*NAME* The name of the alias to remove.

**V2.0** AmigaDOS 2.0 internal command.

**UNSET/UNSETENV NAME: (AmigaDOS 2.0)**

Unsets an environmental variable.

*NAME* The name of the variable to remove.

**V2.0** AmigaDOS 2.0 internal command.

**VERSION NAME, VERSION, REVISION, UNIT:**

Displays the version and revision number of a device, library or Workbench diskette.

*NAME* Library name.  
*VERSION* Set condition flag based on version number.  
*REVISION* Set condition flag based on revision number.  
*UNIT* Specify unit, for multi-unit devices.

**WAIT /N, SEC=SECS/S, MIN=MINS/S, UNTIL/K:**

Shifts the system to a pause mode.

*N* Waiting time in n units.  
*SEC=SECS* Specifies the unit as seconds.  
*MIN=MINS* Specifies the unit as minutes.  
*UNTIL* Waits until the input time.

**WHICH** FILE/A, NORES/S, RES/S: (AmigaDOS 1.3)

This command searches for and displays the path of a command (helps locate the command's location on disk).

*FILE* Name of the command to search for.  
*NORES* Suppress search in resident list.  
*RES* Limits the search to the resident list.

**V2.0** ALL/S

*ALL* You can look for multiple files.

**WHY**

Returns information about the last error that occurred.

**V2.0** Command made an AmigaDOS internal command.



## 2.3 New Startup-sequences

The following startup-sequence allows you to enter the current date on every system start. The Startup-sequence file must be in the `s :` directory on the Workbench diskette to execute. You may wish to add the following sequences to the end of your Startup-sequence. Remember that the Startup-sequence contains important commands for starting up your Amiga. Therefore make sure that you know what a command does before deleting it from the Startup-sequence. It's also a good idea to make a backup copy of the Startup-sequence before changing it or adding commands to it.

```
Echo " "
Echo "Startup-Sequence: ) 1987 by Stefan Maelger"
Echo " "
if exists sys:system
  Path sys:system add
endif
BindDrivers
SetMap d
Date
Echo " "
Echo "Please enter the new date in"
Echo "the displayed format:"
Date ?
Echo " "
Echo "The new date is:"
Date
Echo " "
Info
loadwb
endShell >nil:
```

The sequence below sets the Amiga to tomorrow's date. If you remember to set the date in Preferences before you switch off the Amiga, the date is correct the next time you switch on your Amiga.

```
Echo " "
Echo "Startup-Sequence by Stefan Maelger"

If Exists sys:system
  Path sys:system add
end if
Binddrivers
Setmap d

Date tomorrow
Echo " "
Echo "Today's date is:"
```

```
Date

Echo "System:"
Info

loadwb
endShell >nil:
```

This is the ideal Workbench for Shell enthusiasts. It opens a second **Shell** window and changes the prompt slightly (you'll see how when you try it out).

```
ADDBUFFERS df0:C 20
Echo "This creates a new Shell window and prompt"
Echo " "
If Exists sys:system
Path sys:system add
end if
Binddrivers
PROMPT Shell#%n>
NEWShell
Info

loadwb
endShell >nil:
```

This is the Startup-sequence for the beginner. It closes the big **Shell** window, but opens a smaller **Shell** window. It also shows the RAM disk icon.

```
Echo " "
Echo "Workbench Version 1.2 33.45"
Echo " "
If Exists sys:system
Path sys:system add
end if
Binddrivers
Echo "Welcome everyone"

loadwb
DIR RAM:
NEWShell "CON:0/150/400/50/Alternative"
endShell >nil:
```

---

## 2.4 Using Mount

Users seldom used the `Mount` command in earlier Workbench implementations. To discover more about the command, we must first understand its main purpose. The `Mount` command mounts a new device in the Amiga's operating system. First we should look at the existing devices. This can be done easily with the `Assign` command. If you enter `Assign` without any arguments, your screen may display the following output:

```
Volumes:
Ram Disk [Mounted]
Best T&T [Mounted]
Workbench2.0 [Mounted]

Directories:
CLIPS           Ram Disk:clipboards
ENV             Ram Disk:env
T              Ram Disk:t
ENVARC         Workbench2.0:Prefs/Env-Archive
SYS            Workbench2.0:
C              Workbench2.0:C
S              Workbench2.0:S
LIBS           Workbench2.0:Libs
DEVS           Workbench2.0:Devs
FONTS          Workbench2.0:Fonts
L              Workbench2.0:L

Devices:
PIPE AUX SPEAK RAM CON
RAW SER PAR PRT DF0
DF1
```

Notice the last group (`Devices:`). This tells us the devices available on the Workbench disk.

`DF0:` and `DF1:` should be familiar to you by now. `PRT:` represents the direct printer interface and `PAR:` or `SER:` represent the parallel and serial interfaces. Output can be sent over `RAW:` and `CON:` without access to Intuition. The `RAM:` may be familiar to you as the RAM disk. The `SPEAK:`, `AUX:` and `PIPE:` devices will only be displayed if they have been Mounted. This is usually done in the Startup-sequence.

The devices listed above are placed in the operating system for access at anytime. Whenever you want to address a new device, `Mount` must inform the system of the device's existence. This method makes it easy for improvements to be added to Amiga.

We need an entry in the `Mount` list first, found in the `DEVFS:` directory. The following example creates access to an external drive addressed as `DF1:` (see your `Mount` list for this example or something similar - type `devs:mountlist`):

```
DF1: Device = trackdisk.device
      Unit = 1
      Flags = 1
      Surfaces = 2
      BlocksPerTrack = 11
      Reserved = 2
      PreAlloc = 11
      Interleave = 0
      LowCyl = 0 ; HighCyl = 79
      Buffers = 20
      BufMemType = 3
#
```

Definitions always begin with the new device's name (`DF1:`) and end with the end mark (`#`). Everything between them depends on the respective device. Certain arguments are used frequently:

<i>Disk drives:</i>	<u>Keyword</u>	<u>Function</u>
	<code>Device</code>	Name of the device driver
	<code>Unit</code>	Device number (e.g., 0 for <code>df0:</code> )
	<code>FileSystem</code>	Label of a special <code>FileSystem</code>
	<code>Priority</code>	Task priority (mostly 10)
	<code>Flags</code>	Parameter for Open device (usually 0)
	<code>Surfaces</code>	Number of sides of drive (for disks: 2)
	<code>BlocksPerTrack</code>	Number of blocks per track
	<code>Reserved</code>	Number of boot blocks (usually 2)
	<code>PreAlloc</code>	(no function)
	<code>InterLeave</code>	Device-specific (usually 0)
	<code>LowCyl</code>	Number of small tracks
	<code>HighCyl</code>	Number of large tracks
	<code>Buffers</code>	Size of buffer memory in blocks
	<code>BufMemType</code>	Type of memory: 0,1 = Chip or Fast RAM 2,3 = Only Chip RAM 4,5 = Only Fast RAM
	<code>Mount</code>	1 = Device connected -1 = Device connected on first access

<i>Other devices:</i>	<u>Keyword</u>	<u>Function</u>
	<code>Handler</code>	Path description of the device driver
	<code>Stack</code>	Size of the processor stacks for the task
	<code>Mount</code>	See above

The 2.0 `MOUNT` command reads the keywords described above in addition to the following statements:

<i>Version 2.0:</i>	<u>Keyword</u>	<u>Function</u>
	MaxTransfer	Maximum number of blocks that can be transferred.
	Mask	Address area that can be addressed by the DMA.
	Handler	Path description of the device driver.
	GlobVec	Global vector for the process, 0 sets up a private global vector, -1 is no global vector and if the keyword is absent the shared global vector is used.
	StartUp	A string passed to the file system, handler or device on startup as a BPTR to a BSTR.
	BootPri	Sets boot priority of a device, used with the recoverable RAM disk.
	DOSType	Indicates the filesystem. 0x444F5301 for the FastFileSystem, otherwise 0x444F5300.

---

## 2.4.1 Renaming commands

If you have a PC or PC compatible, you may be having some problems getting accustomed to the Amiga system's DOS commands. For example, instead of entering `A:` (MS-DOS) to change access to the first internal drive, you have to enter `cd DF0:` (AmigaDOS). This becomes especially annoying if you frequently switch between systems, or if you're one of the proud few who own a PC card for your Amiga. In this case it would be best to rename drive names `DF0:`, `DF1:`, etc. to IBM-compatible names.

The AmigaDOS `Assign` command assigns a new name to each disk. Here's an example:

```
Assign B: DF1:
```

Now instead of always having to type `DF1:`, you can just enter `B:`. Now remove the disk from the external drive and insert another disk. The Amiga demands the other disk. `Assign` applies to only the existing directory here, and not the disk drive.

We have a cure for that. Copy the definition for `DF1:` into the `mountlist` because it contains all of the necessary data for a disk drive. Then we change the definition name `DF1:` to `B:`. After saving, enter:

```
Mount B:
```

You can now address drive `DF1:` as `B:`.

You can perform the same change on drive DF0:. You must create a copy of the old entry in the `Mount` list. Change the unit from 1 to 0 so the 0 drive is really addressed. Change the definition name to A:. Finally enter the following:

```
Mount A:
```

The table of the devices is supplied with both new devices, which can be checked with `Assign`:

```
Devices:
```

```
A B NEWCON DF1 DF0
PRT PAR SER RAW CON
RAM
```

## 2.4.2 Less is more

`Mount` can do a great deal more than reassign devices. There are very serious applications with which you can save money and amaze your friends. Next we'll discuss the arguments which accompany `Mount`.

Here's a scenario: You buy a 10-pack of unbranded disks which were on sale for \$10. Unfortunately, they are of inferior quality. The first time you format any of these disks you find that almost all of them have hardware errors on side 1. The formatting stops.

Here's the trick: Enter the `Mount` list and duplicate the definition for `DF1:`. Change this copy definition's name to `SSD:`. Go into the `SSD:` list and change the `Surfaces` argument from 2 to 1. The `SSD:` device formats disk on only one side instead of two sides.

Enter the following in the Shell:

```
Format Drive SSD: Name "1 Surface Test"
```

The formatting seems to go faster because only half the disk is being formatted. We strongly recommend that you read and write this disk using this device only; you will have problems reading single-sided formatted disks using the standard devices (`DF0:`, etc.). You can also access the data only through your own applications designed to read drive `SSD:` (i.e., you cannot access data from these disks using normal applications). The main advantage here is that the disks cannot be copied normally. One final tip: Buy the highest quality disks you can afford and you won't need to do any of this single-sided disk formatting.

The `Mount` command has two other unusual qualities. The first comes into play when you have a disk with more than one side damaged or defective. `Mount` also regulates the beginning disk track and ending

disk track in a formatting process. For example, if you find you have read errors on tracks 0-4, enter the `Mount` list and change the `LowCyl` argument to 5: Formatting begins at track five. Tracks 0-4 remain unformatted, and the rest of the disk formats as normal. The second trick controls the end of the disk: Maybe tracks 71-79 are unreadable. Simply change the `HighCyl` argument to 70 and format as described above.

Experimental formatting may cause incompatibilities between the Workbench and the disk drive. The first problem is the Workbench. When you connect a new external drive and format a disk on it, you'll get a `DF1:NODOS` icon. That's okay, but it still creates the second problem. The `DF1:` drive is no longer addressable whether you insert a disk in the normal Amiga format or not. It responds with "No disk in unit 1" which means that only the new format is accepted. You can also address the new format from the Workbench.

These are some of the interesting applications. When you use a data disk with this format, it is no problem reading it with your own program, but any other programs that shouldn't read the data will not have access, without the correct `Mount` list.

### 2.4.3 Printer spooler

Using a printer spooler with a multitasking computer allows you to work on something else while a file goes to the printer.

The `Shell` has a `RUN` command for executing a new task. You can treat the spooler program as a script file using this command. The procedure is as follows:

Start the `Shell` and enter:

```
ED c:PRINT
```

Now enter the following program:

```
.key filename/a,typ/s           ;take the parameters
;-----
; Printer-Spooler
;-----
;(c) 1987 by Stefan Maelger
;-----

if not exists <filename>       ;check for file

echo "File not found"         ;no?
quit                          ;-then end here
```

```

else                                ;or:

copy <filename> to ram:<filename>
                                   ;copy file to the RAM-Disk

if <typ> eq "DUMP"                  ;Hex-Dump output?

run >nil: type ram:<filename> to prt: opt h
                                   ;-HexDump-Spooling

else                                ;or:

run >nil: type ram:<filename> to prt: opt n
                                   ;-normal Spooling

endif

delete ram:<filename>               ;free memory

endif

echo "printing"                    ;Output message
quit

```

Save the file with **[Esc] [X]**. You can call the routine by entering the following (the DUMP parameter is optional and can be omitted):

```
EXECUTE PRINT filename (DUMP)
```

Since the EXECUTE command takes a while to enter—and can easily be typed in incorrectly—enter the following:

```
run>nil: copy sys:c/EXECUTE to sys:c/DO quiet
```

This creates a command named DO which does the same thing as EXECUTE. For example:

```
DO PRINT filename
```

The ability to put a number of commands into a two-character word is a real time saver. Here's another example of DO:

```
RENAME sys:c/EXECUTE TO sys:c/DO
```



# **3**

# **AmigaBASIC**



## 3. AmigaBASIC



BASIC (Beginner's All-purpose Symbolic Instruction Code) was written when computer programs were assembled by hand using punch cards. Compilers were not good systems for beginners because the programmer had to start over if the programs had errors. Two people at Dartmouth thought about this and developed a "beginner-friendly" language. This language had a command set made of English words and an interpreter instead of a compiler.

BASIC is probably the most used programming language in the world today. BASIC has, over the years, been expanded and improved. An advanced BASIC like AmigaBASIC has the easily learned command words and the advantages of structured programming once found only in compiled languages.

AmigaBASIC was developed by the Microsoft Corporation. Actually, it's closer to a version of Macintosh Microsoft BASIC adapted to the Amiga than an interpreter written specifically for the Amiga.

AmigaBASIC supports the Amiga's windows and menu techniques, but many Amiga-specific features cannot be executed directly from AmigaBASIC. These features, like disk-resident fonts and disk commands, are accessible from the AmigaBASIC LIBRARY command. LIBRARY command demonstrations appear later on in this chapter.

**Note:**

The AmigaBASIC programs in this book show where you should press the  key at the end of a program line. The end of paragraph character `<¶>` means to press . These characters were added because some program lines extend over two lines of text in this book and many of these lines must not be separated.

The BASIC programs listed in this book are available on the companion diskette. For information on ordering the companion diskette, see the order information at the end of the book.

**Workbench  
2.0**

The Workbench 2.0 FD files were not available at the time this book was published, so the following programs have only been tested on Workbench 1.2 and 1.3. You can create the 2.0 `bmap` file when the new 2.0 library FD files are available. The following programs may require minor changes to operate using the 2.0 `bmap` files.

---

## 3.1 Kernel commands

AmigaBASIC allows extremely flexible programming. In addition to the AmigaBASIC commands (such as PRINT, IF/THEN/ELSE, etc.), the interpreter can use new commands if they are organized as machine language routines. This means that you can easily integrate your own commands into the BASIC command set.

Instead of writing new routines, it's easier to access existing machine language routines. The Amiga operating system contains a number of general machine language routines called the *kernel*. Just as a kernel of corn is the basis for a plant, the Amiga kernel is the basis for the operating system.

The operating system can be divided into approximately thirty libraries. These are arranged according to subject. These additional routines require only five of these libraries:

1. `exec.library`  
Responsible for tasks, I/O, general system concerns, memory management.
2. `graphics.library`  
Responsible for displaying text and GELs (graphic elements).
3. `intuition.library`  
Responsible for windows, screens, requesters and alerts.
4. `dos.library`  
Responsible for accessing the Disk Operating System.
5. `diskfont.library`  
Responsible for Amiga fonts stored on diskette.

Each of these libraries is filled with machine language routines for accomplishing these tasks. To use these routines with AmigaBASIC, you need three pieces of information:

1. The interpreter must have a name for every single routine contained in the library. You can assign each machine language routine its own name.
2. The interpreter must convey in which library the corresponding routine can be found. Each library has an *offset table* for this assignment: It begins with offset 6 and jumps in increments of 6. Every machine language routine has its own offset.

3. AmigaBASIC must know which parameter register it needs for the routine. AmigaBASIC uses a total of eight data registers and five address registers:

```

1 = Data register d0
2 = Data register d1
3 = Data register d2
4 = Data register d3
5 = Data register d4
6 = Data register d5
7 = Data register d6
8 = Data register d7

```

```

9 = Address register a0
10 = Address register a1
11 = Address register a2
12 = Address register a3
13 = Address register a4

```

Every library must have a .bmap file. This file contains the necessary information for all commands organized in the library.

You can easily create the necessary .bmap files using the ConvertFd program on the AmigaBASIC (Extras) diskette from Commodore Amiga. See the AboutBmaps AmigaBASIC program, in the BasicDemos folder on the AmigaBASIC diskette, for complete information on creating .bmap files.

Before you continue, you should have the following files available:

```

graphics.bmap
intuition.bmap
exec.bmap
dos.bmap
diskfont.bmap

```

Copy these files to the `libs:` subdirectory of the Workbench diskette. An alternative is to ensure that these files are in the same subdirectory as the AmigaBASIC program using them. The copying procedure goes like this when using the Shell:

```

1> copy graphics.bmap to libs:
1> copy intuition.bmap to libs:
1> copy exec.bmap to libs:
1> copy dos.bmap to libs:
1> copy diskfont.bmap to libs:

```

### **Workbench 2.0**

The Workbench 2.0 FD files were not available at the time this book was published, so the following programs have only been tested on Workbench 1.2 and 1.3. When the new 2.0 library FD files are available, the 2.0 bmap file can be created. The following programs may require minor changes to operate using the 2.0 bmap files.

---

## 3.2 AmigaBASIC graphics

The AmigaBASIC graphic commands are much too complex and exhaustive to describe in this brief section (see *AmigaBASIC Inside and Out* from Abacus for a complete description). The next few pages contain tricks and tips to help you in your graphic programming. We'll spend this section describing the commands in detail.

---

### 3.2.1 Changing drawing modes

The Amiga has four different drawing modes. When you create graphics on the screen, they can be interpreted by the computer in one of four basic ways:

- JAM 1** When you draw a graphic (which also includes the execution of a simple PRINT command), only the drawing color is "jammed" (drawn) into the target area. The color changes at the location of each point drawn and all other points remain untouched (only one color is "jammed" into the target area).
- JAM 2** Two colors are "jammed" (drawn) into the target area. A set point appears in the foreground color (AmigaBASIC color register 1), and an unset point takes on the background color (AmigaBASIC color register 0). The graphic background changes from your actions.
- INVERSEVID** AmigaBASIC color register 0 and color register 1 exchange roles. The result is the familiar screen color inversion.
- COMPLEMENT** This mode works just like JAM 1 except that the set point inverts (complements) instead of filling with AmigaBASIC color register 1. A set point erases and an unset point appears.

These four modes can be mixed with one another, so you can actually have nine combinations.

AmigaBASIC currently has no command to voluntarily change the drawing mode. A command must be borrowed from the internal graphic library. It has the format:

```
SetDrMd (RastPort,Mode)
```

The address for `RastPort` is the pointer to the current window structure stored in `WINDOW(8)`. The AmigaBASIC format looks like this:

SetDrMd(WINDOW(8),Mode)

Here is a set of routines which demonstrate the SetDrMd() command:

```
'#####
'#                                     #
'# Program: Character mode           #
'# Author:   TOB                     #
'# Date:     8-3-87                   #
'# Version:  1.0                      #
'#                                     #
'#####
LIBRARY "T&T2:bmaps/graphics.library"
Shadow "Hello everyone",11
LOCATE 4,8
Outline "OUTLINE: used to emphasize text." ,10
LIBRARY CLOSE
END
SUB Shadow (text$,space%) STATIC
  cX% = POS(0)*8
  cY% = (CSRLIN - 1)*8
  IF cY% < 8 THEN cY% = 8
  CALL SetDrMd(WINDOW(8),0) ' JAM1
  FOR loop% = 1 TO LEN(text$)
    in$ = MID$(text$, loop%,1)
    CALL Move(WINDOW(8),cX%+1,cY%+1)
    COLOR 2,0
    PRINT in$
    CALL Move(WINDOW(8),cX%, cY%)
    COLOR 1,0
    PRINT in$;
    cX% = cX% +space%
  NEXT loop%
  CALL SetDrMd(WINDOW(8),1) ' JAM2
  PRINT
END SUB
SUB Outline (text$, space%) STATIC
  cX% = POS(0)*8
  cY% = (CSRLIN -1) * 8
  IF cY% < 8 THEN cY% = 8
  FOR loop% = 1 TO LEN(text$)
```

```

in$ = MID$(text$, loop%, 1)¶
CALL SetDrMd(WINDOW(8),0) 'JAM1¶
FOR loop1% = -1 TO 1¶
  FOR loop2% = -1 TO 1¶
    CALL Move(WINDOW(8),cX% +loop2%,cY%+loop1%)¶
    PRINT in$;¶
  NEXT loop2%¶
NEXT loop1%¶
CALL SetDrMd(WINDOW(8),2) 'COMPLEMENT¶
CALL Move(WINDOW(8), cX%, cY%)¶
PRINT in$;¶
¶
cX% = cX% + space%¶
NEXT loop%¶
¶
CALL SetDrMd(WINDOW(8),1) 'JAM2¶
PRINT¶
END SUB¶

```

COMPLEMENT mode demonstrates another application: *rubberbanding*. You work with rubberbanding everyday. Every time you change the size of a window, this orange rubberband appears. It helps you find a proper window size.

Intuition normally manages this rubberbanding technique. This technique is quite simple: To prevent the rubberband from changing the screen background, Intuition freezes all screen activities (this is the reason that work stops when you enlarge or reduce a window in a drawing program, for example). The COMPLEMENT drawing mode draws the rubberband on the screen. This erases simply by overwriting, without changing the screen background.

This can be easily programmed in BASIC. The following program illustrates this and uses some interesting AmigaBASIC commands:

```

#####¶
'#                                     #¶
'# Program: Rubberbanding             #¶
'# Author: TOB                         #¶
'# Date: 8-3-87                       #¶
'# Version: 2.0                        #¶
'#                                     #¶
#####¶
¶
LIBRARY "T&T2:bmaps/graphics.library"¶
¶
main:  '* Rubber banding demo¶
      CLS¶
      ¶
      '* rectangle¶
      PRINT "a) Draw a Rectangle"¶
      Rubberband¶
      LINE (m.x,m.y) - (m.s,m.t),,b¶
      ¶

```



```

    '* line
    LOCATE 1,1
    PRINT "b) ...and now a Line!"
    Rubberband
    LINE (m.x,m.y) - (m.s,m.t)
    '
    '* area
    LOCATE 1,1
    PRINT "c) Finally Outline an Area"
    Rubberband
    x = ABS(m.x-m.s)
    y = ABS(m.y-m.t)
    PRINT "width (x)  =";x
    PRINT "Height (y) =";y
    PRINT "Area      =";x*y; "Points."
    '
    LIBRARY CLOSE
    END

'
'
SUB Rubberband STATIC
    SHARED m.x,m.y,m.s,m.t
    CALL SetDRMD(WINDOW(8),2) 'COMPLEMENT
    '
    WHILE MOUSE(0) = 0
        maus = MOUSE(0)
    WEND
    '
    m.x= MOUSE (1)
    m.y = MOUSE(2)
    m.s = m.x
    m.t = m.y
    '
    WHILE maus < 1
        m.a = m.s
        m.b = m.t
        m.s = MOUSE(1)
        m.t = MOUSE(2)
        IF m.a <> m.s OR m.b <> m.t THEN
            LINE (m.x,m.y) - (m.a,m.b) ,,b
            LINE (m.x,m.y) - (m.s,m.t) ,,b
        END IF
        maus = MOUSE(0)
    WEND
    '
    '
    '
    LINE (m.x,m.y) - (m.s,m.t) ,,b
    PSET (m.x,m.y)
    CALL SetDRMD(WINDOW(8), 1)
END SUB

```

### 3.2.2 Changing tpestyles

The Amiga has the ability to modify tpestyles within a program. Tpestyles such as **bold**, underlined and *italic* type can be changed through simple calculations. This is useful to adding class to your text output. Unfortunately, BASIC doesn't support these programmable styles. The SetSoftStyle system function from the graphic library performs this task:

```
SetSoftStyle (WINDOW(8),style,enable)
```

```
style:
  0      = normal
  1      = underline
  2      = bold
  3      = underline and bold
  4      = italic
  5      = underline and italic
  6      = bold and italic
  7      = underline, bold, and italic
```

The following program demonstrates these options:

```
'#####
'#                                     #
'# Program:   Text style               #
'# Author:    TOB                       #
'# Date :     8-12-87                   #
'#                                     #
'#####
¶
DECLARE FUNCTION AskSoftStyle% LIBRARY¶
DECLARE FUNCTION SetSoftStyle% LIBRARY¶
¶
LIBRARY "T&T2:bmaps/graphics.library"¶
¶
var:   'the mode assignments¶
¶
       normal%   = 0¶
       underline% = 1¶
       bold%     = 2¶
       italic%   = 4¶
¶
demo:  ' an example¶
       CLS¶
       Style underline% + italic%¶
       PRINT TAB(20); "This is italic underlined text"¶
       ¶
       LOCATE 5,1¶
```

```

Style normal%¶
PRINT"This is the Amiga's normal text"¶
PRINT"Here are some example styles:"¶
PRINT"a) Normal text"¶
Style underline%¶
PRINT"b) Underlined text"¶
Style bold%¶
PRINT "c) Bold text"¶
Style italic%¶
PRINT "d) Italic text"¶
PRINT¶
Style normal%¶
PRINT "Here are all forms available:"¶
¶
FOR loop% = 0 TO 7¶
  Style loop%¶
  PRINT "Example style number";loop%¶
NEXT loop%¶
¶
' and normal style¶
Style normal%¶
¶
LIBRARY CLOSE¶
END¶
¶
SUB Style (nr%) STATIC¶
  bits% = AskSoftStyle%(WINDOW(8))¶
  news% = SetSoftStyle%(WINDOW(8), nr%, bits%)¶
  ¶
  IF (nr% AND 4) = 4 THEN¶
    CALL SetDrMd(WINDOW(8),0)¶
  ELSE
    ¶
    CALL SetDrMd(WINDOW(8),1)¶
  END IF¶
END SUB¶

```

<b>Variables</b>	bits%	style bits enabling these character styles
	news%	newly set style bits
	nr%	given style bits

**Program description**

The program calls the `Style` SUB command immediately. The `AskSoftStyle&` function returns the style bits of the current font. These bits can later be changed algorithmically. The desired change is made with `SetSoftStyle`, which resets the previously obtained style bits. This function sets the new style when the corresponding mask bits in `bits%` are set. Otherwise, these bits remain unset.

If the italic style is selected in any combination (`nr%` and `4=4`), character mode `JAM 1` is switched on (see Section 3.2.1 above). Italic style uses this mode because `JAM 2` (normal mode) obstructs the characters to the right of the italicized text. If the italic style stays unused, then `SetDrMd()` goes to normal mode (`JAM 2`).

### 3.2.3 Move – cursor control

In some of the previous examples we used the `graphics.library` command `MOVE`. AmigaBASIC can only move the cursor by characters (`LOCATE`), or by pixels in the X-direction (`PTAB`), but it is easy to move the cursor by pixels in both X- and Y-directions with the help of the `MOVE` command.

Call the command in BASIC as follows:

```
Move& (WINDOW (8) ,x%,y%)
```

To simplify things, we have written a command that can be extremely useful:

```
xyPTAB x%,y%
```

**Note:** `graphics.bmap` must be on the diskette.

```
DECLARE FUNCTION Move& LIBRARY¶
¶
LIBRARY "T&T2:bmaps/graphics.library"¶
¶
var:¶
text$="Here we go..."¶
text$=" "+text$+" "¶
empty$=SPACE$(LEN(text$))¶
fontheight%=8¶
¶
main:¶
FOR y%=6 TO 100¶
  xyPTAB x%,y%¶
  PRINT text$¶
  xyPTAB x%,y%-fontheight%¶
  PRINT empty$¶
  x%=x%+1¶
NEXT y%¶
¶
LIBRARY CLOSE¶
END¶
¶
'-----¶
¶
SUB xyPTAB(x%,y%) STATIC¶
  e&=Move& (WINDOW (8) ,x%,y%)¶
END SUB¶
¶
```

<b>Variables</b>	text\$	demo text
	empty\$	empty string, provided for erasing when moving in the y-direction
	fontheight%	font height
	x%,y%	screen coordinates
	e&	Move& command error message

**Program description** The Move& command is declared as a function and the library opens. The demo text moves across the screen in the soft-scroll mode, the library closes and the program ends.

The actual subprogram is extremely simple, since all that happens is the necessary coordinates pass to the Move command.

Although this routine looks simple, it is also very powerful. It can move text in any direction, as in the example, either with the smear effect (SetDrMd mode%=JAM1) or with soft-scrolling (SetDrMd mode%=JAM2).

### 3.2.4 Faster IFF transfer

IFF/ILBM file format is quickly becoming a standard for file structure. IFF format simply means that data can be exchanged between different programs that use the IFF system. Data blocks of different forms can be exchanged (e.g., text, pictures, music). These data blocks are called *chunks*.

You have probably seen many loader programs for ILBM pictures in magazines or even typed in the IFF format video title program from Abacus' *AmigaBASIC Inside and Out*. The long loading time of IFF files is the biggest disadvantage of that format. There are a number of reasons for this delay.

It requires time to identify the different chunks and skip unimportant chunks. Second, there are a number of different ways to store a picture in ILBM format. A graphic with five bitplanes must be saved as line 1 of each bitplane (1-5), line 2 of each bitplane (1-5) and so on. Considering that a bitplane exists in memory as one piece, it takes time to split it up into these elements. Third, programs such as *DeluxePaint II*<sup>®</sup> present another problem: Each line of a bitplane is compressed when a graphic is saved and must be uncompressed when reloading the graphic.

Many professional programs don't use IFF for the reasons stated above. Some programmers don't want graphics compatible with other programs (such as graphics from *Defender of the Crown*<sup>®</sup>). Other programmers prefer to sacrifice that compatibility for speed.

You can add a professional touch to your AmigaBASIC programs with this routine. This program loads an uncompressed IFF-ILBM graphic (you might not want to try this with *DPaint*®) and saves this graphic in the following format:

```
Bitplane 1 (in one piece)
Bitplane 2 ...
...last bitplane
Hardware-color register contents
```

An AmigaBASIC program is generated which loads and displays this graphic after a mouse click. The AmigaBASIC program is an ASCII file, which can be independently MERGED or CHAINED with other programs, and can be started from the Workbench by double-clicking its icon.

The listing below is a fast loader for IFF-ILBM graphics. In-house tests of this loader could call up a graphic in 320 x 200 x 5 format with a loading speed of over 41000 bytes per second (IFF files take a hundred times longer to load).

```
' #####
' #   load pictures like a pro with   #
' #-----#
' #   F A S T - G F X   A m i g a   #
' #-----#
' #           (W) 1987 by Stefan Maelger   #
' #####
'
  DECLARE FUNCTION xOpen& LIBRARY
  DECLARE FUNCTION xRead& LIBRARY
  DECLARE FUNCTION xWrite& LIBRARY
  DECLARE FUNCTION Seek& LIBRARY
  DECLARE FUNCTION AllocMem& LIBRARY
  DECLARE FUNCTION AllocRaster& LIBRARY
'
REM **** OPEN LIBRARIES *****
  LIBRARY "T&T2:bmaps/dos.library"
  LIBRARY "T&T2:bmaps/exec.library"
  LIBRARY "T&T2:bmaps/graphics.library"
'
REM **** ERROR TRAPPING *****
  ON ERROR GOTO errorcheck
'
REM **** INPUT THE FILENAME *****
nameinput:
'

REM **** FREE MEMORY FROM THE BASIC-WINDOW *****
REM **** OPEN NEW WINDOW AND MINISCREEN *****
  WINDOW CLOSE WINDOW(0)
  SCREEN 1,320,31,1,1
  WINDOW 1,"FAST-GFX-CONVERTER",,0,1
```

```

PALETTE 0,0,0,0
PALETTE 1,1,0,0
FOR i=1 TO 4
  MENU i,0,0,""
NEXT
PRINT "IFF-ILBM-Picture:"
LINE INPUT filename$
PRINT "Fast-GFX-Picture:"
LINE INPUT target$
PRINT "Name of the Loader:"
LINE INPUT loader$
CHDIR "df0:"

REM **** OPEN IFF-DATA FILE *****
file$=filename$+CHR$(0)
handle%=xOpen$(SADD(file$),1005)
IF handle%=0 THEN ERROR 255

REM **** CREATE INPUT-BUFFER *****
buffer%=AllocMem$(160,65537)
IF buffer%=0 THEN ERROR 254
colorbuffer%=buffer%+96

REM **** GET AND TEST CHUNK-FORM *****
r%=xRead$(handle%,buffer%,12)
IF PEEKL(buffer%)<>1179603533& THEN ERROR 253
IF PEEKL(buffer%+8)<>1229734477& THEN ERROR 252
bmhdflag%=0
flag%=0

REM **** GET CHUNK NAME + CHUNK LENGTH *****
WHILE flag%<>1
  r%=xRead$(handle%,buffer%,8)
  IF r%<8 THEN flag%=1:GOTO whileend
  length%=PEEKL(buffer%+4)

REM **** BMHD-CHUNK? (CVL("BMHD")) *****
IF PEEKL(buffer%)=1112361028& THEN
  r%=xRead$(handle%,buffer%,length%)

  pwidth%=PEEKW(buffer%)      :REM * PICTUREWIDTH
  pheight%=PEEKW(buffer%+2)   :REM * PICTUREHEIGHT
  pdepth%=PEEK(buffer%+8)     :REM * PICTUREDEPTH
  packed%=PEEK(buffer%+10)    :REM * PACK-STATUS
  swidth%=PEEKW(buffer%+16)   :REM * SCREENWIDTH
  sheight%=PEEKW(buffer%+18)  :REM * SCREENHEIGHT

  bytes%=(pwidth%-1)\8+1
  sbytes%=(swidth%-1)\8+1
  colmax%=2^pdepth%
  IF colmax%>32 THEN colmax%=32
  IF pwidth%<321 THEN mode%=1 ELSE mode%=2
  IF pheight%>256 THEN mode%=mode%+2
  IF pdepth%=6 THEN extraplane%=1 ELSE extraplane%=0

```

```

    ¶
REM **** NEW SCREEN PARAMETERS *****¶
    WINDOW CLOSE 1¶
    SCREEN CLOSE 1¶
    SCREEN 1,pwidth%,pheight%,pdepth%-extraplane%,mode%¶

    WINDOW 1,,,0,1¶
    ¶
REM **** DETERMINE SCREEN-DATA *****¶
    picscreen&=PEEKL(WINDOW(7)+46)¶
    viewport&=picscreen&+44¶
    rastport&=picscreen&+84¶
    colormap&=PEEKL(viewport&+4)¶
    colors&=PEEKL(colormap&+4)¶
    bmap&=PEEKL(rastport&+4)¶
    ¶
REM **** HALFBRIGHT OR HOLD-AND-MODIFY ? *****¶
    IF extraplane%=1 THEN¶
    ¶
REM **** MAKE 6TH BITPLANE *****¶
    plane6&=AllocRaster&(swidth%,sheight%)¶
    IF plane6&=0 THEN ERROR 251¶
    ¶
REM **** AND ADD IT TO THE DATA STRUCTURE *****¶
    POKE bmap&+5,6¶
    POKEL bmap&+28,plane6&¶
    ¶
    END IF¶
    ¶
    bmhdfldflag%=1¶
    ¶
REM **** CMAP-CHUNK (SET EACH COLOR: R,G,B) ***¶
    ELSEIF PEEKL(buffer&)=1129136464& THEN¶
    ¶
    IF (length& OR 1)=1 THEN length&=length&+1¶
    r&=xRead&(handle&,buffer&,length&)¶
    ¶
    FOR i%=0 TO colmax%-1¶
    ¶
REM **** CONVERT TO THE FORM FOR THE ****¶
REM **** THE HARDWARE-REGISTERS ****¶
    POKE colorbuffer&+i%*2,PEEK(buffer&+i%*3)/16¶
    greenblue%=PEEK(buffer&+i%*3+1)¶
    greenblue%=greenblue%+PEEK(buffer&+i%*3+2)/16¶
    POKE colorbuffer&+i%*2+1,greenblue%¶
    ¶
    NEXT¶
    ¶
REM **** CAMG-CHUNK = VIEWMODE (ie. HAM or LACE) ***¶
    ELSEIF PEEKL(buffer&)=1128353095& THEN¶
    ¶
    r&=xRead&(handle&,buffer&,length&)¶
    ¶
    viewmode&=PEEKL(buffer&)¶
REM **** BODY-CHUNK = BITMAPS, LINE FOR LINE *****¶
    ELSEIF PEEKL(buffer&)=1112491097& THEN¶

```



```

┌
REM **** DOES THE SCREEN EXIST AT ALL? *****┐
IF bmhdflag%=0 THEN ERROR 250┐
┌
REM **** IS THIS LINE PACKED? *****┐
IF packed%=1 THEN┐
┌
REM **** THEN UNPACK IT!!! *****┐
FOR y%=0 TO pheight%-1┐
FOR z%=0 TO pdepth%-1┐
ad%=PEEKL(bmap%+8+4*z%)+y%*sbytes%┐
count%=0┐
WHILE count%<bytes%┐
r%=xRead&(handle&,buffer&,1)┐
code%=PEEK(buffer&)┐
IF code%>128 THEN┐
r%=xRead&(handle&,buffer&,1)┐
value%=PEEK(buffer&)┐
endbyte%=count%+257-code%┐
FOR x%=count% TO endbyte%┐
POKE ad%+x%,value%┐
NEXT┐
count%=endbyte%┐
ELSEIF code%<128 THEN┐
r%=xRead&(handle&,ad%+count%,code%+1)┐
count%=count%+code%+1┐
END IF┐
WEND┐
NEXT z%,y%┐
┌
REM **** OR PERHAPS NOT PACKED? *****┐
ELSEIF packed%=0 THEN┐
┌
REM **** FILL IN THE BITMAPS WITH THE DOS-COMMAND READ *┐
FOR y%=0 TO pheight%-1┐
FOR z%=0 TO pdepth%-1┐
ad%=PEEKL(bmap%+8+4*z%)+y%*sbytes%┐
r%=xRead&(handle&,ad%,bytes%)┐
NEXT z%,y%┐
┌
REM **** CODING-METHOD UNKNOWN? *****┐
ELSE┐
┌
ERROR 249┐
┌
END IF┐
┌
ELSE┐
┌
REM **** WE DO NOT HAVE TO BE ABLE TO CHUNK. *****┐
REM **** SHIFT DATA FILE POINTER *****┐
IF (length& OR 1)=1 THEN length&=length&+1┐
now&=Seek&(handle&,length&,0)┐
┌
END IF┐
┌

```

```

REM **** END THE SUBROUTINE *****¶
whileend:¶
¶
WEND¶
¶
REM **** LOAD COLOR AND CLOSE FILE ****¶
IF bmdflag%=0 THEN ERROR 248¶
CALL LoadRGB4(viewport&,colorbuffer&,colmax&)¶
CALL xClose(handle&)¶
¶
REM **** VIEW MODE GOTTEN? THEN ALSO STORE *¶
IF viewmode&<>0 THEN¶
POKEW viewport&+32,viewmode&¶
END IF¶
¶
REM **** OPEN DESTINATION DATA FILE *****¶
file$=target$+CHR$(0)¶
handle&=xOpen&(SADD(file$),1005)¶
IF handle&=0 THEN¶
handle&=xOpen&(SADD(file$),1006)¶
END IF¶
¶
REM *****¶
REM **** SO YOU CAN REMOVE A GRAPHIC *****¶
REM **** FROM MEMORY VERY QUICKLY *****¶
¶
bitmap&=sbytes&*pheight& :REM ONE LARGE BITPLANE¶
¶
FOR i%=0 TO pdepth&-1¶
ad&=PEEKL(PEEKL(WINDOW(8)+4)+8+4*i%)¶
w&=xWrite&(handle&,ad&,bitmap&)¶
NEXT¶
¶
w&=xWrite&(handle&,colorbuffer&,64)¶
¶
REM **** CLOSE DATA FILE, AND FREE BUFFER *****¶
CALL xClose(handle&)¶
CALL FreeMem(buffer&,160)¶
¶
REM *****¶
REM **** GENERATES BASIC-PROGRAM (ASCII-FORMAT) *¶
OPEN loader$ FOR OUTPUT AS 1¶
¶
PRINT#1,"' #####";CHR$(10);¶
PRINT#1,"' # Fast-Gfx Loader #";CHR$(10);¶
PRINT#1,"' #-----#";CHR$(10);¶
PRINT#1,"' # ";CHR$(169);" '87 S. Maelger #";CHR$(10);¶
PRINT#1,"' #####";CHR$(10);¶
PRINT#1,CHR$(10);¶
¶
REM **** DECLARE THE ROM-ROUTINES *****¶
PRINT#1,"DECLARE FUNCTION xOpen& LIBRARY";CHR$(10);¶
PRINT#1,"DECLARE FUNCTION xRead& LIBRARY";CHR$(10);¶
PRINT#1,"DECLARE FUNCTION AllocMem& LIBRARY";CHR$(10);¶
¶
REM **** FOR THE CASE OF H.A.M. OR HALFBRIGHT ****¶

```

```

IF pdepth%=6 THEN¶
¶
  PRINT#1,"DECLARE FUNCTION AllocRaster& LIBRARY";¶
  PRINT#1,CHR$(10);¶
¶
END IF¶
¶
REM **** OPEN NEEDED LIBRARIES *****¶
PRINT#1,CHR$(10);¶
PRINT#1,"LIBRARY ";CHR$(34);"dos.library";CHR$(34);¶
PRINT#1,CHR$(10);¶
PRINT#1,"LIBRARY ";CHR$(34);"exec.library";CHR$(34);¶
PRINT#1,CHR$(10);¶
PRINT#1,"LIBRARY
";CHR$(34);"graphics.library";CHR$(34);¶
PRINT#1,CHR$(10);¶
PRINT#1,CHR$(10);¶
¶
REM **** RESERVE MEMORY FOR PALETTE *****¶
PRINT#1,"b&=AllocMem&(64,65537&);CHR$(10);¶
PRINT#1,"IF b&=0 THEN ERROR 7";CHR$(10);¶
¶
REM **** OPEN PICTURE-DATA FILE *****¶
PRINT#1,"file$=";CHR$(34);target$;CHR$(34);
"+CHR$(0)";¶
PRINT#1,CHR$(10);¶
PRINT#1,"h&=xOpen&(SADD(file$),1005)";CHR$(10);¶
¶
REM **** CREATE SCREEN *****¶
PRINT#1,"WINDOW CLOSE WINDOW(0)";CHR$(10);¶
PRINT#1,"SCREEN 1,";MID$(STR$(swidth%),2);",";¶
PRINT#1,MID$(STR$(pheight%),2);",";¶
PRINT#1,MID$(STR$(pdepth%-extraplane%),2);",";¶
PRINT#1,MID$(STR$(mode%),2);CHR$(10);¶
PRINT#1,"WINDOW 1,,0,1";CHR$(10);¶
PRINT#1,"viewport&=PEEKL(WINDOW(7)+46)+44";CHR$(10);¶
¶
REM **** SET ALL COLORS TO ZERO *****¶
lcm$="CALL LoadRGB4(viewport&,b&,"¶
lcm$=lcm$+MID$(STR$(colmax%),2)+" "+CHR$(10)¶
PRINT#1,lcm$;¶
¶
REM **** IS HAM OR HALFBRIGHT ON, 6 PLANES *****¶
IF pdepth%=6 THEN¶
¶
  PRINT#1,"n&=AllocRaster&(";¶
  PRINT#1,MID$(STR$(swidth%),2);",";¶
  PRINT#1,MID$(STR$(pheight%),2);")";CHR$(10);¶
  PRINT#1,"IF n&=0 THEN ERROR 7";CHR$(10);¶

PRINT#1,"bmap&=PEEKL(PEEKL(WINDOW(7)+46)+88)";CHR$(10);¶
PRINT#1,"POKE bmap&+5,6";CHR$(10);¶
PRINT#1,"POKEL bmap&+28,n&";CHR$(10);¶
PRINT#1,"POKEL viewport&+32,PEEKL(viewport&+32)OR
2^";¶
¶

```

```

REM **** AND SET VIEWMODE *****
IF (viewmode& OR 2^7)=2^7 THEN
  SET HALF BRIGHT-BIT *****
  PRINT#1,"7";
ELSE
  SET HOLD-AND-MODIFY - BIT *****
  PRINT#1,"11";
END IF
PRINT#1,CHR$(10);
END IF
REM **** AND NOW THE MAIN ROUTINE *****
PRINT#1,"FOR i%=0 TO";STR$(pdepth%-1);CHR$(10);
PRINT#1,"
ad%=PEEKL(PEEKL(WINDOW(8)+4)+8+4*i%);CHR$(10);
PRINT#1," r%=xRead&(h&,ad&,";
PRINT#1,MID$(STR$(bitmap&),2);"&";CHR$(10);
PRINT#1,"NEXT";CHR$(10);
REM **** GET PALETTE (ALREADY IN THE RIGHT FORM)
PRINT#1,"r%=xRead&(h&,b&,64)";CHR$(10);
REM **** CLOSE THE FILE AGAIN *****
PRINT#1,"CALL xClose(h&);CHR$(10);
REM **** SET COLOR TABLE *****
PRINT#1,lcm$;
REM **** FREE COLOR BUFFER AGAIN ****
PRINT#1,"CALL FreeMem(b&,64)";CHR$(10);
REM **** CLOSE LIBRARIES AGAIN *****
PRINT#1,"LIBRARY CLOSE";CHR$(10);
REM **** WAIT FOR MOUSE-Click *****
PRINT#1,"WHILE MOUSE(0)<>0:WEND";CHR$(10);
PRINT#1,"WHILE MOUSE(0)=0:WEND";CHR$(10);
REM **** CLOSE SCREEN AND BASIC-WINDOW ****
REM **** TURN WORKBENCH-SCREEN ON AGAIN ****
PRINT#1,"WINDOW CLOSE 1";CHR$(10);
PRINT#1,"SCREEN CLOSE 1";CHR$(10);
PRINT#1,"WINDOW 1,";CHR$(34);"OK";CHR$(34);
PRINT#1,"(0,11)-(310,185),0,-1";
PRINT#1,CHR$(10);CHR$(10);
CLOSE 1
REM **** BACK TO THE WORKBENCH *****
WINDOW CLOSE 1

```

```

SCREEN CLOSE 1
WINDOW 1,,,0,-1
PRINT "Creating Loader-Icon"

REM **** DATA FOR SPECIAL-ICON IMAGE ****
RESTORE icondata

file$=loader$+".info"+CHR$(0)

a$=""
FOR i%=1 TO 486
  READ b$
  a$=a$+CHR$(VAL("&H"+b$))
NEXT

REM **** AND WRITE THE ICON DATA-FILE ****
REM **** TO DISK (MODE=OLDFILE) ****
h%=xOpen$(SADD(file$),1005)
w%=xWrite$(h%,SADD(a$),498)

CALL xClose(h%)

REM **** PERHAPS STILL ANOTHER PICTURE ???
*****
CLS
PRINT "Another Picture (y/n)? >";
pause:
a$=INKEY$
IF a$<>"y" AND a$<>"n" GOTO pause
PRINT UCASE$(a$)
IF a$="y" GOTO nameinput
REM **** WERE DONE... *****
LIBRARY CLOSE
MENU RESET
END

REM **** ERROR-TRAPPING *****
errorcheck:
n%=ERR
IF n%=255 THEN
  PRINT "Picture not found"
  GOTO rerun
ELSEIF n%=254 THEN
  PRINT "Not enough Memory!"
  GOTO rerun
ELSEIF n%=253 OR n%=252 THEN
  PRINT "Not IFF-ILBM-Picture!"
  GOTO rerun
ELSEIF n%=251 THEN
  PRINT "Can Not Open 6th Plane."

```

```

GOTO rerun¶
ELSEIF n%=250 THEN¶
PRINT "Not BMHD-Chunk form BODY!"¶
GOTO rerun¶
ELSEIF n%=249 THEN¶
PRINT "Unknown Crunch-Algorithm."¶
GOTO rerun¶
ELSEIF n%=248 THEN¶
PRINT "No more to view."¶
GOTO rerun¶
¶
ELSE¶
CLOSE¶
CALL xClose(handle&)¶
CALL FreeMem(buffer&,160)¶
LIBRARY CLOSE¶
MENU RESET¶
ON ERROR GOTO 0¶
ERROR n%¶
STOP¶
¶
END IF¶
¶
STOP¶
¶
rerun:¶
¶
IF n%<>255 THEN¶
CALL xClose(handle&)¶
IF n%<>254 THEN CALL FreeMem(buffer&,160)¶
END IF¶
¶
BEEP¶
LIBRARY CLOSE¶
RUN¶
¶
icondata:¶
DATA E3,10,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2E,0,1F,0,5,0,3,0,1¶
DATA 0,1,BD,A0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0¶
DATA 0,0,0,4,0,0,0,0,F2,98,0,0,0,0,0,80,0,0,0,80,0,0,0,0,0,0,0,0,0,0¶
DATA 0,0,0,0,0,0,0,0,0,10,0,0,0,0,0,0,0,0,2E,0,1F,0,2,0,0¶
DATA 2,B1,E0,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3,FF,FF,FF,FF,0¶
DATA 3,0,0,0,3,0,2,0,0,0,1,0,2,0,0,0,1,0,2,7,80,0,1,0¶
DATA 0,2,1,F8,0,1,0,2,0,3F,CO,1,0,2,3,FC,0,1,0,2,0,0¶
DATA 1F,CO,1,0,2,0,1,FE,1,0,2,0,0,1F,F1,0,2,0,0,FF,1¶
DATA 0,3,0,1F,FE,3,0,3,FF,FF,FF,FF,0,0,0,6A,BF,F0,0¶
DATA 0,0,0,7,FE,0,0,0,0,0,FF,80,7F,EF,FF,FD,FF,F8,7F¶
DATA EF,FF,FD,E0,38,7F,EF,FF,FD,FF,F8,0,0,0,0,0,0,0,0,0¶
DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3E,7C,F9,B0,0,0,20,40¶
DATA 80,A0,0,0,3C,4C,FO,40,0,0,20,44,80,A0,0,0,20,7C¶
DATA 81,B0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3,FF,FF,FF,FF,0¶
DATA 4,0,0,0,0,80,4,FF,FF,FF,FC,80,5,FF,FF,FF,FE,80¶
DATA 5,FF,FF,FF,FE,80,5,FF,FF,FF,FE,80,5,FF,FF,FF,FE¶
DATA 80,5,FF,FF,FF,FE,80,5,FF,FF,FF,FE,80,5,FF,FF,FF¶
DATA FE,80,5,FF,FF,FF,FE,80,5,FF,FF,FF,FE,80,5,FF,FF¶
DATA FF,FE,80,4,FF,FF,FF,FC,80,4,0,3,FF,80,80,7,FF¶

```

```

DATA 95,7F,FF,80,1,FF,FF,FF,FE,0,7F,FF,FF,FF,FF,F8
DATA 80,10,0,2,FF,84,80,10,0,2,7F,C4,B0,10,0,2,0,4
DATA 7F,FF,FF,FF,FF,FC,38,0,0,0,0,38,30,0,0,0,0,18,0
DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,C,3A,41
DATA 6D,69,67,61,42,41,53,49,43,0

```

### 3.2.5 IFF brushes as objects

If you own a high-quality paint program like *DeluxePaint*<sup>®</sup>, you can actually use it as an object editor. You can create sprites and bobs with this program.

The program in this section lets you convert any IFF graphic into an object file. The only requirement is that the graphic cannot be too large for an object string.

This graphic object can be activated and moved. Since there are no special techniques used for storing the background, too many bitplanes can cause a flickering effect.

```

' #####
' # Use DPaint as Object-Editor with #
' #-----#
' # BRUSH - TRANSFORMER #
' #-----#
' # (W) 1987 by Stefan Maelger #
' #####
'
CLEAR,30000&
DIM r(31),g(31),b(31)
'
nameinput:
PRINT "Brush-File Name (and Path): ";
LINE INPUT brush$
PRINT
PRINT "Object-Data File (and Path): ";
LINE INPUT objectfile$
PRINT
PRINT "Create Color-Data File? (Y/N) ";
pause:
a$=LEFT$(UCASE$(INKEY$+CHR$(0)),1)
IF a$="N" THEN
PRINT "NO!"
ELSEIF a$="Y" THEN
PRINT "OK."
colorflag%=1
PRINT
PRINT "Color-Data File Name (and Path): ";

```

```

        LINE INPUT colorfile$¶
    ELSE¶
        GOTO pause¶
    END IF¶
    PRINT ¶
¶
    OPEN brush$ FOR INPUT AS 1¶
    a$=INPUT$(4,1)¶
    IF a$<>"FORM" THEN CLOSE 1:RUN¶
    a$=INPUT$(4,1)¶
    a$=INPUT$(4,1)¶
    IF a$<>"ILBM" THEN CLOSE 1:RUN¶
¶
getchunk:¶
    a$=INPUT$(4,1)¶
¶
    IF a$="BMHD" THEN¶
        PRINT "BMHD-Chunk found."¶
        PRINT ¶
        a$=INPUT$(4,1)¶
        bwidth%=ASC(INPUT$(1,1)+CHR$(0))*256¶
        bwidth%=bwidth%+ASC(INPUT$(1,1)+CHR$(0))¶
        PRINT "Image width :";bwidth%;" Pixels"¶
        IF bwidth%>320 THEN¶
            PRINT "It is too wide."¶
            BEEP¶
            CLOSE 1¶
            RUN¶
        END IF¶
        bheight%=ASC(INPUT$(1,1)+CHR$(0))*256¶
        bheight%=bheight%+ASC(INPUT$(1,1)+CHR$(0))¶
        PRINT "Image height :";bheight%;" Pixels"¶
        IF bheight%>200 THEN¶
            PRINT "It is too high."¶
            BEEP¶
            CLOSE 1¶
            RUN¶
        END IF¶
        a$=INPUT$(4,1)¶
        planes%=ASC(INPUT$(1,1))¶
        PRINT "Image Depth :";planes%;" Planes"¶
        IF planes%>5 THEN¶
            PRINT "Too many Planes!"¶
            BEEP¶
            CLOSE 1¶
            RUN¶
        ELSEIF planes%*((bwidth%-1)\16+1)*2*bheight%>32000
    THEN¶
        PRINT "Too many Bytes for the Object-String!"¶
        BEEP¶
        CLOSE 1¶
        RUN¶
    END IF ¶
    a$=INPUT$(1,1)¶
    packed%=ASC(INPUT$(1,1)+CHR$(0))¶
    IF packed%=0 THEN¶

```



```

        PRINT "Pack status: NOT packed."¶
    ELSEIF packed%=1 THEN¶
        PRINT "Pack status: ByteRun1-Algorithm."¶
    ELSE¶
        PRINT "Pack status: Unknown method"¶
        BEEP¶
        CLOSE 1¶
        RUN¶
    END IF¶
    a$=INPUT$(9,1)¶
    Status%=Status%+1¶
    PRINT¶
    PRINT ¶
¶
    ELSEIF a$="CMAP" THEN¶
        PRINT "CMAP-Chunk found."¶
        a$=INPUT$(3,1)¶
        l%=ASC(INPUT$(1,1))¶
        colors%=l%\3¶
        PRINT colors%;"Colors found"¶
        FOR i%=0 TO colors%-1¶
            r(i%)=ASC(INPUT$(1,1)+CHR$(0))/255¶
            g(i%)=ASC(INPUT$(1,1)+CHR$(0))/255¶
            b(i%)=ASC(INPUT$(1,1)+CHR$(0))/255¶
        NEXT¶
        Status%=Status%+2¶
        PRINT ¶
        PRINT ¶
¶
    ELSEIF a$="BODY" THEN¶
        PRINT "BODY-Chunk found."¶
        PRINT ¶
        a$=INPUT$(4,1)¶
        bytes%=(bwidth%-1)\8+1¶
        bmap%=bytes%*bheight%¶
        obj$=STRING$(bytes%*bheight%*planes%,0)¶
        FOR i%=0 TO bheight%-1¶
            PRINT "Getting lines";i%+1¶
            FOR j%=0 TO planes%-1¶
                IF packed%=0 THEN¶
                    FOR k%=1 TO bytes%¶
                        a$=LEFT$(INPUT$(1,1)+CHR$(0),1)¶
                        MID$(obj$,j%*bmap%+i%*bytes%+k%,1)=a$¶
                    NEXT¶
                ELSE¶
                    pointer%=1¶
                    WHILE pointer%<bytes%+1¶
                        a%=ASC(INPUT$(1,1)+CHR$(0))¶
                        IF a%<128 THEN¶
                            FOR k%=pointer% TO pointer%+a%¶
                                a$=LEFT$(INPUT$(1,1)+CHR$(0),1)¶
                                MID$(obj$,j%*bmap%+i%*bytes%+k%,1)=a$¶
                            NEXT¶
                            pointer%=pointer%+a%+1¶
                        ELSEIF a%>128 THEN¶
                            a$=LEFT$(INPUT$(1,1)+CHR$(0),1)¶

```

```

        FOR k%=pointer% TO pointer%+257-a%
            MID$(obj$,j%*bmap%+i%*bytes%+k%,1)=a%
        NEXT
        pointer%=pointer%+256-a%
    END IF
WEND
END IF
NEXT
NEXT
    Status%=Status%+4
¶
ELSE
    PRINT a%;" found."
    a=CVL(INPUT$(4,1))/4
    FOR i%=1 TO a
        a%=INPUT$(4,1)
    NEXT
    GOTO getchunk
¶
END IF
¶
checkstatus:
    IF Status%<7 GOTO getchunk
¶
    CLOSE 1
    PRINT
¶
    PRINT "OK, Creating Object."
    ob$=""
    FOR i%=0 TO 10
        ob%=ob%+CHR$(0)
    NEXT
    ob%=ob%+CHR$(planes%)+CHR$(0)+CHR$(0)
    ob%=ob%+MKI$(bwidth%)+CHR$(0)+CHR$(0)
    ob%=ob%+MKI$(bheight%)+CHR$(0)+CHR$(24)
    ob%=ob%+CHR$(0)+CHR$(3)+CHR$(0)+CHR$(0)
    ob%=ob%+obj$
    PRINT
¶
    PRINT "Create Object-Data File as ";CHR$(34);
    PRINT objectfile$;CHR$(34)
    PRINT
¶
    OPEN objectfile$ FOR OUTPUT AS 2
        PRINT#2,ob%;
    CLOSE 2
    PRINT "Object stored."
¶
    IF colorflag%=1 THEN
        PRINT
        PRINT "Creating Color-Data File:"
        OPEN colorfile$ FOR OUTPUT AS 3
            PRINT#3,CHR$(planes%);
            PRINT " Byte 1 = Number of Bitplanes"
            FOR i%=0 TO 2^planes%-1
                PRINT "Byte";i%*3+2;"= red (";i%;" )*255"
            
```

```

PRINT#3,CHR$(r(i%)*255);¶
PRINT "Byte";i%*3+3;"= green(";i%;" )*255"¶
PRINT#3,CHR$(g(i%)*255);¶
PRINT "Byte";i%*3+4;"= blue (";i%;" )*255"¶
PRINT#3,CHR$(b(i%)*255);¶
NEXT¶
CLOSE 3¶
END IF¶
¶
SCREEN 1,320,200,planes%,1¶
WINDOW 2,,,0,1¶
FOR i%=0 TO 2^planes%-1¶
    PALETTE i%,r(i%),g(i%),b(i%)¶
NEXT¶
¶
OBJECT.SHAPE 1,ob$¶
OBJECT.PLANES 1,2^planes%-1,0¶
¶
FOR i=0 TO 300 STEP .1¶
    OBJECT.X 1,i¶
    OBJECT.Y 1,(i\2)¶
    OBJECT.ON¶
NEXT¶
¶
WINDOW CLOSE 2¶
SCREEN CLOSE 1¶
¶
RUN¶

```

**Variables**

status	status of chunks read
a	help variable
b	array, blue scales of a color
bmap	size of BOB bitplane in bytes
bwidth	width of BOB in pixels
brush	name of IFF-ILBM file
bytes	width of BOB in bytes
colorfile	color filename
colors	number of IFF file colors stored
g	array, green scales of a color
packed	pack status 0=not packed; 1=byterun 1
bheight	height of BOB in pixels
i	loop variable
j	loop variable
k	loop variable
l	loop variable
ob	object string
obj	image string
objectfile	file stored in ob\$
planes	bitplane depth of BOB
pointer	counter variable for bytes read from a line
r	array, red scale of a color

<i>Color file data (optional)</i>	Byte 1=	number of bitplanes in the object
	Byte 2=	red scale of background color * 255
	Byte 3=	green scale of background color * 255
	Byte 4=	blue scale of background color * 255
	Byte 5=	red scale of 1st color * 255
	Byte 6=	green scale of 1st color * 255
	Byte 7=	blue scale of 1st color * 255

*IFF structure* Now a few words about IFF-ILBM-format. A file in this format has several adjacently stored files called chunks. Every chunk has the following design:

1	Chunk name	=	4-byte-long string (e.g., "BODY")
2	Chunk length	=	4-byte integer (i.e., LONG format)
3	Chunk data	=	#chunk-long bytes

The header chunk which begins every IFF file has a similar design:

1	Filetype	=	"FORM" (IFF file header)
2	File length	=	Long value
3	Data type	=	"ILBM" (interleaved bitmaps)

The most important chunks:

<i>BMHD chunk</i>	1	long	=	"BMHD" (bitmap header chunk)
	2	long	=	chunk length
	3	word	=	graphic width in pixels
	4	word	=	graphic height in pixels
	5	word	=	X-position of graphic
	6	word	=	Y-position of graphic
	7	byte	=	number of bitplanes on screen
	8	byte	=	masking
	9	byte	=	crunch type
	10	byte	=	??
	11	word	=	transparent color
	12	byte	=	X-aspect
	13	byte	=	Y-aspect
	14	word	=	screen width in pixels
	15	word	=	screen height in pixels

<i>CMAP chunk</i>	1	long	=	"CMAP" (ColorMap)
	2	long	=	chunk length
	3	byte	=	color 0 red value *255
	4	byte	=	color 0 green value *255
	5	byte	=	color 0 blue value *255
	6	byte	=	color 1 red value *255

<b>CRNG chunk</b> <i>(Deluxe Paint)</i>	1	long	=	"CRNG" (ColorCycle chunk-4 times)
	2	long	=	chunk length
	3	word	=	always 0 (at this time)
	4	word	=	speed
	5	word	=	active/inactive
	6	byte	=	lower color
	7	byte	=	upper color
<b>CCRT chunk</b> <i>(Graphic-raft)</i>	1	long	=	"CCRT" (ColorCycle chunk from Graphicraft)
	2	long	=	chunk length
	3	word	=	direction
	4	byte	=	starting color
	5	byte	=	ending color
	6	long	=	seconds
	7	long	=	microseconds
<b>BODY chunk</b>	1	long	=	"BODY" (Bitmaps)
	2	long	=	chunk length
	3		=	1st line of 1st bitplane (for eventual packing - see BMHD above)
				1st line of 2nd bitplane 1st line of 3rd bitplane 2nd line of 1st bitplane...

**ByteRun1-Crunch Algorithm**

There is never more than one line of a bitplane packed at a time. This packing can occur in line order. The coding consists of one code byte. If this byte has a value larger than 128, then the next byte repeats with a value at least 3 times more (e.g., 129 results in the next byte at 258 more). Since FOR/NEXT loops require a starting value for loop variables, this construct must begin with the value 1, listed as follows:

```
FOR i=startvalue TO startvalue+258-codebyte-1
```

Or as shown above, 257-codebyte. The second coding applies to codebytes less than 128. Here the next codebyte+1 byte is not used. In short, you could say that the first and second coding types use a maximum of 128 bytes. Since the width of a 640\*x screen only requires 80 bytes, then one line of one bitplane only requires one coding.

### 3.2.6 Another floodfill

The Amiga has the ability to execute complicated area filling at a rate of one million pixels per second in any color. The AmigaBASIC PAINT command performs this task. This command has one disadvantage in its current form: It can only fill an area that is bordered by only one predetermined color. This limits anyone who might want to use this in their own applications (e.g., drawing programs). A solution might be to set up parameters with the PAINT command that uses any color for the floodfill border. A routine like this exists in the operating system. Since the graphics library handles it as one of its own routines, the program stays in memory and doesn't disappear when the Workbench reboots.

The routine is called FLOOD and can be called from AmigaBASIC as follows:

```
CALL Flood& (Rastport,Mode,x,y)
```

Here is a SUB routine that uses FLOOD:

```
REM #####
REM # F L O O D F I L L Amiga #
REM #-----#
REM # PAINT until to any #
REM # other color if found #
REM #-----#
REM # (W) 1987 by Stefan Maelger #
REM #####
LIBRARY "T&T2:bmaps/graphics.library"
SCREEN 1,640,255,2,2
WINDOW 2,"FLOODFILL",,0,1
LOCATE 2,2
PRINT "Floodfill-Demo"
CIRCLE (200,80),150,2
CIRCLE (400,80),150,3
FLOODFILL 200,80,1
FLOODFILL 300,80,1
FLOODFILL 400,80,1
LIBRARY CLOSE
LOCATE 4,2
PRINT "PRESS ANY KEY"
```

```

WHILE INKEY$=""
WEND
STOP
SUB FLOODFILL(x%,y%,fcolor%) STATIC
  PSET (0,0),0
  PAINT (0,0),0
  COLOR fcolor
  rastport%=WINDOW(8)
  ToAnyColorMode%=1
  CALL Flood$(rastport%,ToAnyColorMode%,x%,y%)
END SUB

```

Initializing this routine is as simple as calling PAINT.

### 3.2.7 Window manipulation

You already know that windows can do a lot. This section shows you a few extra ideas for working with windows in AmigaBASIC.

#### 3.2.7.1 Borderless BASIC windows

An Amiga expert published a long program listing in a recent magazine. This listing looked up a bitmap address and erased the border bit by bit—it took more than a minute to execute. Here's an easier way to get the same result:

```

' #####
' # BORDERLESS for AmigaBASIC-Windows #
' #-----#
' # (W) 1987 by Stefan Maelger #
' #####
'
LIBRARY "T&T2:bmaps/intuition.library"
CLS
PRINT "Here is a Default Window with a Border-"
PRINT
pause 2
PRINT "And Without a Border (Frame)-"
PRINT
PRINT "Press any Key to Restore Default Window"
'
'
killborder
'

```

```

waitkey¶
remake¶
LIBRARY CLOSE¶
END¶
¶
¶
SUB remake STATIC¶
WINDOW CLOSE 1¶
WINDOW 1¶
END SUB¶
¶
SUB pause(seconds%) STATIC¶
t=TIMER+seconds%¶
WHILE t>TIMER¶
WEND¶
END SUB¶
¶
SUB waitkey STATIC¶
WHILE INKEY$=""¶
WEND¶
END SUB¶
¶
SUB killborder STATIC¶
borderless& =2^11¶
gimmezerozero&=2^10¶
window.base&=WINDOW(7)¶
window.modi&=window.base&+24¶
Mode&=PEEKL(window.modi&)¶
Mode&=Mode& AND(2^26-1-gimmezerozero&)¶
Mode&=Mode& OR borderless&¶
POKEL window.modi&,Mode&¶
CALL RefreshWindowFrame(window.base&)¶
END SUB¶

```

### 3.2.7.2 Gadgets on, gadgets off

This program removes and adds gadgets to windows.

```

' #####
' # GADGETon/off in AmigaBASIC-Windows #¶
' #-----#¶
' # (W) 1987 by Stefan Maelger #¶
' #####
'¶
LIBRARY "T&T2:bmaps/intuition.library"¶
¶
PRINT "Make all the Gadgets disappear!"¶
SaveGadgetPointer GadgetStore&¶
pause 5¶
UnlinkGadgets¶

```



```

    pause 10¶
    PRINT "And now bring them back again."¶
    pause 5¶
    SetGadgets GadgetStore&¶
    LIBRARY CLOSE¶
    WINDOW CLOSE 1¶
    WINDOW 1¶
    END¶
¶
SUB pause(seconds%) STATIC¶
    t=TIMER+seconds*¶
    WHILE t>TIMER¶
        WEND¶
    END SUB¶
¶
SUB SaveGadgetPointer(Pointer&) STATIC¶
    window.base& =WINDOW(7)¶
    gadget.pointer&=window.base&+62¶
    Pointer&=PEEKL(gadget.pointer&)¶
    END SUB¶
¶
SUB UnlinkGadgets STATIC¶
    window.base& =WINDOW(7)¶
    gadget.pointer&=window.base&+62¶
    POKE L gadget.pointer&,0¶
    CALL RefreshWindowFrame(window.base&)¶
    END SUB¶
¶
SUB SetGadgets(Pointer&) STATIC¶
    window.base& =WINDOW(7)¶
    gadget.pointer&=window.base&+62¶
    POKE L gadget.pointer&,Pointer&¶
    CALL RefreshWindowFrame(window.base&)¶
    END SUB¶

```

### 3.2.7.3 DrawBorder

Imagine that you want to draw a border from Intuition. You must first know the structure of the border, and the address of a border structure for the DrawBorder routine to execute. Here's the structure:

1st word	Horizontal spacing from X-coordinate called by the routine (defines only one form and can be drawn in any spacing)
2nd word	Vertical spacing of Y-coordinate
3rd byte	Character color (from BASIC)
4th byte	Background color
5th byte	Character mode (JAM1=0)
6th byte	Number of X/Y coordinate pairs
7th long	Coordinate table address
8th long	Address of next structure or value of 0

The 7th part of the structure needs a coordinate table consisting of words. These words contain the X-coordinate and the Y-coordinate of one pixel. One pixel requires four bytes (two words) of memory.

When you call the routine with the Window Rastport instead of the Border Rastport (WINDOW(8)), you can draw any complex structure you wish in the BASIC window. There is one problem with this: The window's character cursor appears after the last pixel of the last structure. A PRINT command starts output at this position. AmigaBASIC uses the cursor position as the starting place for PRINT. Be careful with your use of the PRINT statement after calling DrawBorder.

```
' #####
' # DRAWBORDER - The Border Drawer #
' # (W) 1987 by Stefan Maelger #
' #####
'
LIBRARY "T&T2:bmaps/intuition.library"

PRINT "Putting the Coordinate-String Together"

bwidth%=PEEKW(WINDOW(7)+8)-1
bheight%=PEEKW(WINDOW(7)+10)-1
xleft%=0
ytop%=0
xy$=MKI$(xleft%)+MKI$(ytop%)
xy$=xy$+MKI$(xleft%)+MKI$(bheight%)
xy$=xy$+MKI$(bwidth%)+MKI$(bheight%)
xy$=xy$+MKI$(bwidth%)+MKI$(ytop%)
Pairs%=4
xOffset%=0
yOffset%=0
bcolor%=0

PRINT "Draw the border"

Setborder xy$,Pairs%,bcolor%,xOffset%,yOffset%

FOR i%=3 TO 1 STEP -1
  PRINT "Wait for a few seconds"
  t=TIMER+10:WHILE t>TIMER:WEND
  PRINT "Drawing in Color";i%
  Setborder xy$,Pairs%,i%,xOffset%,yOffset%
NEXT

LIBRARY CLOSE
END

SUB Setborder(xy$,number%,bcolor%,x%,y%) STATIC
  window.base&=WINDOW(7)
  borderrastport&=PEEKL(window.base&+58)
  IF borderrastport&=0 THEN EXIT SUB
```

```

a$=MKI$(0)           'Horizontal Distance¶
a$=a$+MKI$(0)       'Vertical Distance¶
a$=a$+CHR$(bcolor%) 'Drawing Color¶
a$=a$+CHR$(0)       'Background (unused)¶
a$=a$+CHR$(0)       'Mode: JAM1¶
a$=a$+CHR$(number%) 'Number of x-y-Pairs¶
a$=a$+MKL$(SADD(xy$)) 'Pointer to Coordinate¶
a$=a$+MKL$(0)       'Pointer to Next Structure¶
CALL DrawBorder(borderrastport%,SADD(a$),x%,y%)¶
' --Last Parameters are relative X- and Y-Coordinates¶
END SUB¶

```

### 3.2.7.4 ChangeBorderColor

The next routine can change a window's border color, including the title bar. The entire process occurs in the form of a SUB command.

```

' #####¶
' # CHANGE BORDER COLOR #¶
' #-----#¶
' # (W) 1987 by Stefan Maelger #¶
' #####¶
'¶
LIBRARY "T&T2:bmaps/intuition.library"¶
¶
PRINT "Have you ever been disturbed that the"¶
PRINT "drawing color in which borders are always"¶
PRINT "drawn is in color register 0 and that the"¶
PRINT "background is always register 1?"¶
PRINT¶
PRINT "We can change the colors defined"¶
PRINT "in the Window command itself!"¶
¶
LOCATE 10,1:PRINT "Foreground"¶
LOCATE 13,1:PRINT "Background"¶
t=TIMER+15:WHILE t>TIMER:WEND
FOR i=0 TO 3
  LINE (i*30,136)-STEP(30,20),i,bf¶
  LINE (i*30,136)-STEP(30,20),1,b¶
NEXT¶
¶
FOR b%=0 TO 3¶
  FOR f%=0 TO 3¶
    ChangeBorderColor f%,b%¶
    LOCATE 10,14:PRINT f%¶
    LOCATE 13,14:PRINT b%¶
    t=TIMER+5¶
    WHILE t>TIMER¶
      WEND¶
    NEXT f%,b%¶
  NEXT b%

```

```

┌
ChangeBorderColor 1,0┐
┌
LIBRARY CLOSE┐
END┐
┌
SUB ChangeBorderColor(DetailPen%,BlockPen%) STATIC┐
  window.base&=WINDOW(7)┐
  Detail.pen& =window.base&+98┐
  Block.pen&  =window.base&+99┐
  POKE Detail.Pen&,Detail.Pen%┐
  POKE BlockPen&,BlockPen%┐
  CALL RefreshWindowFrame(window.base&)┐
END SUB┐

```

---

### 3.2.7.5 Monocolor Workbench

This program supplies you with an additional 16K of memory by setting up a single bitplane for color on the Workbench. A mono-color Workbench increases the screen editing speed of BASIC programs.

```

' #####
' #           MONOCOLOR WORKBENCH           #
' #-----#
' #           (W) 1987 by Stefan Maelger     #
' #####
'
LIBRARY "T&T2:bmaps/intuition.library"┐
LIBRARY "T&T2:bmaps/graphics.library"┐
┐
Setplanes 1┐
┐
LIBRARY CLOSE┐
SYSTEM┐
┐
SUB Setplanes(planes%) STATIC┐
  IF planes%<1 OR planes%>6 THEN EXIT SUB┐
  rastport&      =WINDOW(8)┐
  bitmaps&       =PEEKL(rastport&+4)┐
  current.planes%=PEEK(bitmaps&+5)┐
  window.base&   =WINDOW(7)┐
  screen.base&   =PEEKL(window.base&+46)┐
  screen.width%  =PEEKW(screen.base&+12)┐
  screen.height% =PEEKW(screen.base&+14)┐
  IF current.planes%>planes% THEN┐
    POKE bitmaps&+5,planes%┐
    FOR kill.plane%=current.planes% TO planes%+1 STEP -1┐
      plane.ad&=PEEKL(bitmaps&+4+4*kill.plane%)┐
      CALL┐
FreeRaster(plane.ad&,screen.width%,screen.height%)┐
  CALL RemakeDisplay┐
  CALL RefreshWindowFrame(WINDOW(7))┐

```

```

        CLS
        NEXT
    END IF
END SUB

```

### 3.2.7.6 PlaneCreator and HAM-Halfbrite

You've seen an example of how `FreeRaster` can free a bitplane from memory. You can also insert other bitplanes, if you know the addresses of these new bitplanes. The programmers of `AmigaBASIC` skipped over support for the Hold-and-Modify (HAM) and Halfbrite modes. These modes require six bitplanes and must be accessed using the `LIBRARY` command (they cannot be used through `AmigaBASIC` commands). Here is a multi-purpose program which lets you switch between modes and insert additional bitplanes.

This program displays all 4096 colors available to `AmigaBASIC` in the `AmigaBASIC` window. Pressing a mouse key displays the 64 colors contained in Halfbrite mode.

```

' #####
' #HAM P L A N E C R E A T O R  HALFBRIGHT #
' #          (W) 1987 by Stefan Maelger #
' #####
DECLARE FUNCTION AllocMem& LIBRARY
LIBRARY "T&T2:bmaps/exec.library"
LIBRARY "T&T2:bmaps/intuition.library"
SCREEN 1,320,200,1,1      :REM *** just ONE Plane
WINDOW 1,"What a wonderful feeling",,,1
PALETTE 0,0,0,0
PALETTE 1,1,1,1
FOR i%=2 TO 6
    CreateNewPlane
    LOCATE 1,1
    PRINT "I have";i%;"Planes";
    FOR j%=1 TO i%
        PRINT "!";
    NEXT
    PRINT
    PRINT "Press left Mouse-Button"
    Wait.for.the.click.of.the.Left.MouseButton
NEXT
HAM
FOR green=0 TO 15
    blue=0
    red=0
    LINE(0,green*10)-STEP(0,9),0
    LINE(1,green*10)-STEP(0,9),green+48
    FOR x=0 TO 7

```

```

FOR red=1 TO 15¶
  LINE(x*32+red+1,green*10)-STEP(0,9),red+32¶
NEXT red¶
blue=blue+1¶
LINE(x*32+17,green*10)-STEP(0,9),blue+16¶
FOR red=14 TO 0 STEP -1¶
  LINE(x*32+17+15-red,green*10)-STEP(0,9),red+32¶
NEXT red¶
blue=blue+1¶
IF blue<16 THEN LINE(x*32+33,green*10)-
STEP(0,9),blue+16¶
NEXT x¶
NEXT green¶
Wait.for.the.click.of.the.Left.MouseButton¶
CLS¶
HB¶
FOR i%=0 TO 3¶
  FOR j%=0 TO 15¶
    LINE(j%*18,i%*45)-STEP(18,45),i%*16+j%,bf¶
    LINE(j%*18,i%*45)-STEP(18,45),1,b¶
  NEXT¶
NEXT¶
Wait.for.the.click.of.the.Left.MouseButton¶
WINDOW 1,"What a wonderful feeling",,-1¶
SCREEN CLOSE 1¶
LIBRARY CLOSE¶
END¶
SUB CreateNewPlane STATIC¶
  bitmap%=PEEKL(WINDOW(7)+46)+184¶
  bitplane%=PEEKW(bitmap%)*PEEKW(bitmap%+2)¶
  wdepth%=PEEK(bitmap%+5)¶
  IF wdepth%>5 THEN EXIT SUB¶
  newplane%=AllocMem$(bitplane%,65538%)¶
  IF newplane%=0 THEN ERROR 7¶
  POKEL bitmap%+8+wdepth%*4,newplane%¶
  POKE bitmap%+5,wdepth%+1¶
  IF wdepth%<5 THEN CALL RemakeDisplay¶
END SUB¶
SUB HAM STATIC¶
  viewmode%=PEEKL(WINDOW(7)+46)+76¶
  POKEW viewmode%,2^11¶
  CALL RemakeDisplay¶
END SUB¶
SUB HB STATIC¶
  viewmode%=PEEKL(WINDOW(7)+46)+76¶
  POKEW viewmode%,2^7¶
  CALL RemakeDisplay¶
END SUB¶
SUB Wait.for.the.click.of.the.Left.MouseButton STATIC¶
  WHILE MOUSE(0)<>0¶
  WEND¶
  WHILE MOUSE(0)=0¶
  WEND¶
END SUB¶

```

You can now draw with colors from 0 to 63. The Amiga normally doesn't support this mode or the setup of the screens. If you want to work in these modes, there are some details you must know.

Let's begin with the Halfbrite mode. Here are a total of 32 colors (0 to 31), spread over the course of 5 planes. The PALETTE command initializes these colors, as well as those for Hold-And-Modify mode. The colors in Halfbrite mode (32 to 63) correspond directly to the colors 0 to 31. Therefore, color number 33 is half as bright as color 1 ( $33-32=1$ ). This equation applies to the other colors as well. You should be careful about the color selection with the PALETTE command. The following calculation returns the RGB proportions of Halfbrite colors:

$$\text{Proportion}(x) = \text{INT}(\text{Proportion}(x-32) * 15/2) / 15$$

This equation uses INT with the slashes ( $x/y$  is the same as  $\text{INT}(x/y)$  here). A PALETTE command for Halfbrite colors would look like this:

```
PALETTE 1,15/15,12/15,11/15
```

The command above assigns color 33 the values 7/15, 6/15, 5/15. Now try assigning the values 14/15, 13/15, 10/15 to another color—it should be another color altogether, but the result is two equal halfbrite colors. Just one reminder: PALETTE doesn't allow colors over 31.

HAM poses even more problems. Colors 0-15 are usable here. When you set a pixel in one of these colors, a point always appears in this color.

Colors 16-31 are another matter. First the RGB value of the pixel is set to the left of the pixel to be drawn (Hold), and then the blue proportion is changed (Modify). The equation for setting the new blue portion is:

$$\text{new\_blue\_portion} = (\text{color}-16) / 15$$

Colors 32-47 change the red portion:

$$\text{new\_red\_portion} = (\text{color}-32) / 15$$

Colors 48-63 modify the green portion of the color:

$$\text{new\_green\_portion} = (\text{color}-48) / 15$$

Now you can set up the desired color using not more than 3 pixels for one "color."

### 3.2.7.7 The coordinate problem

The pixel with the coordinates 0,0 lies below the title bar and to the right of the left border. Most programmers would expect 0,0 to be at the upper left corner of the screen. This can pose problems if you want to place an untitled window directly over the title bar of a standard window (e.g., the **BASIC** window).

What you want is a window eight pixels higher than normal. You could enter the **WINDOW** command as follows:

```
WINDOW 2,,(0,0)-(311,-2),16,-1
```

Although the Y-coordinate moves from 0 to -2, the result is a system error. The first coordinate set (0,0) interprets correctly; the second coordinate pair views the Y-value as false at best, since the interpreter reads the relative coordinates of the standard **BASIC** window. You could also try making a window with the following:

```
WINDOW 2,,(0,0)-(311,8),16,-1
```

This gives you a window 18 pixels high. In this case, you need a window the height of the title bar (10 pixels) to re-establish the screen coordinate system (8-10=-2).

If you only need to cover the title bar of the standard window, you'll need the following coordinate sets:

```
y2=10      height of the new window
y2=y2-10   subtract height of the title bar in proportion to the
            coordinates
y2=y2-4     subtract the top and bottom borders of the new window
```

The result:

```
WINDOW 2,,(0,0)-(311,-4),16,-1
```



---

## 3.3 Fade-in and fade-out

*Fading* is the term used to describe gradual increases or decreases. For example, a *fade-out* is when a song on a record ends by decreasing in volume instead of ending abruptly. A graphic fade-out occurs when a movie scene gradually fades to black. A *fade-in* is the opposite action.

You can create some interesting effects using fading. For example, you can fade text in or out or constantly ("cycle") change graphic colors. One program helps you do all this.

---

### 3.3.1 Basic fading

Like the other programs in this book, these fade programs are simply an example. You can install these routines into your own programs and adapt them to your own uses.

This first program shows the basic idea. It shows you how to change the screen from black to any color on the palette and return this color gradually to black:

```
' Fading-In and Out of colored areas¶
'¶
' (W) by Wgb in June '87¶
'¶
¶
Variables:¶
¶
DEFINT a-z¶
¶
In=1¶
Out=-1¶
Number=7¶
¶
DIM SHARED Red!(Number),Green!(Number),Blue!(Number)¶
¶
MainProgram:¶
¶
GOSUB CreateColorScreen¶
¶
Fading:¶
¶
GOSUB SetColors¶
CALL Fade (0,7,16,In)¶
CALL Fade (0,7,16,Out)¶
¶
```

```

GOTO Fading¶
¶
END¶
¶
SetColors:¶
¶
FOR i=1 TO Number¶
  Red!(i)=RND¶
  Green!(i)=RND¶
  Blue!(i)=RND¶
NEXT i¶
¶
RETURN¶
¶
CreateColorScreen:¶
¶
SCREEN 2,640,256,3,2¶
WINDOW 1,"Color Test", (0,0)-(623,200),0,2¶
¶
FOR i=0 TO Number¶
  PALETTE i,0,0,0¶
NEXT i¶
¶
SWidth=640/Number¶
FOR j=0 TO 20¶
  FOR i=1 TO Number¶
    x=RND*600 ¶
    y=RND*150¶
    LINE (x,y)-(x+SWidth,y+SWidth/2),i,bf¶
  NEXT i¶
NEXT j¶
¶
RETURN¶
¶
SUB Fade (Start,Number,NumSteps,Mode) STATIC¶
¶
  StartState=0 : EndState=NumSteps¶
  IF Mode=-1 THEN¶
    StartState=NumSteps : EndState=0¶
  END IF¶
  FOR j=StartState TO EndState STEP Mode¶
    Factor!=j/NumSteps¶
    FOR i=Start TO Start+Number¶
      PALETTE
i,Red!(i)*Factor!,Green!(i)*Factor!,Blue!(i)*Factor!¶
    NEXT i¶
  NEXT j¶
  ¶
END SUB¶

```

**Arrays**

Blue	blue scale array
Green	green scale array
Red	red scale array

<b>Variables</b>	<b>StartState</b>	starting state of colors
	<b>Number</b>	number of colors (in SUB: number of faded colors)
	<b>SWidth</b>	width of sample area
	<b>EndState</b>	ending state of colors
	<b>Factor</b>	color scale at current time
	<b>In</b>	fadein pointer
	<b>Mode</b>	mode: fade in or fade out
	<b>Out</b>	fadeout pointer
	<b>NumSteps</b>	number of steps for process
	<b>Start</b>	first color number
	<b>i,j</b>	floating variables
	<b>x,y</b>	coordinates for sample field

**Program  
description**

The program defines a function which allows the fading in or fading out of any color on the palette. Combined color groups can be faded as well. First, two variables are set up for the type of fading required. You can only use the variable names once numbers are assigned to them. Next, 7 colors are set as the resolution (e.g., the background). Every color is defined by an array which accesses the individual subroutine. These arrays contain the color values used in the fading process.

The `CreateColorScreen` subroutine opens a new screen for demonstration purposes. It uses the color depths set above. The output window shows colored rectangles.

The main section of the program branches to a subroutine which fills the color arrays with "random" numbers. The main subroutine is then called twice. It gives the number of the first color and the increment needed for fading. Then it indicates whether the fade should be into the desired color or out to black. The ending point determines the individual increments.

Now on to the routine itself. The starting value is set depending upon the pointer setting—either 0 for black, or the value taken from `NumSteps` for "full color" display. The loop used to move through the increments is computed through `Factor` and sets the next color up from black through the `PALETTE` command contained in an inner loop. This loop repeats until either the full brightness or blackness is reached.

### 3.3.2 Fade-over

This is a variation on the above program. Instead of fading to and from black, however, this program fades to and from the starting and ending colors set by you.

```

' Fade-From one Color to Another¶
'¶
' by Wgb in June '87¶
'¶
¶
Variables:¶
¶
DEFINT a-z¶
¶
Number=7¶
¶
DIM SHARED
Red!(Number,1),Green!(Number,1),Blue!(Number,1)¶
¶
MainProgram:¶
¶
GOSUB CreateColorScreen¶
¶
Fading:¶
¶
GOSUB SetColors¶
CALL Fade (0,7,8)¶
¶
GOTO Fading¶
¶
END¶
¶
¶
SetColors:¶
¶
FOR i=1 TO Number¶
  Red!(i,0)=Red!(i,1)¶
  Green!(i,0)=Green!(i,1)¶
  Blue!(i,0)=Blue!(i,1)¶
  Red!(i,1)=RND¶
  Green!(i,1)=RND¶
  Blue!(i,1)=RND¶
NEXT i¶
¶
RETURN¶
¶
CreateColorScreen:¶
¶
SCREEN 2,640,256,3,2¶
WINDOW 1,"Color Test",(0,0)-(623,200),0,2¶
¶
FOR i=0 TO Number¶
  PALETTE i,0,0,0¶
NEXT i¶
¶
SWidth=640/Number¶
FOR j=0 TO 20¶
  FOR i=1 TO Number¶
    x=RND*600 ¶
    y=RND*150¶
    LINE (x,y)-(x+SWidth,y+SWidth/2),i,bf¶
  
```

```

        NEXT i
    NEXT j
    RETURN
SUB Fade (Start,Number,NumSteps) STATIC
FOR j=0 TO NumSteps
    FOR i=Start TO Start+Number
        Rdiff!=(Red!(i,1)-Red!(i,0))/NumSteps*j
        Gdiff!=(Green!(i,1)-Green!(i,0))/NumSteps*j
        Bdiff!=(Blue!(i,1)-Blue!(i,0))/NumSteps*j
        PALETTE
        i,Red!(i,0)+Rdiff!,Green!(i,0)+Gdiff!,Blue!(i,0)+Bdiff!
    NEXT i
NEXT j
END SUB

```

***Program  
description***

This program maintains the basic structure of the earlier fade program, but fine tunes portions of it. The variable definitions no longer require the pointer In and pointer Out for fading to new colors. This is also why the main program call to the fade routine is missing; the program goes to the new color setting for the fade.

The color arrays have an identifier which shows whether the starting color (0) or ending color (1) is set. Reaching the new color value copies the last new value in the starting value register and redefines the ending value. The program can then tell the current status although no reading function exists.

The fading subroutine now goes in any increment of color change. The difference is divided by the step value and multiplied by the number in the already set NumSteps. The result is added to the individual values of the RGB colors. The new color is on the screen when the outermost loop executes.

### 3.3.3 Fading RGB color scales

This last fading option originates from the program in Section 3.3.1. PALETTE commands let you fade RGB colors individually. This means that you can start a screen in red, fade it to green, then end by fading to blue.

```

' Fading-In and Out of Colored Areas
'
' by Wgb in June '87
'

```

```

¶
Variables:¶
¶
  DEFINT a-z¶
¶
  In=1¶
  Out=-1¶
  Number=7¶
¶
  DIM SHARED Red!(Number),Green!(Number),Blue!(Number)¶
¶
MainProgram:¶
  ¶
  GOSUB CreateColorScreen¶
  ¶
  Fading:¶
  ¶
  GOSUB SetColors¶
  CALL Fade (0,7,16,In)¶
  CALL Fade (0,7,16,Out)¶
  ¶
  GOTO Fading¶
  ¶
END¶
  ¶
SetColors:¶
  ¶
  FOR i=1 TO Number¶
    Red!(i)=RND¶
    Green!(i)=RND¶
    Blue!(i)=RND¶
  NEXT i¶
  ¶
RETURN¶
  ¶
CreateColorScreen:¶
  ¶
  SCREEN 2,640,256,3,2¶
  WINDOW 1,"Color Test",(0,0)-(623,200),0,2¶
  ¶
  FOR i=0 TO Number¶
    PALETTE i,0,0,0¶
  NEXT i¶
  ¶
  SWidth=640/Number¶
  FOR j=0 TO 20¶
    FOR i=1 TO Number¶
      x=RND*600 ¶
      y=RND*150¶
      LINE (x,y)-(x+SWidth,y+SWidth/2),i,bf¶
    NEXT i¶
  NEXT j¶
  ¶
RETURN¶
  ¶

```

```

SUB Fade (Start,Number,NumSteps,Mode) STATIC¶
¶
  NumSteps=NumSteps/2¶
  StartState=0 : EndState=NumSteps¶
  IF Mode=-1 THEN¶
    StartState=NumSteps : EndState=0¶
  END IF¶
  StartAt=StartState/NumSteps¶
  EndAt=EndState/NumSteps¶
  FOR j=StartState TO EndState STEP Mode¶
    Factor!=j/NumSteps¶
    FOR i=Start TO Start+Number¶
      PALETTE i,Red!(i)*Factor!,Green!(i)*StartAt,
Blue!(i)*StartAt¶
    NEXT i¶
  NEXT j¶
  FOR j=StartState TO EndState STEP Mode¶
    Factor!=j/NumSteps¶
    FOR i=Start TO Start+Number¶
      PALETTE i,Red!(i)*EndAt,Green!(i)*Factor!,
Blue!(i)*StartAt¶
    NEXT i¶
  NEXT j¶
  FOR j=StartState TO EndState STEP Mode¶
    Factor!=j/NumSteps¶
    FOR i=Start TO Start+Number¶
      PALETTE i,Red!(i)*EndAt,Green!(i)*EndAt,
Blue!(i)*Factor!¶
    NEXT i¶
  NEXT j¶
¶
END SUB¶

```

***Program  
description***

The first section of this listing is identical to the first program up until the subroutine. Use Copy and Paste from the Edit pulldown menu to copy the first section from the program in Section 3.3.1.

First the SUB routine divides the increment number in half. This sets all the programs to about the same "speed setting." Then the same loop executes three times (it executes three times longer). The program looks for the starting value of the fade loop. The mouse pointer is set by this value whether you start with black or with the color.

Since the PALETTE instruction uses all color values, you must set the starting value of the red color scale in the first loop. Then set the other color scales in the other two loops. The other loops bring the program to the end value, as already handled by the red scale. This is computed by the SUB routine at the start under two factors (StartAt and EndAt). All other routines run similar to those in the first fade program.

---

## 3.4 Fast vector graphics

Vector graphics are the displayed outlines of objects on the screen, rather than the complete objects. This speeds up display, since the computation time is minimized for complicated graphics, and the computer is limited to the corner point and the resulting outline.

---

### 3.4.1 Model grids

Working with three-dimensional objects requires storing the corner point as three-dimensional coordinates. First, you must create a compound specification and then combine the coordinate triplets.

Once you have all this data, you must project the space on the screen followed by an area. The following program selects a central spot on the screen plane. All objects here are based upon a single vanishing point perspective.

Since the plane of your screen is set by its Z-coordinate, this value is uninteresting for all points. The grid network comes from this setup.

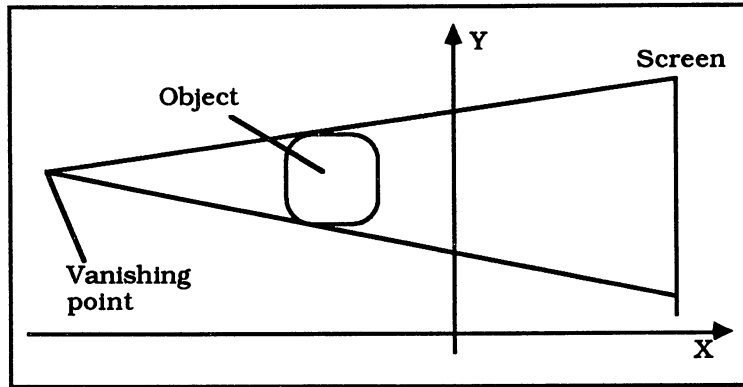
To find the X- and Y-coordinates on the screen, a space must be provided for the 3-D object. Furthermore, this space must have a point set as the vanishing point. The Z-value lies between the object and the vanishing point on the screen plane. Now draw a line from every corner of our object to the vanishing point. When you intersect these lines with the screen plane, you'll find the desired X- and Y-values for these corner points, and their positions on the screen.

The illustration on the next page shows a cross section of the Y- and Z-coordinates.

How should you design a program that reproduces the three dimensional grid illustration? The most important factor is setting up the corner point data. You can place this data in `DATA` statements without much trouble. First, however, the corner point coordinates must be on hand in the compound specification, which can also go into `DATA` statements.



*Three-dimensional grid*



When the program identifies all spatial coordinates, it can begin calculating the screen coordinates. The following line formula is used in three-dimensional space computation:

3D Line formula

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} px \\ py \\ pz \end{pmatrix} + 1 * \begin{pmatrix} dx \\ dy \\ dz \end{pmatrix}$$

You must remember the following when using the above formula: The desired screen coordinates are called X and Y. You figured out the Z-coordinate above. The P-coordinate belongs to the point used as part of the multiplication. All that remains is the D-value. This is the difference of individual point coordinate subtracted from the vanishing point (px-vx, py-vy, pz-vz).

```
' 3D Vector-Graphics I¶
'¶
' © 8.5.1987 Wgb¶
' ¶
¶
Variables:¶
¶
  RESTORE CubeData¶
  DEFINT B,C¶
¶
  MaxPoints=25 ' Maximum Number of Object Points¶
  ZCoord=-25 ' Z-Coordinates of Screen¶
  NumPoints=0 ' Number of Object Points¶
  Connections=0 ' Number of Connections¶
¶
  OPTION BASE 1¶
  DIM P(MaxPoints,3) ' Spatial Coordinates¶
  DIM B(MaxPoints,2) ' Screen Coordinates¶
  DIM C(MaxPoints*1.8,2) ' Connecting Instructions¶
  DIM D(3) ' Difference¶
```

```

┌
DIM F(3)           ' Vanishing Point (x,y,z)┐
┌
F(1)=-70          ' Vanishing Point x┐
F(2)=-50          ' y┐
F(3)=240          ' z┐
┌
MainProgram:┌
┌
PRINT "Vanishing Point (x,y,z): ";F(1),"F(2)","F(3)┐
┌
GetPoint:┌
┌
CBase=NumPoints   ' Base for Connections┐
┌
Loop:┌
┌
READ px,py,pz┐
IF px<>255 THEN ┐
  NumPoints=NumPoints+1 ┐
  P(NumPoints,1)=px┐
  P(NumPoints,2)=py*-1┐
  P(NumPoints,3)=pz┐
  GOTO Loop┐
END IF┐
┌
GetConnection:┌
┌
READ v1,v2┐
IF v1<>255 THEN┐
  Connections=Connections+1┐
  C(Connections,1)=CBase+v1┐
  C(Connections,2)=CBase+v2┐
  GOTO GetConnection┐
END IF┐
┌
READ Last┐
IF Last<>0 THEN GOTO GetPoint┐
┌
┌
CalculatePicture:┌
┌
FOR i=1 TO NumPoints┐
  FOR j=1 TO 3┐
    D(j)=F(j)-P(i,j)┐
  NEXT j┐
  lambda=(ZCoord-P(i,3))/D(3)┐
  B(i,1)=P(i,1)+lambda*D(1)┐
  B(i,2)=P(i,2)+lambda*D(2)┐
NEXT i┐
┌
CreatePicture:┌
┌
FOR i=1 TO Connections┐
  x1=B(C(i,1),1)+50┐
  x2=B(C(i,2),1)+50┐

```

```

        y1=B(C(i,1),2)+100
        y2=B(C(i,2),2)+100
        LINE (x1,y1)-(x2,y2)
    NEXT i
END
CubeData:
REM x,y,z
DATA 32, 20, 20
DATA -32, 20, 20
DATA -32,-20, 20
DATA 32,-20, 20
DATA 32, 20,-20
DATA -32, 20,-20
DATA -32,-20,-20
DATA 32,-20,-20
DATA 255,0,0
REM p1,p2
DATA 1,2
DATA 2,3
DATA 3,4
DATA 4,1
DATA 1,5
DATA 5,6
DATA 6,7
DATA 7,8
DATA 8,5
DATA 4,8
DATA 3,7
DATA 2,6
DATA 255,0,1
PyramidData:
DATA -32, 25,-20
DATA 32, 25,-20
DATA 32, 25, 20
DATA -32, 25, 20
DATA 0, 65, 0
DATA 255,0,0
DATA 1,2
DATA 2,3
DATA 3,4
DATA 4,1
DATA 5,1
DATA 5,2
DATA 5,3
DATA 5,4
DATA 255,0,0

```

<b>Arrays</b>	P ( )	spatial coordinates
	B ( )	int, screen coordinates
	D ( )	differences from the illustration
	F ( )	vanishing point coordinates
	C ( )	int, connection specifications for all objects
<b>Variables</b>	Last	value read, equals 0 when program ends
	CBase	object connection identifier
	NumPoints	number of points to be drawn
	MaxPoints	maximum number of object points
	Connections	number of connections
	ZCoord	Z-coordinate of screen plane
	i,j	floating variables
	lambda	coordinate calculation factor
	px,py,pz	coordinates of one point in space
	v1	first point of a connection
	v2	second point of a connection
	x1,y1	screen coordinates for output (1st point)
x2,y2	screen coordinates for connection (2nd point)	

**Program description**

First, the variable definition sets the DATA pointer to the beginning of the pixel data. In this particular case, the coordinates are a cube. Then all variables starting with B or C are set up as integers. You'll see why soon. Since the arrays for the points are dimensioned later, the program sets the maximum number of points to be stored in the MaxPoints variable. Also, the screen plane's position in space appears through the Z-coordinate. Then the number of points and connections to be read are set to null.

Now follow the dimensioning of necessary variable arrays. These are the P array, into which the point coordinates are stored (an index of 3), then the B array which holds the later screen coordinates for every spatial point. Also, the C array always contains two point numbers which indicate which points should be connected with one another. The last array, D, shows the differences between point computations.

The F array contains the vanishing point position, holding an index for automatic computations (Fpx,Fpy,Fpz).

The next line displays the vanishing point coordinates. Then the point reading routine follows. This routine first sets the CBase pointer to the first number of the point to be read. It works with several objects, so all you need is to enter a coordinate for the first point of the next object later. The loop reads spatial coordinates and checks these coordinates for a px value of 255. This marker reads all the points of an object. The connection specification follows next. If not, new points are entered into the table and new coordinates are read.

The loop for reading connections works in much the same way. It reads the number of points to be connected. Then the loop ends. Otherwise, the two numbers are entered in the array. Finally, a number is read from

the data that indicates whether another object follows. This occurs when the value does not equal zero.

At the conclusion of both loops, the program computes the screen points of the objects. This occurs in a loop which goes through the list point by point and computes all screen values.

Once the difference between the vanishing point value and the current point goes in the D array, the program computes the lambda factor. Next, the program sets the equations up for the X- and Y-values.

The grid display follows. A loop executes for setting up all connections, and sets up all the necessary point coordinates. A previously set point cannot exchange connections and therefore you cannot use them. Since the object next to the null point was defined, you must move the screen center to make the object visible. This redraws it line by line.

---

### 3.4.2 Moving grid models

Movement is just a shifting of a standing screen. You can program the display and easily change the spatial coordinates of any graphic. Unfortunately, the movement is far too slow for practical use.

For faster movement on the screen, you must compute all values before the movement. Also, you have to rely on an operating system routine for drawing lines, instead of the multiple `LINE` commands.

---

### 3.4.3 Moving with operating system routines

The developers of the Amiga operating system thought a great deal about applications which would later run on this computer. Vector graphics were probably part of the plan for future expansion. These make real-time graphics possible under certain conditions. This next routine places all points into a list. This routine is the best option for us, although a faster method exists. It lets you draw a grid network. Then you enter the corner point for your spatial coordinates to be projected later on the screen. The corner point moves within the space, while retaining the original corner coordinates. The routine loses little time, since the program computes all movements before the scenes and places these computations into an array.

Now you'll encounter the first problem. The routine waits for a list of screen coordinates connected in a given sequence. There is an advantage

and a disadvantage to this process. Not every coordinate pair is stored and the figure must be designed in such a way that a constant line can be drawn. If not, those sections considered unnecessary are skipped. However, you can draw flat objects with just an endless line.

To adapt this to the operating system, you must change the connection specification. Enter the corners of the object and the number of corners instead of the coordinate pairs.

When the program has this data, it can start its calculations. First the object is moved in space by the screen coordinates. Then the new graphic transfer occurs. This section enters the available screen values in a long list for later use by the operating system.

If the list is complete, the program branches to the display loop. Here all scenes execute and a corresponding pointer points to the data list for the current scene. Then these values transfer to the display routine. The color changes to the background to clear the screen, and the program redraws the object at its new location on the screen. The program branches after displaying all graphics to the beginning of display and restarts the process.

```
' 3D Vector Graphics V¶
'¶
' Faster by using¶
' The PolyDraw Routine¶
'¶
' by Wgb in June '87¶
'¶
¶
LIBRARY "T&T2:bmaps/graphics.library"¶
RESTORE¶
OPTION BASE 1¶
¶
Variables:¶
¶
DEFINT B,C,G¶
¶
READ MaxPoints      ' Number of Object Points¶
READ Connections   ' Number of Connections¶
ZCoord=25          ' Z-Coordinate in Screen Plane¶
Scenes=50           ' Number of Scenes¶
¶
DIM P(MaxPoints,3)      ' Spatial Coordinates¶
DIM B(Scenes,MaxPoints,2) ' Screen Coordinates¶
DIM G(Connections*2*Scenes)¶
DIM C(Connections)     ' Connection Rules¶
DIM D(3)               ' Differences¶
¶
DIM F(3)              ' Vanishing Point (x,y,z)¶
¶
F(1)=-70             ' Vanishing Point x¶
F(2)=-50             ' y ¶
```

```

F(3)=180          ' z
GetPoint:
PRINT "Vanishing Point (x,y,z) : ";F(1),"F(2)","F(3)
GetPoint:
RESTORE PyramidData ' Object
FOR i=1 TO MaxPoints
  READ px,py,pz
  P(i,1)=px
  P(i,2)=py*-1      ' Transfer to other Coordinate
  System
  P(i,3)=pz
NEXT i
GetConnection:
FOR i=1 TO Connections
  READ C(i)
NEXT i
PreCalculatePicture:
FOR sz=1 TO Scenes
  FOR i=1 TO MaxPoints
    FOR j=1 TO 3
      D(j)=F(j)-P(i,j)
    NEXT j
    P(i,3)=P(i,3)+3
    P(i,2)=P(i,2)-2
    P(i,1)=P(i,1)+2
    Lambda=(ZCoord-P(i,3))/D(3)
    B(sz,i,1)=P(i,1)+Lambda*D(1)+200
    B(sz,i,2)=P(i,2)+Lambda*D(2)+200
  NEXT i
NEXT sz
GraphicTransfer:
FOR j=0 TO Scenes-1
  FOR i=1 TO Connections*2 STEP 2
    G(i+j*Connections*2)=B(j+1,C(i/2+.5),1)
    G(i+1+j*Connections*2)=B(j+1,C(i/2+.5),2)
  NEXT i
NEXT j
ConstructScreen:
FOR i=0 TO Scenes-1
  Pointer=Connections*2*i
  FOR j=1 TO 0 STEP -1
    COLOR j
    CALL Move(WINDOW(8),G(1+Pointer),G(2+Pointer))
    CALL PolyDraw(WINDOW(8),Connections-
VARPTR(G(3+Pointer)))
  NEXT j

```

```

NEXT i¶
¶
GOTO ConstructScreen¶
¶
¶
GraphicData:¶
¶
DATA 5,10¶
' MaxPoints,Connections¶
¶
PyramidData:¶
¶
DATA -32, 25,-20¶
DATA 32, 25,-20¶
DATA 32, 25, 20¶
DATA -32, 25, 20¶
DATA 0, 65, 0¶
¶
PointConnections:¶
¶
DATA 2,1,5,4,3,5,2,3,4,1¶
¶
DATA 4,1¶

```

**Arrays**

B ()	screen coordinates
D ()	differences from the illustration
F ()	vanishing point coordinates
G ()	coordinates of all scenes
P ()	spatial coordinates
C ()	connection specifications

**Variables**

Lambda	coordinate calculation factor
Pointer	pointer to coordinate list of one scene
MaxPoints	maximum number of object points
Scenes	number of scenes to be computed
Connections	number of connections
ZCoord	Z-coordinate of screen plane
i,j	floating variables
px,py,pz	spatial coordinates of corner point
sz	loop pointer for scenes

**Program description**

Before the variable definition, the program opens the graphics library. This supplies the graphic routines needed for the grid network. Then all variables beginning with B, C or G are declared as integers allowing the integer variable character to be left off these variables. The grid network display uses the new G array into which all coordinates are stored in their proper sequences. Each set consists of a 2-byte integer for the X-coordinate and a 2-byte integer for the Y-coordinate.

The new features of this program are the point and connection loops. They work from established values placed in DATA statements which



begin the program. The program runs slightly faster if you delete the end marker. The connection array is defined as one dimensional instead of as a string of characters.

After the computation, the data must be converted to a form that the operating system can handle. The `PolyDraw` routine places a table at the X- and Y-values stated as integer values. In addition, the table must list how many elements are used. The table can be fairly long. This table doesn't need a pointer to the end of data. You place the graphic data for all scenes into one array, and move the routine to the address of the first element of the next scene. The next input is the number of corner points required. The rest of the `PolyDraw` program is self-explanatory.

The display occurs in a new loop. It corresponds to the number of scenes executed. This loop first computes the pointer to the first element to be displayed on the grid network. The second loop executes twice. It draws the network, sets the graphic cursor to the starting point and executes your drawing in the `PolyDraw` routine. The second run of the loop sets the floating variables from 1 to 0, and sets the drawing color to constantly "cycle" the background color through the `COLOR` command. The Amiga draws the grid network in the background color, erasing the net. This process repeats as long as there are scenes available for plotting. The display loop exits when no more scenes are available.

---

### 3.4.4 3-D graphics for 3-D glasses

While experimenting with the multiple-point system and random 3-D production, this idea came up for making a graphic you can view with 3-D glasses. You've seen these glasses; one lens is red and the other lens is usually green or sometimes blue.

This program works under the same principle as 3-D movies. Since you have two eyes, you're actually viewing two different graphics. These two graphics appear to merge into one when you look at the screen through 3-D glasses. The red lens blocks red light and shows you every other color. The green lens blocks green light and allows other colors to show through. The problem in most cases is that some colors are combinations of red and green. This means that you cannot view some objects in the way you want them seen through the 3-D glasses. If you use simple colors with 3-D glass viewing, the effect is dramatic.

This 3-D graphic is based on the grid network used in the previous programs. The programming principle circles around having one vanishing point for each eye. Since both eyes are set fairly close to one

another, you must set the vanishing points close together as well. In this case, two graphics are drawn with horizontally shifted vanishing points. One graphic is drawn in red, and the other in green. All overlapping areas appear in brown (the color you get when you combine a red light and green light).

We've integrated the slider from Chapter 4 into this program (see Section 4.1.1). You can change the degrees of red, green and blue to suit your 3-D glasses. You can even change the locations of the vanishing points for an optimal 3-D effect. When you are satisfied with your settings, press a key to see the result. You can use these values in this program or in your own 3-D programming.

```
' 3D Vector Graphics for Red-Green Glasses ¶
'¶
' © 24.5.1987 Wgb¶
' ¶
¶
LIBRARY "T&T2:bmaps/graphics.library"¶
¶
RESTORE CubeData¶
DEFINT B,C¶
OPTION BASE 1¶
¶
Variables:¶
¶
MaxPoints=25 ' Maximum Number of Object Points¶
ZCoord=-25 ' Z-coordinates of Screen Plane¶
NumPoints=0 ' Number of Object Points¶
Connections=0 ' Number of Connections¶
¶
NumClicks=0¶
MaxClicks=20¶
¶
DIM SHARED ClickTable(MaxClicks,4)¶
DIM SHARED ClickValue(MaxClicks)¶
DIM SHARED ClickID(MaxClicks)¶
¶
DIM P(MaxPoints,3) ' Spatial Coordinates¶
DIM B(2,MaxPoints,2) ' Screen Coordinates¶
DIM C(MaxPoints*1.8,2) ' Connection Rules¶
DIM D(3) ' Difference¶
DIM F(2,3) ' Vanishing Point (x,y,z)¶
¶
F(1,1)=-40 ' 1st Vanishing Point x¶
F(1,2)=-50 ' y¶
F(1,3)=240 ' z¶
¶
F(2,1)=-80 ' 2nd Vanishing Point x¶
F(2,2)=-50 ' y¶
F(2,3)=240 ' z¶
¶
DisplayText:¶
¶
```

```

CLS
LOCATE 1,40
PRINT "Vanishing Point 1 (x,y,z) : "
LOCATE 2,40
PRINT "Vanishing Point 2 (x,y,z) : "
GOSUB DisplayCoordinates

SetColors:
PALETTE 0,.6,.55,.4 ' Background = bright-beige
PALETTE 1,.4,.35,0 ' Neutral Color = Dark Brown
PALETTE 2,.7,0,0 ' Red 70%
PALETTE 3,0,.65,0 ' Green 65%

SliderControl:

Text$="Red"
DefMove 40!,8!,100!,70!,2!
Text$="Green"
DefMove 45!,8!,100!,65!,2!
Text$="Brown"
DefMove 50!,8!,100!,40!,2!

Text$="VPoint1"
DefMove 60!,8!,100!,40!,2!
Text$="VPoint2"
DefMove 65!,8!,100!,80!,2!

GetPoint:
CBase=NumPoints ' Base for Connections

Loop:
READ px,py,pz
IF px<>255 THEN
    NumPoints=NumPoints+1
    P(NumPoints,1)=px
    P(NumPoints,2)=py*-1
    P(NumPoints,3)=pz
    GOTO Loop
END IF

GetConnections:
READ v1,v2
IF v1<>255 THEN
    Connections=Connections+1
    C(Connections,1)=CBase+v1
    C(Connections,2)=CBase+v2
    GOTO GetConnections
END IF

READ Last
IF Last<>0 THEN GOTO GetPoint

CalculateScreen:
FOR k=1 TO 2 ' 2 Vanishing Points

```

```

FOR i=1 TO NumPoints ' All Points¶
  FOR j=1 TO 3      ' Difference for x,y,z¶
    D(j)=F(k,j)-P(i,j)¶
  NEXT j¶
  lambda=(ZCoord-P(i,3))/D(3)¶
  B(k,i,1)=P(i,1)+lambda*D(1)¶
  B(k,i,2)=P(i,2)+lambda*D(2)¶
NEXT i¶
NEXT k¶
¶
DrawScreen:¶
LINE (0,0)-(300,200),0,bf ' Clear Area¶
FOR j=1 TO 2¶
  COLOR 1+j¶
  IF j=2 THEN CALL SetDrMd&(WINDOW(8),7)¶
  FOR i=1 TO Connections¶
    x1=B(j,C(i,1),1)+100¶
    x2=B(j,C(i,2),1)+100¶
    y1=B(j,C(i,1),2)+70¶
    y2=B(j,C(i,2),2)+70 ¶
    LINE (x1,y1)-(x2,y2)¶
  NEXT i¶
NEXT j¶
¶
CALL SetDrMd&(WINDOW(8),1)¶
COLOR 1¶
¶
Interrupt:¶
¶
ON MOUSE GOSUB CheckTable¶
ON TIMER (.5) GOSUB ColorSet¶
¶
TIMER ON¶
MOUSE ON¶
¶
Pause:¶
IF ClickValue(4)*-1<>F(1,1) THEN¶
  F(1,1)=ClickValue(4)*-1¶
  ReDraw:¶
  GOSUB DisplayCoordinates¶
  GOTO CalculateScreen¶
END IF¶
IF ClickValue(5)*-1<>F(2,1) THEN¶
  F(2,1)=ClickValue(5)*-1¶
  GOTO ReDraw¶
END IF¶
IF INKEY$="" THEN GOTO Pause¶
¶
OBJECT.OFF¶
TIMER OFF¶
MOUSE OFF¶
LOCATE 15,1¶
PRINT "Red Value :";ClickValue(1);"%¶
PRINT "Green Value:";ClickValue(2);"%¶
PRINT "Brown Value from :"%¶

```

```

PRINT ClickValue(3);"% Red and "ClickValue(3)*.875;"%
Green"¶
PRINT "Vanishing Point Value's X-Coordinate:"¶
PRINT "V1 ";ClickValue(4)*-1;" and V2 ";ClickValue(5)*-1¶
END¶
¶
¶
DisplayCoordinates:¶
¶
LOCATE 1,63¶
PRINT F(1,1)","F(1,2)","F(1,3)¶
LOCATE 2,63¶
PRINT F(2,1)","F(2,2)","F(2,3)¶
RETURN¶
¶
CheckTable:¶
¶
IF NumClicks=0 THEN RETURN¶
¶
FOR i=1 TO NumClicks¶
  mstat=MOUSE(0)¶
  mx=MOUSE(1)-6¶
  my=MOUSE(2)¶
  IF mx>=ClickTable(i,1) THEN¶
    IF my>=ClickTable(i,2) THEN¶
      IF mx<=ClickTable(i,3) THEN¶
        IF my<=ClickTable(i,4) THEN¶
          ¶
          ClickValue(i)=(my-ClickTable(i,2))¶
          OBJECT.Y i,ClickTable(i,2)+ClickValue(i)+12¶
          ¶
        END IF¶
      END IF¶
    END IF¶
  END IF¶
NEXT i¶
IF MOUSE(0)=-1 THEN CheckTable¶
RETURN¶
¶
ColorSet:¶
  Red=ClickValue(1)/100¶
  Green=ClickValue(2)/100¶
  DrawColor=ClickValue(3)/100¶
  PALETTE 2,Red,0,0¶
  PALETTE 3,0,Green,0¶
  PALETTE 1,DrawColor,(COLOR*.875),0¶
RETURN¶
¶
¶
SUB DefMove (sx,sy,yd,po,mo) STATIC¶
  SHARED NumClicks¶
  ¶
  x=sx*8 'Coordinates for Line *10 at 60 Drawing Color¶
  ¶
  y=sy*8¶
  ¶

```

```

LINE (x,y)-(x+20,y+8+yd),,B¶
¶
'Extras desired?¶
¶
IF mo AND 1 THEN ' Scale¶
¶
  FOR sk=y TO y+yd+8 STEP (yd+8)/16 '16 Units¶
    LINE (x,sk)-(x+2,sk)¶
    LINE (x+20,sk)-(x+18,sk)¶
  NEXT sk¶
¶
END IF¶
¶
IF mo AND 2 THEN ' Text¶
¶
  SHARED Text$¶
  sy=sy-LEN(Text$)¶
  FOR txt=1 TO LEN(Text$)¶
    LOCATE sy+txt,sx+2¶
    PRINT MID$(Text$,txt,1)¶
  NEXT txt¶
¶
END IF¶
¶
'Enter Click Value in Table ¶
¶
NumClicks=NumClicks+1¶
ClickTable(NumClicks,1)=x¶
ClickTable(NumClicks,2)=y¶
ClickTable(NumClicks,3)=x+20¶
ClickTable(NumClicks,4)=y+yd¶
ClickID(NumClicks)=1 '1 set for Slider¶
ClickValue(NumClicks)=po 'Beginning Value defined by
the User¶
¶
OPEN "T&T2:slider2" FOR INPUT AS NumClicks¶
OBJECT.SHAPE NumClicks,INPUT$(LOF(NumClicks),NumClicks)¶
CLOSE NumClicks¶
OBJECT.X NumClicks,x-1¶
OBJECT.Y
NumClicks,ClickTable(NumClicks,2)+ClickValue(NumClicks)+1
2¶
OBJECT.ON NumClicks¶
¶
END SUB¶
¶
CubeData:¶
REM x,y,z¶
DATA 32, 20, 20¶
DATA -32, 20, 20¶
DATA -32,-20, 20¶
DATA 32,-20, 20¶
DATA 32, 20,-20¶
DATA -32, 20,-20¶
DATA -32,-20,-20¶

```

```

DATA 32,-20,-20¶
DATA 255,0,0¶
¶
REM p1,p2¶
DATA 1,2¶
DATA 2,3¶
DATA 3,4¶
DATA 4,1¶
DATA 1,5¶
DATA 5,6¶
DATA 6,7¶
DATA 7,8¶
DATA 8,5¶
DATA 4,8¶
DATA 3,7¶
DATA 2,6¶
DATA 255,0,1¶
¶
PyramidData:¶
DATA -32, 25,-20¶
DATA 32, 25,-20¶
DATA 32, 25, 20¶
DATA -32, 25, 20¶
DATA 0, 65, 0¶
DATA 255,0,0¶
¶
DATA 1,2¶
DATA 2,3¶
DATA 3,4¶
DATA 4,1¶
DATA 5,1¶
DATA 5,2¶
DATA 5,3¶
DATA 5,4¶
DATA 255,0,0¶

```

**Arrays**

B	screen coordinates
D	differences from the coordinate computation
F	vanishing point coordinates (both graphics)
ClickID	identifier for slider
ClickTable	slider coordinates
ClickValue	value of a slider
P	spatial coordinates
C	compound specification

<i>Variables</i>	NumClicks	number of defined click arrays
	Last	value read, equals 0 when program ends
	Green	value for green
	CBase	object connection identifier
	NumPoints	number of points to be drawn
	MaxPoints	maximum number of object points
	Red	value for red
	Text	text output for slider definition
	Connections	number of connections
	ZCoord	Z-coordinates of screen plane
	DrawColor	drawing color for "Brown"
	i,j,k	floating variables
	lambda	coordinate calculation factor
	mo	mode parameters for slider extras
	mstat	mouse status
	mx,my	mouse coordinates
	po	slider starting position
	px,py,pz	coordinates of one point in space
	sk	floating variable scaling
	sx,sy	text output coordinates
	txt	text output floating variable
	v1,v2	combination points
	x,y	slider positions
	x1,y1	screen coordinates for output (1st point)
	x2,y2	screen coordinates for connection (2nd point)
	yd	slider status

***Program description***

First the `graphics` library opens, which contains the important graphic routines. The `DATA` pointer then moves to the needed data, and all arrays beginning with `B` or `C` are defined as integers. Base array indices are set to 1. The variables here have similar functions to those in the earlier programs. The slider arrays and variables are new. There are also changes to most of the previous variables as well.

The array containing the vanishing point has an additional index on it. This index corresponds to the number of vanishing points and makes later development easier. This index lets you put up to 40 pixels as vanishing points. This index is ideal for spacing between projection surfaces and vanishing points.

A new method must be used for setting the vanishing points. This new value is set in a subroutine.

The color setting is new as well. All four colors are available; the background can prevent the proper effect if you select the wrong color. The other three colors need no explanation.

The slider definitions follow. The values of the first three sliders affect the colors. The last two sliders make it possible for you to set the vanishing points in horizontal directions.



The point and connection reader routines act as normal. Only the computation of the graphic has a slight change to it. The loop counts from one vanishing point to the next. This counter also depends on the screen coordinates as an index.

Before screen display, the screen clears. Both vanishing points appear in their respective colors. When the grid for the second point is drawn, the program goes into a new character mode (see the table in Chapter 4 for the modes). When you draw with the second color, any overlapping between this color and red lines change to brown. At the end of the loop, the character mode returns to normal status and the drawing color returns to 1.

A mouse and time interrupt activate. The first interrupt reads the sliders. The second interrupt resets the colors when you change them. The wait loop checks the program for one vanishing point or two vanishing points. If there are two, the value transfers over and the screen is recalculated.

The system waits for a key press. When this occurs, the program turns all objects, sliders, mouse and time readers off, and displays all established values on the screen.

## 3.5 The Amiga fonts

There are two sources of fonts on the Amiga:

1. ROM fonts which are memory resident in the Amiga.
2. Disk-resident fonts included in the `fonts` directory of the Workbench diskette.

The following program lets you access character sets through the `SUB` command `FontSet` which gives you access to both ROM and RAM character sets. This is called as follows:

```
DiskFont "name",height%
```

To tell which character sets are on the Workbench diskette under which names, enter a directory command such as the following:

```
FILES "SYS:fonts"
```

Along with these character sets, you can also access the ROM character set `topaz` in 8- and 9-point sizes. It's extremely important that you enter the name `topaz` in lowercase characters. The `OpenFont()` function is case sensitive. It will not read entries like `Topaz` or `TOPAZ` as the ROM character set `topaz`. Instead, it loads the 11-point disk font `Topaz`.

```
'#####
'#                                     #
'# Program: Set TextFont             #
'# Author: tob                       #
'# Date: 12/8/87                     #
'# Version: 1.0                      #
'#                                     #
'#####

DECLARE FUNCTION OpenDiskFont& LIBRARY
DECLARE FUNCTION OpenFont& LIBRARY

LIBRARY "T&T2:bmaps/diskfont.library"
LIBRARY "T&T2:bmaps/graphics.library"

demo: ' Demonstration of SetFont Command
LOCATE 4,1
FontSet "Sapphire", 19
PRINT "This is Sapphire 19 Points"
FontSet "Diamond", 20
PRINT "...another TextFont..."
FontSet "Garnet", 16
```

```

        PRINT "...and yet another! Amiga has still
more!"
        FontSet "ruby", 12
        PRINT "However this should be enough to
demonstrate the point!"
        FontSet "topaz", 8
        ¶
        LIBRARY CLOSE
        END
¶
SUB FontSet (FontName$, FontHeight%) STATIC
    f.old&    = PEEKL(WINDOW(8)+52)
    f.prefs%  = 0
    FontName0$ = FontName$ + ".font" + CHR$(0)
    tAttr&(0) = SADD(FontName0$)
    tAttr&(1) = FontHeight%*2^16 + f.prefs%
    f.new&    = OpenFont&(VARPTR(tAttr&(0)))
    f.check%  = PEEKW(WINDOW(8) + 60)
    ¶
    IF f.new& = 0 THEN
        f.new& = OpenDiskFont&(VARPTR(tAttr&(0)))
    ELSEIF f.check% <> FontHeight% THEN
        CALL CloseFont(f.new&)
        f.new& = OpenDiskFont&(VARPTR(tAttr&(0)))
    END IF
¶
    IF f.new& <> 0 THEN
        CALL CloseFont(f.old&)
        CALL SetFont(WINDOW(8), f.new&)
    ELSEIF UCASE$(FontName$) = "UNDO" THEN
        CALL CloseFont(f.old&)
        CALL SetFont(original&)
    ELSE
        BEEP
    END IF
END SUB

```

**Variables**

FontName\$	character set name
FontName0\$	similar to FontName\$ except it ends with CHR\$(0)
FontHeight%	height of the font in pixels
f.old&	address of previously active character set
f.prefs%	preference bits
tAttr&()	text attribute structure; variable array used as memory
f.new&	address of newly opened character set
f.check%	current height of new character set

***Program  
description***

In order to open a character set, a `TextAttr` structure must be filled out. This is stored in the `tAttr&` array. The address at the beginning of this field (taken from `VARPTR`) calls the graphic routine `OpenFont()`. This looks for a character set matching the parameters stated in the `TextAttr` structure. The normal fonts are the ROM font `topaz` in 8-point and 9-point. However if other fonts remain open, these fonts can be accessed by `OpenFont()`.

`OpenFont()` is so flexible that if it can't find a font matching the given parameters, it loads the font most closely matching the desired font. This means that the font loaded may not be the one you want. The `check%` variable checks the height of the found font, and compares it with the height found in `FontHeight%`. If the two are unequal, the opened font closes and `OpenFont()` looks for another font on diskette.

If, on the other hand, the program finds a font (`f.old&<>0`), `CloseFont()` closes the currently active font, and activates the new font with `SetFont()`. Otherwise the Amiga emits a warning beep and returns to the old font.

## 3.6 Fast and easy PRINT

The weakest command in AmigaBASIC is PRINT. This command has three disadvantages to it: Slow execution, no word wrap and no editing capabilities.

An entire page of text can take several seconds to display in a window. In addition, PRINT doesn't know when it reaches the end of a screen line. Long strings of characters move past the right border of the window instead of "wrapping around" to the next screen line. Finally, PRINT displays text and nothing more. PRINT cannot execute editor commands that might exist, such as CLEAR SCREEN, CURSOR UP, INSERT LINE, etc.

Since PRINT is one of the most frequently used commands in AmigaBASIC, here is a program that solves all of these problems. The solution is a simple one: The program activates the internal system's Console Device. This system component handles text input and output. Once active, Console Device handles all the tasks that PRINT can't handle: Fast text display, adaptation to window size and a number of editor commands.

Unfortunately, it's not that easy to adapt Console Device for your own purposes, since it must be treated as an I/O device. A number of Exec functions are necessary. However, once initialized, you have a PRINT command of much larger dimensions. With this new command's help, your program runs faster, and editor commands make programming easier.

The following program consists of the SUB programs CreatePort, RemovePort, CreateStdIO, RemoveStdIO, OpenConsole, CloseConsole, SystemOn, SystemOff and ConPrint:

```
'#####
'#                                     #
'# Program: Console Device           #
'# Author:  tob                       #
'# Date:    04/08/87                  #
'# Version: 1.0                       #
'#                                     #
'#####
¶
DECLARE FUNCTION OpenDevice% LIBRARY¶
DECLARE FUNCTION AllocMem% LIBRARY¶
DECLARE FUNCTION AllocSignal% LIBRARY¶
DECLARE FUNCTION FindTask% LIBRARY¶
DECLARE FUNCTION DoIO% LIBRARY¶
¶
```

```

LIBRARY "T&T2:bmaps/exec.library"
¶
init:  '* Control-Sequence definitions¶
        C1$ = CHR$(155) 'Control Sequence Introducer¶
        C2$ = CHR$(8)   'Backspace¶
        C3$ = CHR$(10)  'Line Feed¶
        C4$ = CHR$(11)  'VTab¶
        C5$ = CHR$(12)  'Form Feed¶
        C6$ = CHR$(13)  'CR¶
        C7$ = CHR$(14)  'SHIFT IN¶
        C8$ = CHR$(15)  'SHIFT OUT¶
        C9$ = CHR$(155) + "1E" 'RETURN¶
¶
demo:  '* Demonstration¶
        ConPrint C1$+"20CA Good Day to You!" + C9$¶
        ConPrint "It had been a normal day so far, but
while on the way to the barn we saw a very big bear!"¶
¶
SystemOff¶
¶
SUB ConPrint (text$) STATIC¶
    SHARED c.io&¶
    IF c.io& = 0 THEN : SystemOn¶
    POKEL c.io& + 36, LEN(text$)¶
    POKEL c.io& + 40, SADD(text$)¶
    e& = DoIO&(c.io&)¶
END SUB¶
¶
SUB SystemOff STATIC¶
    SHARED c.io&¶
    CloseConsole c.io&¶
END SUB¶
¶
SUB SystemOn STATIC¶
    SHARED c.io&, c.c$¶
    OpenConsole c.io&¶
    POKEW c.io& + 28, 3¶
END SUB¶
¶
SUB OpenConsole (result&) STATIC¶
    CreatePort "basic.con", 0, c.port&¶
    IF c.port& = 0 THEN ERROR 255¶
    CreateStdIO c.port&, c.io&¶
    POKEL c.io& + 36, 124¶
    POKEL c.io& + 40, WINDOW(7)¶
    dev$ = "console.device" + CHR$(0)¶
    c.error% = OpenDevice%(SADD(dev$), 0, c.io&, 0)¶
    IF c.error% <> 0 THEN ERROR 255¶
    result& = c.io&¶
END SUB¶
¶
SUB CloseConsole (io&) STATIC¶
    port& = PEEKL (io& + 14)¶
    CALL CloseDevice(io&)¶
    RemovePort port&¶
    RemoveStdIO io&¶

```

```

END SUB¶
¶
SUB CreateStdIO (port&, result&) STATIC¶
  opt& = 2^16¶
  result& = AllocMem&(48, opt&)¶
  IF result& = 0 THEN ERROR 7¶
  POKE result& + 8, 5¶
  POKE result& + 14, port&¶
  POKEW result& + 18, 50¶
END SUB¶
¶
SUB RemoveStdIO (io&) STATIC¶
  IF io& <> 0 THEN¶
    CALL FreeMem(io&, 48)¶
  END IF¶
END SUB¶
¶
SUB CreatePort (port$, pri%, result&) STATIC¶
  opt& = 2^16¶
  byte& = 38 + LEN(port$)¶
  port& = AllocMem&(byte&, opt&)¶
  IF port& = 0 THEN ERROR 7¶
  POKEW port&, byte&¶
  port& = port& + 2¶
  sigBit% = AllocSignal%(-1)¶
  IF sigBit% = -1 THEN¶
    CALL FreeMem(port&, byte&)¶
    ERROR 7¶
  END IF¶
  sigTask& = FindTask& (0)¶
  ¶
  POKE port& + 8 , 4¶
  POKE port& + 9 , pri%¶
  POKE port& + 10, port& + 34¶
  POKE port& + 15, sigBit%¶
  POKE port& + 16, sigTask&¶
  POKE port& + 20, port& + 24¶
  POKE port& + 28, port& + 20¶
  FOR loop% = 1 TO LEN(port$)¶
    char% = ASC(MID$(port$, loop%, 1))¶
    POKE port& + 33 + loop%, char%¶
  NEXT loop%¶
  CALL AddPort (port&)¶
  result& = port&¶
END SUB¶
¶
SUB RemovePort (port&) STATIC¶
  byte& = PEEKW(port& - 2)¶
  sigBit% = PEEK (port& + 15)¶
  CALL RemPort (port&)¶
  CALL FreeSignal (sigBit%)¶
  CALL FreeMem (port& - 2, byte&)¶
END SUB¶

```

As you can see, you can use the new ConPrint much the same as you used the normal PRINT:

```
ConPrint "displayed text"
```

However, ConPrint works much faster than PRINT. Also, long lines of text are tailored to fit the width of the window. If the text is longer than the window is wide, the text wraps around to the next window line. You also have the following editor sequences available:

C1\$	CSI (Control Sequence Introducer)
C2\$	Backspace (1 character to the left)
C3\$	Linefeed (1 line down)
C4\$	VTab (one line up)
C5\$	Formfeed (clear screen)
C6\$	CR (start of next line)
C7\$	SHIFT IN (caps)
C8\$	SHIFT OUT (normal)
C9\$	RETURN (end of line)

These are the simplest editor text sequences. You add them to text strings using the plus sign character (+). For example:

```
ConPrint "Hello, Worker!"+C9$
```

Console Device can do a lot more. The following editor sequences begin immediately after the control sequence introducer (C1\$). The editor sequences are as follows:



C1\$ +	Definition
"[n]@"	Insert [n] characters in this line
"[n]A"	Cursor [n] lines up
"[n]B"	Cursor [n] lines down
"[n]C"	Cursor [n] characters right
"[n]D"	Cursor [n] characters left
"[n]E"	Cursor [n] characters down + to start of line
"[n]F"	Cursor [n] characters up + to start of line
"[n];[n]H"	Cursor to line [n], column [n]
"J"	Clear screen from current cursor position
"K"	Delete line at current cursor position
"L"	Insert line
"M"	Delete line
"[n]P"	Delete character to right of cursor
"[n]S"	Scroll [n] lines up
"[n]T"	Scroll [n] lines down
"20h"	Set mode
"20l"	Reset mode
"[n];[n];[n]m"	Graphic mode
	Style:
	0=normal
	1=bold
	3=italic
	4=underline
	7=reverse
	Foreground color:
	30-37
	Background color:
	40-47
"[n]t"	Window height in raster lines
"[n]u"	Line length in pixels
"[n]x"	Indent [n] characters
"[n]y"	[n] lines spacing from top border



**4**

# **User-friendliness**



## 4. User-friendliness

A few years ago, the term "user-friendly" didn't exist in computing. The user had to enter or type in data to instruct the computer exactly what he or she wanted the computer to do. If the data was entered incorrectly, the computer returned an error message (if the user was lucky). The manual was a necessity for the user to survive computing.

As home computers became more common, designers helped shape the technology which brought about user-friendly interfaces between the computer and user. *Intuition* is the Amiga's user interface, using windows, icons and the mouse as user input.

User-friendly program design is important to the developer, and even more important to the user. Most users prefer a program that makes operation simple and clear, without having to even pick up a manual. In addition, user-friendly programs are more attractive to the consumer, and may mean more profits for the developer.

This chapter shows you how you can make your programs as user-friendly as possible. This sort of programming focuses on input, selection and control. Often an icon or other self-explanatory graphic helps the user understand program operation better. In any case, most programming for user response should be mouse-based, and not just for starting and quitting the program. Here are some easily implemented functions that you can include in your own programs.

---

## 4.1 Input gadgets

Not everything required for program control is accomplished using menus. Therefore, we must look for alternatives. What are those alternatives? See the Workbench disk for some examples. Preferences is a good example of alternatives to drop-down menus. When you open Preferences, you can easily select any of the possible options. Therefore, Preferences is considered user-friendly. The Preferences program uses normal gadgets, sliders, filled gadgets and even scrolling tables to allow the user to make the selections.

Sliders control colors, key repeat delay, key repeat speed and the time between the clicks of a double-click. Filled gadgets indicate the number of characters per line and the status of Workbench interlace mode. Scrolling tables in the Change Printer section helps the user select the correct printer driver. Normal gadgets on the main screen execute an action such as Save, Use or Cancel, by clicking on them.

The following programs show examples of all the above user-friendly gadgets. For openers, we need an output window to display these gadgets. AmigaBASIC usually opens a window directly after loading it. However, BASIC windows have some limitations, so we'll directly open a window using the Intuition library. The Amiga operating system libraries offer much more control than standard BASIC programming.

---

### 4.1.1 An Intuition window

The first example program does nothing more than open an Intuition window on the Workbench screen. We call the file named `intuition.library` for this and which is also used in the other example programs in this section. This program requires both `intuition.library` and `exec.library` to function. Use the `ConvertFD` BASIC program in the `BASICDemos` drawer on the `Extras` disk to create the `exec.library` and `intuition.library` files. When running the `ConvertFD` program, be sure to enter the correct and complete disk identifier, pathname and filename at the prompt. If you don't, the `ConvertFD` program won't find the correct file. Here is an example run of the `ConvertFD` program, for Workbench 1.3 users that creates the `exec.bmap` file on a RAM disk:

```
Enter name of .fd file to read> "Extras 1.3:FD1.3/exec_lib.fd"  
Enter name of .bmap file to produce > ram:exec.bmap
```

The example programs require the `exec.library` (`exec_lib.FD`), the `intuition.library` (`intuition_lib.f.d`) and the `graphics.library` (`graphics_lib.f.d`) to operate.

The companion diskette which accompanies this book contains the `.bmap` files in a drawer named `bmaps`. Therefore the example programs that follow “look” in this drawer for the `.bmap` files. If your `.bmap` files are on a disk with a different name, or in a different drawer, alter the `LIBRARY` commands so the `.bmap` files can be opened.

The example programs that follow contain some BASIC lines which you must enter as one line in AmigaBASIC although they appear on two lines in this book. Formatting the program listings to fit into this book has split some long BASIC lines into two lines. An end-of-paragraph (¶) character shows where a BASIC line actually ends. When you see this character, press the (↵) key in the BASIC editor. For example, the following line appears as two lines in this book, but the ¶ marker indicates that you must enter it as one line in AmigaBASIC:

```
WinDef NWindow, 100, 50, 460, 150, 32+64+512&, 15&+4096&,
0&, Title$¶
```

The ¶ marker shows the actual end of the BASIC line. Here is our first example program, which opens an Intuition window on the Workbench screen:

```
'*****¶
'*                                     *¶
'* Open Window under Intuition *¶
'* ----- *¶
'*                                     *¶
'* Author : Wolf-Gideon Bleek *¶
'* Date   : May 22 '88 *¶
'* Version: 1.1 *¶
'* Operating system:V1.2 & V1.3 *¶
'* Name   : Intuition-window *¶
'*                                     *¶
'*****¶
    OPTION BASE 1¶
    DEF LNG a-z¶
¶
' Bmaps located on disk named T&T2,yours may differ¶
    LIBRARY "T&T2:bmaps/exec.library"¶
    DECLARE FUNCTION AllocMem LIBRARY¶
    LIBRARY "T&T2:bmaps/intuition.library"¶
    DECLARE FUNCTION OpenWindow LIBRARY¶
    ¶
    MList      = 0&¶
    ¶
MainProgram:¶
    GOSUB OpenAll¶
    ¶
    ' Main part¶
```

```

FOR i = 1 TO 10000 : NEXT i
  1
  GOSUB CloseAll
  1
END
OpenAll:
  Title$ = "My first BASIC-Window"
  WinDef NWindow, 100, 50, 460, 150, 32+64+512, 15+4096,
0, Title$
  WinBase = OpenWindow(NWindow)
  IF WinBase = 0 THEN ERROR 7
RETURN
CloseAll:
  CloseWindow(WinBase)
  CALL UnDef
RETURN
'-----
SUB DefChip(Buffer, Size) STATIC
  SHARED MList
  Size=Size+8
  Buffer=AllocMem(Size, 65538)
  IF Buffer>0 THEN
    POKE Buffer, MList
    POKE Buffer+4, Size
    MList=Buffer
    Buffer=Buffer+8
  ELSE
    ERROR 7
  END IF
END SUB
SUB UnDef STATIC
  SHARED MList
undef.loop:
  IF MList>0 THEN
    Address = PEEKL(MList)
    ListSize = PEEKL(MList+4)
    FreeMem MList, ListSize
    MList = Address
    GOTO undef.loop
  END IF
END SUB
1
SUB WinDef(bs, x%, y%, b%, h%, IDCMP, f, gad, T$) STATIC
  Size = 48+LEN(T$)+1
  DefChip bs, Size
  POKEW bs, x% ' Left Corner
  POKEW bs+ 2, y% ' Top Corner
  POKEW bs+ 4, b% ' Width
  POKEW bs+ 6, h% ' Height
  POKEW bs+ 8, 65535 ' Detail- BlockPen
  POKEW bs+10, IDCMP ' IDCMP Flags
  POKEW bs+14, f ' Flags
  POKEW bs+18, gad ' First Gadget
  POKEW bs+26, bs+48 ' Title
  POKEW bs+46, 1 ' ScreenType
  FOR i%=1 TO LEN(T$)

```



```

        POKE bs+47+i%,ASC(MID$(T$,i%,1))
    NEXT i
END SUB

```

### ***Program description***

The most important elements of the example program appear at the end of the program. Here you'll find the three subprograms which fulfill the three tasks required to open our Intuition window. `DefChip()` requests a memory area of the desired size. This is reserved using `AllocMem()`, an operating system function of `exec.library`. The routine stores two values in the first eight bytes (a list of the allocated memory). `UnDef()` releases the allocated memory. `Mlist` releases all memory areas one after another.

The first two subprograms are initialization routines. The `WinDef()` subprogram holds the most importance here, since it places the specified data in a `NewWindow` structure. Intuition needs this data before it can open a new window. `WinDef()` only places the data in a new memory area. Everything else is considered a subroutine of the main program.


Now that we know the task of the subprograms, let's look at the main program. First the `Intuition` and `Exec` libraries open. The program uses functions from both. The main section jumps to the `OpenAll` subroutine. The definition of a `NewWindow` structure using `WinDef()` is called here. This structure passes to Intuition by means of `OpenWindow()`. A correctly opened new window returns a pointer to the `Window` structure; if an error occurs it returns a 0. The new structure contains all the data necessary to create our own Intuition window.

After returning from the `OpenAll` subroutine, the program pauses using a `FOR/NEXT` loop so you can see the window for a moment. Then the BASIC interpreter jumps to another subroutine called `CloseAll`. This routine closes our window and releases the allocated memory.

Now we have a basis for our own user-friendly professional programs. Using this Intuition window, we can insert the user-friendly input facilities similar to those found in the Preferences program. Let's look at the first of these: gadgets.

## **4.1.2 Gadgets**

The first user-friendly gadgets that we see in the Preferences program perform a direct action after the user selects them with the mouse. By moving the pointer onto one of these gadgets and pressing the left mouse button, you select different choices such as `Change Printer`, `Edit Pointer` or `Cancel`.

These gadgets can be accessed through BASIC programming. AmigaBASIC pulls its gadget data from Intuition. Our new gadgets can easily be merged into the Intuition window program you read about in the preceding segment. The following subprogram defines a new gadget field (remember that the ¶ character tells you when to press the  key). You can either save it as a separate BASIC program and append it later, or type it in at the end of the Intuition window program you entered previously:

```
SUB GadgetDef(bs,nx,x%,y%,b%,h%,f%,a%,T%,i,txt,si,n%)
STATIC¶
  DefChip bs,44¶      ' Gadget-Structure length¶
  POKEL bs ,nx       '*NextGadget¶
  POKEW bs+ 4,x%     ' LeftEdge¶
  POKEW bs+ 6,y%     ' TopEdge¶
  POKEW bs+ 8,b%     ' Width¶
  POKEW bs+10,h%     ' Height¶
  POKEW bs+12,f%     ' Flags¶
  POKEW bs+14,a%     ' Activation¶
  POKEW bs+16,T%     ' GadgetType¶
  POKEL bs+18,i      ' GadgetRender¶
  POKEL bs+26,txt    '*GadgetText¶
  POKEL bs+34,si     ' SpecialInfo¶
  POKEW bs+38,n%    ' GadgetID¶
END SUB¶
```

This routine places the correct values necessary for a gadget in an area of memory. The variable *bs* contains the base address of our memory area. It also handles the return value of the memory allocation routine. *nx* marks the starting address of the next free gadget area, allowing us to use more than one gadget. We'll make use of this value later to add more gadgets to the window. Use *x%*, *y%*, *b%* and *h%* to define the dimensions of the gadget.

Two flags, which we will discuss later, are defined with *f%* and *a%*. Use *t%* to set the gadget type (set at 1 in this first example—later on we'll add other gadget types). *i* and *txt* contain additional graphic information. *i* allows you to define the type of border to be displayed. *txt* defines the text inside the gadget. Finally, we place an identification value in *n%*. This value helps the program determine which gadget was selected by the user.

Insert the *GadgetDef* subprogram from above at the end of the Intuition window example program, and enter the following line in the *OpenAll* subroutine:

```
GadgetDef Gadget,0&,50,50,90,15,0,1,1,0&,0&,0&,1
```

This call to *GadgetDef* defines our new gadget. This causes the following results:

- The address of the gadget structure is found after the close of the routine in *Gadget*.

- Only one gadget is used; no more gadgets are contained in the gadget structure.
- The position is 50,50.
- The gadget is 90 pixels wide and 15 pixels high.
- It is handled as a gadget that reacts to a mouse click.
- The gadget is of boolean type and can only be activated.
- There are no border graphics and no text inside the gadget (an invisible gadget).
- No additional structure is needed.
- The gadget is accessed by number 1.

This defined gadget structure interfaces with the new window structure using the following command sequence:

```
WinDef NWindow, 100, 50, 460, 150, 32+64+512&, 15&+4096&,
Gadget, Title$¶
```

Although you could start the program at this time, we recommend not doing so yet. The program provides no routines for gadget checking. Since the gadget structure contains no graphic border the gadget is invisible. The window appears if you enter RUN now. After doing some clicking, you might or might not find the region allocated for the gadget definition. Clicking the gadget produces no reaction. We must write another subroutine that jumps from the main program when it encounters some information.

The new program section below reads a gadget and gets a new Intuition message from the information port of the Intuition window. Then it branches to a new subroutine, `IntuitionMsg`, which reads and determines the result of the message:

```
GADGETDOWN = 32&¶
GADGETUP   = 64&¶
CLOSEW     = 512&¶
¶
MList      = 0&¶
Info       = 1¶
¶
MainProgram:¶
GOSUB OpenAll¶
' Main part¶
MainLoop:¶
IF Info = 1 THEN¶
  IntuiMsg = GetMsg(UserPort)¶
  IF IntuiMsg > 0 THEN GOSUB IntuitionMsg¶
  GOTO MainLoop¶
```

```

        END IF ¶
    ¶
    GOSUB CloseAll¶
    ¶
END¶

```

The `GetMsg()` function makes it possible to test the message port of a window for a message. This message port lies in the window structure that `OpenWindow()` returned to us. In this routine a new variable must be initialized to read the message port:

```
UserPort = PEEKL(WinBase+86) ¶
```

Now we can add the subroutine which controls the handling of the newly received message. It first determines the type of the message, since different actions on our window will return different messages. One message that we can handle is the `CloseWindow` message. This message returns when the user clicks on the window's close gadget. Our subroutine also displays the gadget number on the screen which was written in the gadget structure.

```

IntuitionMsg: ¶
    MsgTyp    = PEEKL(IntuiMsg+20) ¶
    Item      = PEEKL(IntuiMsg+28) ¶
    GadgetNr% = PEEK(Item+39) ¶
    CALL ReplyMsg(IntuiMsg) ¶
    ¶
    IF (MsgTyp = GADGETDOWN) THEN¶
        'activated¶
        PRINT "DOWN Gadget-Nr.:";GadgetNr%¶
    END IF¶
    ¶
    IF (MsgTyp = GADGETUP) THEN¶
        'rel verify mode¶
        PRINT "UP  Gadget-Nr.:";GadgetNr%¶
    END IF¶
    ¶
    IF (MsgTyp = CLOSEW) THEN¶
        'System-Gadget Window closer¶
        PRINT "CLOSE WINDOW"¶
        Info = 0¶
    END IF¶
RETURN¶

```

Assemble the program sections and start the program. The Intuition window appears on the Workbench screen. Now for our first test. Click in the upper left quarter of the screen until you find the invisible gadget. It's invisible because no border or text definition exists in the Gadget structure (the gadget lies at screen location 50,50). When you click on it with the mouse, the new gadget becomes visible. It disappears after you release the mouse button. Then the gadget number appears in the AmigaBASIC Output window.

For the second test of our intuition message subroutine, click on the close gadget of the window. First the text appears in the Output window, then the window closes. We have just completed the groundwork required for using gadgets.

Here is the complete program:

```

*****
'*                                     *
'* Gadgets with Intuition             *
'* -----                            *
'*                                     *
'* Author : Wolf-Gideon Bleek        *
'* Date   : May 23 '88                *
'* Name   : Gadget-one                *
'* Version: 1.2                       *
'* System : V1.2 & V1.3               *
'*                                     *
*****
OPTION BASE 1
DEFLNG a-z

' Bmaps located on disk named T&T2,yours may differ
LIBRARY "t&T2:bmaps/exec.library"
DECLARE FUNCTION AllocMem LIBRARY
DECLARE FUNCTION GetMsg LIBRARY
LIBRARY "t&t2:bmaps/intuition.library"
DECLARE FUNCTION OpenWindow LIBRARY
GADGETDOWN = 32
GADGETUP   = 64
CLOSEW     = 512

MList     = 0
Info      = 1

MainProgram:
GOSUB OpenAll
' Main part
MainLoop:
IF Info = 1 THEN
  IntuiMsg = GetMsg(UserPort)
  IF IntuiMsg > 0 THEN GOSUB IntuitionMsg
  GOTO MainLoop
END IF

GOSUB CloseAll

END

OpenAll:
GadgetDef Gadget, 0, 50, 50, 90, 15, 0, 1, 1, 0, 0, 0, 1
Title$ = "The invisible gadget"
WinDef NWindow, 100, 50, 460, 150, 32+64+512, 15+4096,
Gadget, Title$
WinBase = OpenWindow(NWindow)

```

```

    IF WinBase = 0 THEN ERROR 7
    UserPort = PEEKL(WinBase+86)
RETURN
┌
CloseAll:
    CALL CloseWindow(WinBase)
    CALL UnDef
RETURN
┌
IntuitionMsg:
    MsgTyp = PEEKL(IntuiMsg+20)
    Item = PEEKL(IntuiMsg+28)
    GadgetNr% = PEEK(Item+39)
    CALL ReplyMsg(IntuiMsg)
┌
    IF (MsgTyp = GADGETDOWN) THEN
        'activated
        PRINT "DOWN Gadget-Nr.:";GadgetNr%
    END IF
┌
    IF (MsgTyp = GADGETUP) THEN
        'rel verify mode
        PRINT "UP Gadget-Nr.:";GadgetNr%
    END IF
┌
    IF (MsgTyp = CLOSEW) THEN
        'System-Gadget Window closer
        PRINT "CLOSE WINDOW"
        Info = 0
    END IF
RETURN
'-----
┌
SUB DefChip(Buffer,Size)STATIC
    SHARED MList
    Size=Size+8
    Buffer=AllocMem(Size,65538&)
    IF Buffer>0 THEN
        POKEL Buffer,MList
        POKEL Buffer+4,Size
        MList=Buffer
        Buffer=Buffer+8
    ELSE
        ERROR 7
    END IF
END SUB
┌
SUB UnDef STATIC
    SHARED MList
undef.loop:
    IF MList>0 THEN
        Address = PEEKL(MList)
        ListSize = PEEKL(MList+4)
        FreeMem MList, ListSize
        MList = Address
        GOTO undef.loop

```

```

    END IF
END SUB
Sub WinDef (bs, x%, y%, b%, h%, IDCMP, f, gad, T$) STATIC
    Size = 48+LEN(T$)+1
    DefChip bs, Size
    POKEW bs, x% ' Left Corner
    POKEW bs+ 2, y% ' Top Corner
    POKEW bs+ 4, b% ' Width
    POKEW bs+ 6, h% ' Height
    POKEW bs+ 8, 65535& ' Detail- BlockPen
    POKEW bs+10, IDCMP ' IDCMPFlags
    POKEW bs+14, f ' Flags
    POKEW bs+18, gad ' First Gadget
    POKEW bs+26, bs+48 ' Title
    POKEW bs+46, 1 ' Screen Type
    FOR i%=1 TO LEN(T$)
        POKEW bs+47+i%, ASC(MID$(T$, i%, 1))
    NEXT
END SUB
Sub GadgetDef (bs, nx, x%, y%, b%, h%, f%, a%, T$, i, txt, si, n%) STATIC
    DefChip bs, 44& ' Gadget-Structure length
    POKEW bs, nx '*NextGadget
    POKEW bs+ 4, x% ' LeftEdge
    POKEW bs+ 6, y% ' TopEdge
    POKEW bs+ 8, b% ' Width
    POKEW bs+10, h% ' Height
    POKEW bs+12, f% ' Flags
    POKEW bs+14, a% ' Activation
    POKEW bs+16, T% ' GadgetType
    POKEW bs+18, i ' GadgetRender
    POKEW bs+26, txt '*GadgetText
    POKEW bs+34, si ' SpecialInfo
    POKEW bs+38, n% ' GadgetID
END SUB

```

---

### 4.1.3 Gadget borders and text

A very important feature has been missing from our gadgets: visibility. You had to hunt around for this gadget and could only see it when you clicked on it.

Let's clear up this small problem. In the description of the subprogram `GadgetDef` we learned about the variables `txt` and `i`. `GadgetDef` uses these variables to enter the basic graphic elements of Intuition. We can define an `IntuiText` structure for the gadget with `txt` and define a graphic or line border with `i`.

First, let's look at the text. For this we need the `IntuiText` structure. It defines the position, color, character set, character size and the text. The following subprogram allocates an area of memory and fills this allocated area with the required data:

```

SUB IntuiText(bs, c1%, x%, y%, T$, nx) STATIC
  Size=20+LEN(T$)+1 ' Structure length+ Text length+
  Nullbyte
  DefChip bs, Size
  POKE bs, c1 ' FrontPen
  POKE bs+ 2, 1 ' DrawMode
  POKEW bs+ 4, x% ' Left corner
  POKEW bs+ 6, y% ' Top corner
  POKEL bs+12, bs+20 ' IText
  POKEL bs+16, nx ' NextText
  FOR i%=1 TO LEN(T$)
    POKE bs+19+i%, ASC(MID$(T$, i%, 1))
  NEXT
END SUB

```

Following the starting address of the structure we insert the character color of the text and the text's starting position in pixels. We then supply a pointer to the text. Since the pointer is allowed more text, you can supply a pointer to another `IntuiText` structure as the ending value.

The routine itself sets the drawing mode to JAM2 (i.e., foreground and background colors are "jammed" into the selected area). This ensures that the drawing mode overwrites the background, so you can clearly read the text later. You can adjust the second color to some degree by increasing a value in the parameter list and `POKE` the new color value into `bs+1`!. To insert text in the gadget, first you must put a text into the subroutine and then append the text to the `Gadget` definition routine as follows:

```

TestTxt$ = "Test-Text"
IntuiText Text, 2, 10, 2, TestTxt$, 0
GadgetDef Gadget, 0, 50, 50, 90, 15, 0, 1, 1, Edge, Text, 0, 1

```

Next we should be concerned with border lines. Their main purpose is to create a border for the mouse click. In addition, border lines supply underlining for texts that need it. Because the edges are also passed in the gadget structure, we can create a separate structure. This structure needs a coordinate table, the color and the position, which can be treated as a normal border structure. The coordinates appear in memory following the structure.

We have developed a somewhat different subprogram for the border structure. It inserts the value into memory and calculates the values of the coordinates required by the table. This makes it very simple to define a box around a gadget. Here is the complete function:



```

SUB Border (bs, x%, y%, c%, b%, h%) STATIC
  DefChip bs, 48&      ' Structure length+Coordinate table
  POKEW bs, x%        ' Left corner
  POKEW bs+2, y%      ' Top corner
  POKE  bs+4, c%      ' FrontPen
  POKE  bs+7, 8       ' Count
  POKEW bs+8, bs+16   '*XY
  FOR i%=0 TO 1
    POKEW bs+22+i%*4, h%-1
    POKEW bs+24+i%*4, b%-1
    POKEW bs+32+i%*4, 1
    POKEW bs+38+i%*4, h%-1
    POKEW bs+40+i%*4, b%-2
  NEXT
END SUB

```

The routine above contains the corresponding values needed to define the border. Then we insert the structure's base address in the definition of the gadget (as we did in the `IntuiText` structure). Now the `Border` structure combines with the `Gadget` structure. Here is an example:

```

Border Edge, 0, 0, 3, 90, 15
GadgetDef Gadget, 0&, 50, 50, 90, 15, 0, 1, 1, Edge, Text, 0&, 1

```

The gadget now contains text and is surrounded by a border. You can enlarge this border if you wish.

We would now like to show the complete listing. This program listing contains all of the subroutines and definitions mentioned above. You can use this to determine whether you have made any errors in putting the modules together:

```

*****
'*                                     *
'* Boolean-Gadgets with Intuition *
'* ----- *
'*                                     *
'* Author : Wolf-Gideon Bleek *
'* Date  : May 23 '88 *
'* Name   : Gadgets *
'* Version: 1.2 *
'* System : V1.2 & V1.3 *
'*                                     *
*****
OPTION BASE 1
DEFLNG a-z

' Bmaps located on disk named T&T2,yours may differ
LIBRARY "t&T2:bmaps/exec.library"
DECLARE FUNCTION AllocMem LIBRARY
DECLARE FUNCTION GetMsg LIBRARY
LIBRARY "t&t2:bmaps/intuition.library"
DECLARE FUNCTION OpenWindow LIBRARY
GADGETDOWN = 32&

```

```

GADGETUP      = 64&¶
CLOSEW       = 512&¶
¶
MList        = 0&¶
Info         = 1¶
¶
MainProgram:¶
GOSUB OpenAll¶
' Main part¶
MainLoop:¶
  IF Info = 1 THEN¶
    IntuiMsg = GetMsg(UserPort)¶
    IF IntuiMsg > 0 THEN GOSUB IntuitionMsg¶
    GOTO MainLoop¶
  END IF ¶
¶
GOSUB CloseAll¶
¶
END¶
¶
OpenAll:¶
  Border Edge, 0, 0, 3, 90, 15¶
  TestTxt$ = "Test-Text"¶
  IntuiText Text, 2, 10, 2, TestTxt$, 0& ¶
  GadgetDef Gadget, 0&, 50, 50, 90, 15, 0, 1, 1, Edge, Text, 0&,
1¶
  Title$ = "My first complete gadget"¶
  WinDef NWindow, 100, 100, 460, 100, 32+64+512&, 15&+4096&,
Gadget, Title$¶
  WinBase = OpenWindow(NWindow)¶
  IF WinBase = 0 THEN ERROR 7¶
  RastPort = PEEKL(WinBase+50)¶
  UserPort = PEEKL(WinBase+86)¶
RETURN¶
¶
CloseAll:¶
  CALL CloseWindow(WinBase)¶
  CALL UnDef¶
RETURN¶
¶
IntuitionMsg: ¶
  MsgTyp = PEEKL(IntuiMsg+20)¶
  Item = PEEKL(IntuiMsg+28)¶
  GadgetNr% = PEEK(Item+39)¶
  CALL ReplyMsg(IntuiMsg)¶
¶
  IF (MsgTyp = GADGETDOWN) THEN¶
    'immediately activated¶
    PRINT "DOWN Gadget-Nr.:";GadgetNr%¶
  END IF¶
¶
  IF (MsgTyp = GADGETUP) THEN¶
    'verify mode¶
    PRINT "UP Gadget-Nr.:";GadgetNr%¶
  END IF¶
¶
¶

```

```

IF (MsgTyp = CLOSEW) THEN¶
  'System-Gadget Window close¶
  PRINT"CLOSE WINDOW"¶
  Info = 0¶
END IF¶
RETURN¶
'-----¶
¶
SUB DefChip(Buffer,Size)STATIC¶
  SHARED MList¶
  Size=Size+8¶
  Buffer=AllocMem(Size,65538&)¶
  IF Buffer>0 THEN¶
    POKEL Buffer,MList¶
    POKEL Buffer+4,Size¶
    MList=Buffer¶
    Buffer=Buffer+8¶
  ELSE¶
    ERROR 7¶
  END IF¶
END SUB¶
SUB UnDef STATIC¶
  SHARED MList¶
undef.loop:¶
  IF MList>0 THEN¶
    Address = PEEKL(MList)¶
    ListSize = PEEKL(MList+4)¶
    FreeMem MList, ListSize¶
    MList = Address¶
    GOTO undef.loop¶
  END IF¶
END SUB ¶
SUB WinDef(bs, x%, y%, b%, h%, IDCMP, f, gad, T$) STATIC¶
  Size = 48+LEN(T$)+1¶
  DefChip bs,Size¶
  POKEW bs ,x% ' Left Corner¶
  POKEW bs+ 2,y% ' Top Corner¶
  POKEW bs+ 4,b% ' Width¶
  POKEW bs+ 6,h% ' Height¶
  POKEW bs+ 8,65535& ' Detail- BlockPen¶
  POKEL bs+10,IDCMP ' IDCMP Flags¶
  POKEW bs+14,f ' Flags¶
  POKEW bs+18,gad ' First Gadget¶
  POKEW bs+26,bs+48 ' Title¶
  POKEW bs+46,1 ' Screen Type ¶
  FOR i%=1 TO LEN(T$)¶
    POKE bs+47+i%,ASC(MID$(T$,i%,1))¶
  NEXT¶
END SUB¶
SUB GadgetDef(bs, nx, x%, y%, b%, h%, f%, a%, T%, i, txt, si,
n%) STATIC¶
  DefChip bs,44& ' Gadget-Structure length¶
  POKEW bs ,nx '*NextGadget¶
  POKEW bs+ 4,x% ' LeftEdge¶
  POKEW bs+ 6,y% ' TopEdge¶
  POKEW bs+ 8,b% ' Width¶

```

```

        POKEW bs+10,h%      ' Height¶
        POKEW bs+12,f%      ' Flags¶
        POKEW bs+14,a%      ' Activation¶
        POKEW bs+16,T%      ' GadgetType¶
        POKEW bs+18,i       ' GadgetRender¶
        POKEW bs+26,txt     '*GadgetText¶
        POKEW bs+34,si      ' SpecialInfo¶
        POKEW bs+38,n%      ' GadgetID¶
    END SUB¶
¶
SUB IntuiText(bs, c1%, x%, y%, T$, nx) STATIC¶
    Size=20+LEN(T$)+1      ' Structure length+ Text length+
    Nullbyte¶
    DefChip bs,Size¶
    POKE  bs  ,c1%         ' FrontPen¶
    POKE  bs+ 2,1         ' DrawMode¶
    POKEW bs+ 4,x%        ' Left corner¶
    POKEW bs+ 6,y%        ' Top corner¶
    POKEW bs+12,bs+20     ' IText¶
    POKEW bs+16,nx        ' NextText¶
    FOR i%=1 TO LEN(T$)¶
        POKE bs+19+i%,ASC(MID$(T$,i%,1))¶
    NEXT¶
END SUB¶
¶
SUB Border(bs, x%, y%, c%, b%, h%) STATIC¶
    DefChip bs,48&        ' Structure length+ Coordinate table¶
    POKEW bs  ,x%         ' Left corner¶
    POKEW bs+2,y%         ' Top corner¶
    POKE  bs+4,c%         ' FrontPen¶
    POKE  bs+7,8          ' Count¶
    POKEW bs+8,bs+16     '*XY¶
    FOR i%=0 TO 1¶
        POKEW bs+22+i%*4,h%-1¶
        POKEW bs+24+i%*4,b%-1¶
        POKEW bs+32+i%*4,1¶
        POKEW bs+38+i%*4,h%-1¶
        POKEW bs+40+i%*4,b%-2¶
    NEXT¶
END SUB¶

```

---

#### 4.1.4 User-friendly gadgets

Let's define four gadgets that are used in many programs. The gadgets we will define are OK, Cancel, Reset and Undo—these should all look familiar to you. OK would be used to continue a program, and Cancel would stop a function. Reset could be programmed to reset variables to their original status. Undo could be coded to reset the variable that was changed last.

We'll do this in a specific order. First we'll show you the gadget, text and border definitions. Then we'll list the reading routines with empty subroutines. Selecting one of the four gadgets calls one of the subroutines.

#### The gadget definitions:

```

Border Bord, -1, -1, 1,67,14¶
IntuiText OKTxt, 1, 26, 2, "OK", 0&¶
IntuiText CancelTxt, 1, 10, 2, "Cancel", 0&¶
IntuiText ResetTxt, 1, 14, 2, "Reset", 0&¶
IntuiText UndoTxt, 1, 20, 2, "Undo", 0&¶
GadgetDef UndoGad, 0&, 380, 52, 65, 12, 0, 1, 1, Bord,
UndoTxt, 0&, 1¶
GadgetDef ResetGad, UndoGad, 380, 68, 65, 12, 0, 1, 1, Bord,
ResetTxt, 0&, 2 ¶
GadgetDef OKGad, ResetGad, 380, 84, 65, 12, 0, 1, 1, Bord,
OKTxt, 0&, 3¶
GadgetDef CancGad, OKGad, 380, 100, 65, 12, 0, 1, 1, Bord,
CancelTxt, 0&, 4 ¶
Title$ = "An example of four user friendly Gadgets"¶
WinDef NWindow, 100, 50, 460, 150, 32+64+512&, 15&+4096&,
CancGad, Title$¶

```

#### The reading routines:

```

'The check routines:¶
IF (MsgType = GADGETDOWN) THEN¶
  'activation¶
  PRINT "DOWN Gadget-Nr.:";GadgetNr%¶
END IF¶
¶
IF (MsgType = GADGETUP) THEN¶
  'rem verify mode¶
  PRINT "UP Gadget-Nr.:";GadgetNr%¶
  IF GadgetNr% = 1 THEN¶
    GOSUB Undo ' put in old value¶
  END IF¶
  IF GadgetNr% = 2 THEN¶
    GOSUB ResetRoutine 'all values back to original¶
  END IF¶
  IF GadgetNr% = 3 THEN¶
    GOSUB Ok ' end value entry¶
  END IF¶
  IF GadgetNr% = 4 THEN¶
    GOSUB Cancel ' interrupt value entry¶
  END IF¶
END IF¶
¶
IF (MsgType = CLOSEW) THEN¶
  'close system Gadget window¶
  PRINT "CLOSE WINDOW"¶
  Info = 0¶
END IF¶
¶

```

```

RETURN
Undo:
    PRINT "The UNDO gadget was selected"
RETURN
ResetRoutine:
    PRINT "The RESET gadget was selected"
RETURN
Ok:
    PRINT "The OK gadget was selected"
RETURN
Cancel:
    PRINT "The CANCEL gadget was selected"
RETURN

```

Here is a listing of the complete program including the new gadget definitions:

```

*****
'*
'* Friendly-Gadgets with Intuition*
'* ----- *
'*
'* Author : Wolf-Gideon Bleek *
'* Date : May 23 '88 *
'* Name : Friendly-Gadgets *
'* Version: 1.2 *
'* System : V1.2 & V1.3 *
'*
'*
*****
OPTION BASE 1
DEFNG a-z
' Bmaps located on disk named T&T2,yours may differ
LIBRARY "t&t2:bmaps/exec.library"
DECLARE FUNCTION AllocMem LIBRARY
DECLARE FUNCTION GetMsg LIBRARY
LIBRARY "t&t2:bmaps/intuition.library"
DECLARE FUNCTION OpenWindow LIBRARY
GADGETDOWN = 32
GADGETUP = 64
CLOSEW = 512
MList = 0
Info = 1
MainProgramm:
GOSUB OpenAll
' Main part
MainLoop:
IF Info = 1 THEN
    IntuiMsg = GetMsg(UserPort)
    IF IntuiMsg > 0 THEN GOSUB IntuitionMsg
    GOTO MainLoop

```

```

        END IF ¶
    ¶
    GOSUB CloseAll¶
    ¶
END¶
¶
OpenAll:¶
¶
    Border Bord, -1, -1, 1,67,14¶
    IntuiText OKTxt, 1, 26, 2, "OK", 0&¶
    IntuiText CancelTxt, 1, 10, 2, "Cancel", 0&¶
    IntuiText ResetTxt, 1, 14, 2, "Reset", 0&¶
    IntuiText UndoTxt, 1, 20, 2, "Undo", 0&¶
    GadgetDef UndoGad, 0&, 380, 52, 65, 12, 0, 1, 1, Bord,
UndoTxt, 0&, 1¶
    GadgetDef ResetGad, UndoGad, 380, 68, 65, 12, 0, 1, 1, Bord,
ResetTxt, 0&, 2 ¶
    GadgetDef OKGad, ResetGad, 380, 84, 65, 12, 0, 1, 1, Bord,
OKTxt, 0&, 3¶
    GadgetDef CancGad, OKGad, 380, 100, 65, 12, 0, 1, 1, Bord,
CancelTxt, 0&, 4 ¶
    Title$ = "An example of four user friendly Gadgets"¶
    WinDef NWindow, 100, 50, 460, 150, 32+64+512&, 15&+4096&,
CancGad, Title$¶
    WinBase = OpenWindow(NWindow)¶
    IF WinBase = 0 THEN ERROR 7¶
    RastPort = PEEKL(WinBase+50)¶
    UserPort = PEEKL(WinBase+86)¶
RETURN¶
¶
CloseAll:¶
    CALL CloseWindow(WinBase)¶
    CALL UnDef¶
RETURN¶
¶
IntuitionMsg: ¶
    MsgType = PEEKL(IntuiMsg+20)¶
    Item = PEEKL(IntuiMsg+28)¶
    GadgetNr% = PEEK(Item+39)¶
    CALL ReplyMsg(IntuiMsg)¶
'The check routines:¶
IF (MsgType = GADGETDOWN) THEN¶
    'activation¶
    PRINT "DOWN Gadget-Nr.:";GadgetNr%¶
END IF¶
¶
IF (MsgType = GADGETUP) THEN¶
    'rem verify mode¶
    PRINT "UP Gadget-Nr.:";GadgetNr%¶
    IF GadgetNr% = 1 THEN¶
        GOSUB Undo ' put in old value¶
    END IF¶
    IF GadgetNr% = 2 THEN¶
        GOSUB ResetRoutine 'all values back to original¶
    END IF¶
    IF GadgetNr% = 3 THEN¶

```

```

    GOSUB Ok ' end value entry
  END IF
  IF GadgetNr% = 4 THEN
    GOSUB Cancel ' interrupt value entry
  END IF
END IF
¶
IF (MsgType = CLOSEW) THEN
  'close system Gadget window
  PRINT "CLOSE WINDOW"
  Info = 0
END IF
¶
RETURN
¶
Undo:
  PRINT "The UNDO gadget was selected"
RETURN ¶
¶
ResetRoutine:
  PRINT "The RESET gadget was selected"
RETURN ¶
¶
Ok:
  PRINT "The OK gadget was selected"
RETURN ¶
¶
Cancel:
  PRINT "The CANCEL gadget was selected"
RETURN ¶
¶
'-----¶
¶
SUB DefChip(Buffer,Size)STATIC
  SHARED MList
  Size=Size+8
  Buffer=AllocMem(Size,65538)
  IF Buffer>0 THEN
    POKEL Buffer,MList
    POKEL Buffer+4,Size
    MList=Buffer
    Buffer=Buffer+8
  ELSE
    ERROR 7
  END IF
END SUB
SUB UnDef STATIC
  SHARED MList
undef.loop:
  IF MList>0 THEN
    Address = PEEKL(MList)
    ListSize = PEEKL(MList+4)
    FreeMem MList, ListSize
    MList = Address
    GOTO undef.loop
  END IF

```



```

END SUB      ¶
SUB WinDef(bs, x%, y%, b%, h%, IDCMP, f, gad, T$) STATIC¶
  Size = 48+LEN(T$)+1¶
  DefChip bs,Size¶
  POKEW bs ,x%      ' Left Corner¶
  POKEW bs+ 2,y%    ' Top Corner¶
  POKEW bs+ 4,b%    ' Width¶
  POKEW bs+ 6,h%    ' Height¶
  POKEW bs+ 8,65535& ' Detail- BlockPen¶
  POKEW bs+10,IDCMP ' IDCMP Flags¶
  POKEW bs+14,f     ' Flags¶
  POKEW bs+18,gad   ' First Gadget¶
  POKEW bs+26,bs+48 ' Title¶
  POKEW bs+46,1     ' Screen Type ¶
  FOR i%=1 TO LEN(T$)¶
    POKE bs+47+i%,ASC(MID$(T$,i%,1))¶
  NEXT¶
END SUB¶
SUB GadgetDef(bs, nx, x%, y%, b%, h%, f%, a%, T%, i, txt, si,
n%) STATIC¶
  DefChip bs,44&    ' Gadget-Structure length¶
  POKEW bs ,nx     '*NextGadget¶
  POKEW bs+ 4,x%   ' LeftEdge¶
  POKEW bs+ 6,y%   ' TopEdge¶
  POKEW bs+ 8,b%   ' Width¶
  POKEW bs+10,h%   ' Height¶
  POKEW bs+12,f%   ' Flags¶
  POKEW bs+14,a%   ' Activation¶
  POKEW bs+16,T%   ' GadgetType¶
  POKEW bs+18,i    ' GadgetRender¶
  POKEW bs+26,txt  '*GadgetText¶
  POKEW bs+34,si   ' SpecialInfo¶
  POKEW bs+38,n%   ' GadgetID¶
END SUB¶
¶
SUB IntuiText(bs, c1%, x%, y%, T$, nx) STATIC¶
  Size=20+LEN(T$)+1 ' Structure length+ Text length+
Nullbyte¶
  DefChip bs,Size¶
  POKE bs ,c1%     ' FrontPen¶
  POKE bs+ 2,1     ' DrawMode¶
  POKEW bs+ 4,x%   ' Left corner¶
  POKEW bs+ 6,y%   ' Top corner¶
  POKEW bs+12,bs+20 ' IText¶
  POKEW bs+16,nx   ' NextText¶
  FOR i%=1 TO LEN(T$)¶
    POKE bs+19+i%,ASC(MID$(T$,i%,1))¶
  NEXT¶
END SUB¶
¶
SUB Border(bs, x%, y%, c%, b%, h%) STATIC¶
  DefChip bs,48&    ' Structure length+ Coordinate
table¶
  POKEW bs ,x%     ' Left corner¶
  POKEW bs+2,y%    ' Top corner¶
  POKEW bs+4,c%    ' FrontPen¶

```

```

POKE bs+7,8      ' Count
POKEL bs+8,bs+16 ' *XY
FOR i%=0 TO 1
  POKEW bs+22+i%*4,h%-1
  POKEW bs+24+i%*4,b%-1
  POKEW bs+32+i%*4,1
  POKEW bs+38+i%*4,h%-1
  POKEW bs+40+i%*4,b%-2
NEXT
END SUB

```

### 4.1.5 Scrolling tables

A large number of selections or an undefined number of elements can be difficult to program using filled gadgets. Scrolling tables are a logical choice. A scrolling table allows you to see only a small part of the complete table. This portion can be moved up or down.

You've seen scrolling tables at work if you've ever selected the Change Printer gadget in the Preferences program. The Change Printer screen appears when you click Change Printer. This screen has a scrolling table in the upper right corner of the screen displaying available printer drivers. This type of table is used because of its flexibility—it doesn't matter how many printer drivers are on a disk, you can view and select all of them. You control the selection by clicking on the up and down arrows to the left of the printer list.

Our program will need to create a table to display the possible choices and two gadgets, one for each arrow. Clicking on the arrow gadgets alters the display of possible choices. The following listing defines our scrolling table with two arrow gadgets and three display areas:

```

OpenAll:
  Border Bord, -1, -1, 1, 200, 14
  Border Box, 0, -1, 1, 50, 21
  GadgetDef Higher, 0, 51, 60, 48, 18, 0, 1, 1, 0, 0, 0, 1
  GadgetDef Lower, Higher, 51, 80, 48, 18, 0, 1, 1, 0, 0, 0, 2
  Title$ = "Scrolling-Table"
  WinDef NWindow, 100, 50, 460, 150, 32+64+512,
15+4096, Lower, Title$
  WinBase = OpenWindow(NWindow)
  IF WinBase = 0 THEN ERROR 7
  RastPort = PEEKL(WinBase+50)
  UserPort = PEEKL(WinBase+86)
  DrawBorder RastPort, Bord, 100, 60
  DrawBorder RastPort, Bord, 100, 73
  DrawBorder RastPort, Bord, 100, 86
  DrawBorder RastPort, Box, 50, 60

```

```

DrawBorder RastPort, Box, 50&, 79&¶
¶
x = 50      : y = 60¶
'x-        : y- value¶
x(1) =17+x  : x(2) =16+y¶
x(3) =34+x  : x(4) =16+y¶
x(5) =34+x  : x(6) =10+y¶
x(7) =40+x  : x(8) =10+y¶
x(9) =25+x  : x(10)=2+y¶
x(11)=10+x  : x(12)=10+y¶
x(13)=17+x  : x(14)=10+y¶
x(15)=17+x  : x(16)=16+y¶
¶
Move RastPort, 17+50&, 16+60&¶
PolyDraw RastPort, 8&, VARPTR(x(1))¶
¶
y = 62¶
'y- Value¶
x(2) =20+y¶
x(4) =20+y¶
x(6) =26+y¶
x(8) =26+y¶
x(10)=34+y¶
x(12)=26+y¶
x(14)=26+y¶
x(16)=20+y¶
¶
Move RastPort, 17+50&, 20+62&¶
PolyDraw RastPort, 8&, VARPTR(x(1))¶
¶
FOR i = 1 TO 5¶
    READ Table$(i)¶
    IntuiText ITxt(i), 1, 0, 0, Table$(i), 0&¶
NEXT i¶
TabOut Active¶
RETURN¶
CloseAll:¶
    CALL CloseWindow(WinBase)¶
    CALL UnDef¶
RETURN¶

```

After the window opens and the gadgets are drawn. `DrawBorder()` draws the boxes that are to contain our choices. `PolyDraw()` uses the `graphics.library` to draw a polygon. This function allows coordinate tables to be created in only a few lines of code. Here we use the `Polydraw()` routine to draw the up and down arrow graphics next to the scrolling table.

Following that, `OpenAll` reads five texts from `DATA` statements, which will later be used in our table. The `TabOut` subroutine handles the output of our choices.

Below we've printed the entire program with this new output routine and the `DATA` statements:

```

*****
*                               *
* Scrolling-Table-Gadgets      *
* -----                      *
*                               *
* Author   : Wolf-Gideon Bleek *
* Date    : May 31 '88         *
* Name     : Scroll-Gadgets    *
* Version  : 1.2               *
* System   : V1.2 & V1.3      *
*                               *
*****
OPTION BASE 1
DEFNIG a-w
DEFINT x
' Bmaps located on disk named T&T2,yours may differ
LIBRARY "t&t2:bmaps/exec.library"
DECLARE FUNCTION AllocMem LIBRARY
DECLARE FUNCTION GetMsg LIBRARY
LIBRARY "t&t2:bmaps/intuition.library"
DECLARE FUNCTION OpenWindow LIBRARY
LIBRARY "t&t2:bmaps/graphics.library"
GADGETDOWN = 32
GADGETUP   = 64
CLOSEW     = 512
MList      = 0
Info       = 1
Active     = 2
DIM x(16)
DIM SHARED Table$(5), ITxt(5)
MainProgramm:
GOSUB OpenAll
' Main part
MainLoop:
IF Info = 1 THEN
    IntuiMsg = GetMsg(UserPort)
    IF IntuiMsg > 0 THEN GOSUB IntuitionMsg
    GOTO MainLoop
END IF
GOSUB CloseAll
END
OpenAll:
Border Bord, -1, -1, 1, 200, 14
Border Box, 0, -1, 1, 50, 21
GadgetDef Higher, 0, 51, 60, 48, 18, 0, 1, 1, 0, 0, 0, 1
GadgetDef Lower, Higher, 51, 80, 48, 18, 0, 1, 1, 0, 0, 0,
2
Title$ = "Scrolling-Table"
WinDef NWindow, 100, 50, 460, 150, 32+64+512, 15+4096,
Lower, Title$

```

```

WinBase = OpenWindow(NWindow)¶
IF WinBase = 0 THEN ERROR 7¶
RastPort = PEEKL(WinBase+50)¶
UserPort = PEEKL(WinBase+86)¶
DrawBorder RastPort, Bord, 100&, 60&¶
DrawBorder RastPort, Bord, 100&, 73&¶
DrawBorder RastPort, Bord, 100&, 86&¶
DrawBorder RastPort, Box, 50&, 60&¶
DrawBorder RastPort, Box, 50&, 79&¶
¶
x = 50      : y = 60¶
'x-        y- value¶
x(1) =17+x : x(2) =16+y¶
x(3) =34+x : x(4) =16+y¶
x(5) =34+x : x(6) =10+y¶
x(7) =40+x : x(8) =10+y¶
x(9) =25+x : x(10)=2+y¶
x(11)=10+x : x(12)=10+y¶
x(13)=17+x : x(14)=10+y¶
x(15)=17+x : x(16)=16+y¶
¶
Move RastPort, 17+50&, 16+60&¶
PolyDraw RastPort, 8&, VARPTR(x(1))¶
¶
y = 62¶
'y- Value¶
x(2) =20+y¶
x(4) =20+y¶
x(6) =26+y¶
x(8) =26+y¶
x(10)=34+y¶
x(12)=26+y¶
x(14)=26+y¶
x(16)=20+y¶
¶
Move RastPort, 17+50&, 20+62&¶
PolyDraw RastPort, 8&, VARPTR(x(1))¶
¶
FOR i = 1 TO 5¶
    READ Table$(i)¶
    IntuiText ITxt(i), 1, 0, 0, Table$(i), 0&¶
NEXT i¶
TabOut Active¶
RETURN¶
CloseAll:¶
    CALL CloseWindow(WinBase)¶
    CALL UnDef¶
RETURN¶
IntuitionMsg: ¶
    MsgTyp    = PEEKL(IntuiMsg+20)¶
    Item      = PEEKL(IntuiMsg+28)¶
    GadgetNr% = PEEK(Item+39)¶
    CALL ReplyMsg(IntuiMsg)¶
    ¶
    IF (MsgTyp = GADGETDOWN) THEN¶
        'activated¶

```

```

        PRINT "DOWN Gadget-Nr.:";GadgetNr%¶
    END IF¶
    ¶
    IF (MsgTyp = GADGETUP) THEN¶
        'verify mode¶
        PRINT "UP Gadget-Nr.:";GadgetNr%¶
        IF GadgetNr% = 1 AND Active<>4 THEN Active=Active+1
: TabOut(Active)¶
        IF GadgetNr% = 2 AND Active<>1 THEN Active=Active-1
: TabOut(Active)¶
    END IF¶
    ¶
    IF (MsgTyp = CLOSEW) THEN¶
        'System-Gadget Window closer¶
        PRINT "CLOSE WINDOW"¶
        Info = 0¶
    END IF¶
RETURN¶
'-----¶
SUB DefChip(Buffer,Size)STATIC¶
    SHARED MList¶
    Size=Size+8¶
    Buffer=AllocMem(Size,65538&)¶
    IF Buffer>0 THEN¶
        POKEL Buffer,MList¶
        POKEL Buffer+4,Size¶
        MList=Buffer¶
        Buffer=Buffer+8¶
    ELSE¶
        ERROR 7¶
    END IF¶
END SUB¶
SUB UnDef STATIC¶
    SHARED MList¶
undef.loop:¶
    IF MList>0 THEN¶
        Address = PEEKL(MList)¶
        ListSize = PEEKL(MList+4)¶
        FreeMem MList, ListSize¶
        MList = Address¶
        GOTO undef.loop¶
    END IF¶
END SUB ¶
SUB WinDef(bs, x%, y%, b%, h%, IDCMP, f, Gad, T$) STATIC¶
    Size = 48+LEN(T$)+1¶
    DefChip bs,Size¶
    POKEW bs ,x% ' LeftEdge¶
    POKEW bs+ 2,y% ' TopEdge¶
    POKEW bs+ 4,b% ' Width¶
    POKEW bs+ 6,h% ' Height¶
    POKEW bs+ 8,65535& ' Detail- BlockPen¶
    POKEL bs+10,IDCMP ' IDCMPFlags¶
    POKEL bs+14,f ' Flags¶
    POKEL bs+18,Gad ' FirstGadget¶
    POKEL bs+26,bs+48 ' Title¶
    POKEW bs+46,1 ' ScreenType ¶

```

```

FOR i%=1 TO LEN(T$)¶
    POKE bs+47+i%,ASC(MID$(T$,i%,1))¶
NEXT¶
END SUB¶
SUB GadgetDef(bs, nx, x%, y%, b%, h%, f%, a%, T%, i, txt, si,
n%) STATIC¶
    DefChip bs,44&      ' Gadget-Structure length¶
    POKEL bs      ,nx      '*NextGadget¶
    POKEW bs+ 4,x%      ' LeftEdge¶
    POKEW bs+ 6,y%      ' TopEdge¶
    POKEW bs+ 8,b%      ' Width¶
    POKEW bs+10,h%      ' Height¶
    POKEW bs+12,f%      ' Flags¶
    POKEW bs+14,a%      ' Activation¶
    POKEW bs+16,T%      ' GadgetType¶
    POKEL bs+18,i      ' GadgetRender¶
    POKEL bs+26,txt     '*GadgetText¶
    POKEL bs+34,si     ' SpecialInfo¶
    POKEW bs+38,n%      ' GadgetID¶
END SUB¶
SUB IntuiText(bs, c1%, x%, y%, T$, nx) STATIC¶
    Size=20+LEN(T$)+1  ' Structure length+ Text length+
Nullbyte¶
    DefChip bs,Size¶
    POKE bs      ,c1%      ' FrontPen¶
    POKE bs+ 2,1      ' DrawMode¶
    POKEW bs+ 4,x%      ' LeftEdge¶
    POKEW bs+ 6,y%      ' TopEdge¶
    POKEL bs+12,bs+20  ' IText¶
    POKEL bs+16,nx     ' NextText¶
    FOR i%=1 TO LEN(T$)¶
        POKE bs+19+i%,ASC(MID$(T$,i%,1))¶
    NEXT¶
END SUB¶
SUB Border(bs, x%, y%, C%, b%, h%) STATIC¶
    DefChip bs,48&      ' Structure length+ coordinate table¶
    POKEW bs      ,x%      ' LeftEdge¶
    POKEW bs+2,y%      ' TopEdge¶
    POKE bs+4,C%      ' FrontPen¶
    POKE bs+7,8      ' Count¶
    POKEL bs+8,bs+16  '*XY¶
    FOR i%=0 TO 1¶
        POKEW bs+22+i%*4,h%-1¶
        POKEW bs+24+i%*4,b%-1¶
        POKEW bs+32+i%*4,1¶
        POKEW bs+38+i%*4,h%-1¶
        POKEW bs+40+i%*4,b%-2¶
    NEXT¶
END SUB¶
'exchange returned in PropInfo¶
SUB STRINGINFO(bs,max%,buff$) STATIC¶
    IF LEN(buff$)>max% THEN¶
        nmax%=LEN(buff$)¶
    ELSE¶
        nmax%=max%¶
    END IF¶

```

```

IF (nmax% AND 1) THEN nmax%=nmax%+1
Size=36+2*(nmax%+4)
DefChip bs,Size
POKEL bs,bs+36
POKEL bs+4,bs+40+nmax%
POKEW bs+10,max%+1
IF buff$<>""THEN
  FOR i%=1 TO LEN(buff$)
    POKE bs+35+i%,ASC(MID$(buff$,i%,1))
  NEXT
END IF
END SUB
SUB TabOut(Active) STATIC
SHARED RastPort
  COLOR 0,0
  FOR i = 0 TO 2
    SetAPen RastPort, 0
    RectFill RastPort, 101&, 13*i+60&, 296&, 13*i+71&
  NEXT i
  COLOR 1,0
  FOR i = Active-1 TO Active+1
    IF i>0 AND i<5 THEN
      POKEW ITxt(i)+6, 62+(i-Active+1)*13
      PrintIText RastPort, ITxt(i), 110&, 0&
    END IF
  NEXT i
  SetDrMd RastPort, 2
  RectFill RastPort, 101&, 73&, 296&, 84&
  SetDrMd RastPort, 1
END SUB
DATA Scroll-Table
DATA Closer
DATA Table
DATA Gadget
DATA System gadgets

```



## 4.1.6 Proportional gadgets

Proportional gadgets (also called sliders) allow the user to enter values that change in a proportional manner. Setting screen colors in the Preferences program is a good example of proportional gadgetry. Each controller can accept a value between 0 and 15, with each number representing the intensity of a color from no intensity (0) to high intensity (15).

You could also add three string gadgets into which a number may be entered, but for this kind of selection a proportional gadget is much more convenient than a string gadget. You can change the red, green and blue values by selecting the knob in the desired proportional gadget and moving the mouse pointer to the left or right, thus moving the knob.

The `intuition.library` file provides help in programming proportional gadgets. The container in a proportional gadget is the region containing the knob. This container sets the borders in which the proportional gadget's knob may be moved. This movement can be in the horizontal direction, vertical direction or in both directions at the same time. The user defines the knob graphic, or the proportional gadget routine uses the default Intuition knob graphic. The container usually has a visible border to allow easy selection by the user.

The `Gadget` structure must be enlarged to contain a `PropInfo` structure. This structure has connections to the `SpecialInfo` pointer. For this we have a subprogram which places the parameters in memory.

```
SUB PropInfo(bs, Flags%, HPot%, VPot%, HBody%, VBody%)
STATIC
  DefChip bs, 22&
  POKEW bs, Flags%
  POKEW bs+ 2, HPot%
  POKEW bs+ 4, VPot%
  POKEW bs+ 6, HBody%
  POKEW bs+ 8, VBody%
END SUB
```

We should address the values we added to the `Info` structure first. `Flags` here define whether the knob should move horizontally (2) or vertically (4). The `autoknob` (1) flag informs Intuition that no graphic exists for the knob, and that Intuition should use the default knob graphic available from Intuition. `HPot` and `VPot` define the knob's position. 0 indicates the lower right axis while `&HFFFF` indicates the upper left axis. After the user moves the knob the new position can be read from here. `HBody` and `VBody` return the step increment of the knob. Both values are calculated as part of the whole

(&HFFFF). Intuition inserts all further values found in the structure—these values don't have to be defined by the program.

Autoknob graphically defines the knob. Autoknob requires an eight-byte memory area which contains the knob's position (X and Y coordinates) and width. If all four values are not set, the initialization routine sets them. We still need two more structures to complete our proportional gadget:

```
PropInfo PropI, 1+2, 0, 0, &HFFF, 0¶
IntuiText, Text, 2, -80, 2, "Mover:", 0&¶
DefChip Buffer, 8&¶
GadgetDef Gadget, 0&, 150, 30, 100, 10, 0, 1+2, 3, Buffer,
Text, PropI, 1¶
```

We can construct complete proportional gadgets from these few pieces of data. As an example, we have a listing that uses three such gadgets in a window. These three proportional gadgets allow the user to change the value of the corresponding color register when selected with the mouse.

```
*****¶
*¶
* Proportional-Gadgets ¶
* ----- ¶
*¶
* Author : Wolf-Gideon Bleek ¶
* Date :May 23 '88 ¶
* Name :Proportional-Gadgets ¶
* Version: 1.2 ¶
* System : V1.2 & V1.3 ¶
*¶
*****¶
OPTION BASE 1¶
DEFLNG a-z¶
¶
LIBRARY "T&T2:bmaps/exec.library"¶
DECLARE FUNCTION AllocMem LIBRARY¶
DECLARE FUNCTION GetMsg LIBRARY¶
LIBRARY "T&T2:bmaps/intuition.library"¶
DECLARE FUNCTION OpenWindow LIBRARY¶
GADGETDOWN = 32&¶
GADGETUP = 64&¶
CLOSEW = 512&¶
¶
MList = 0&¶
Info = 1¶
¶
MainProgramm:¶
GOSUB OpenAll¶
' Main part¶
MainLoop:¶
IF Info = 1 THEN¶
IntuiMsg = GetMsg(UserPort)¶
IF IntuiMsg > 0 THEN GOSUB IntuitionMsg¶
```

```

        GOTO MainLoop
    END IF
    GOSUB CloseAll
END
OpenAll:
    IntuiText RedTxt, 2, -80, 2, "Red", 0
    IntuiText GrnTxt, 2, -80, 2, "Green", 0
    IntuiText BluTxt, 2, -80, 2, "Blue", 0
    PropInfo Prop1, 1+2, 0, 0, &HFFF, 0
    PropInfo Prop2, 1+2, 0, 0, &HFFF, 0
    PropInfo Prop3, 1+2, 0, 0, &HFFF, 0
    DefChip Buffer(1), 8
    DefChip Buffer(2), 8
    DefChip Buffer(3), 8
    GadgetDef RedGad, 0, 150, 30, 114, 10, 0, 1+2, 3,
    Buffer(1), RedTxt, Prop1, 1
    GadgetDef GrnGad, RedGad, 150, 45, 114, 10, 0, 1+2, 3,
    Buffer(2), GrnTxt, Prop2, 2
    GadgetDef BluGad, GrnGad, 150, 60, 114, 10, 0, 1+2, 3,
    Buffer(3), BluTxt, Prop3, 3
    Title$ = "Color initialization"
    WinDef NWindow, 100, 50, 460, 150, 32+64+512&
    15&+4096&, BluGad, Title$
    WinBase = OpenWindow(NWindow)
    IF WinBase = 0 THEN ERROR 7
    RastPort = PEEKL(WinBase+50)
    UserPort = PEEKL(WinBase+86)
RETURN
CloseAll:
    CALL CloseWindow(WinBase)
    CALL UnDef
RETURN
IntuitionMsg:
    MsgTyp = PEEKL(IntuiMsg+20)
    Item = PEEKL(IntuiMsg+28)
    GadgetNr% = PEEK(Item+39)
    CALL ReplyMsg(IntuiMsg)
    IF (MsgTyp = GADGETDOWN) THEN
        'activated
        PRINT "DOWN Gadget-Nr.:";GadgetNr%
    END IF
    IF (MsgTyp = GADGETUP) THEN
        'verify mode
        PRINT "UP Gadget-Nr.:";GadgetNr%;
        PRINT " Pos:";PEEKW(Buffer(GadgetNr%))
        Red = PEEKW(Buffer(1))
        Grn = PEEKW(Buffer(2))
        Blu = PEEKW(Buffer(3))
        PALETTE 1, Red/100, Grn/100, Blu/100
    END IF

```

```

    ¶
    IF (MsgTyp = CLOSEW) THEN¶
        'System-Gadget Window closer¶
        PRINT "CLOSE WINDOW"¶
        Info = 0¶
    END IF¶
RETURN¶
'-----¶
SUB DefChip(Buffer,Size)STATIC¶
    SHARED MList¶
    Size=Size+8¶
    Buffer=AllocMem(Size,65538&)¶
    IF Buffer>0 THEN¶
        POKEL Buffer,MList¶
        POKEL Buffer+4,Size¶
        MList=Buffer¶
        Buffer=Buffer+8¶
    ELSE¶
        ERROR 7¶
    END IF¶
END SUB¶
SUB UnDef STATIC¶
    SHARED MList¶
undef.loop:¶
    IF MList>0 THEN¶
        Address = PEEKL(MList)¶
        ListSize = PEEKL(MList+4)¶
        FreeMem MList, ListSize¶
        MList = Address¶
        GOTO undef.loop¶
    END IF¶
END SUB ¶
SUB WinDef(bs, x%, y%, b%, h%, IDCMP, f, Gad, T$) STATIC¶
    Size = 48+LEN(T$)+1¶
    DefChip bs,Size¶
    POKEW bs ,x% ' Left corner¶
    POKEW bs+ 2,y% ' Top corner¶
    POKEW bs+ 4,b% ' Width¶
    POKEW bs+ 6,h% ' Height¶
    POKEW bs+ 8,65535& ' Detail- BlockPen¶
    POKEW bs+10,IDCMP ' IDCMP Flags¶
    POKEW bs+14,f ' Flags¶
    POKEW bs+18,Gad ' First Gadget¶
    POKEW bs+26,bs+48 ' Title¶
    POKEW bs+46,1 ' Screen Type ¶
    FOR i%=1 TO LEN(T$)¶
        POKE bs+47+i%,ASC(MID$(T$,i%,1))¶
    NEXT¶
END SUB¶
SUB GadgetDef(bs, nx, x%, y%, b%, h%, f%, a%, T%, i, Txt, si,
n%) STATIC¶
    DefChip bs,44& ' Gadget-Structure length¶
    POKEW bs ,nx '*NextGadget¶
    POKEW bs+ 4,x% ' Left corner¶
    POKEW bs+ 6,y% ' Top corner¶
    POKEW bs+ 8,b% ' Width¶

```

```

POKEW bs+10,h%      ' Height¶
POKEW bs+12,f%      ' Flags¶
POKEW bs+14,a%      ' Activation¶
POKEW bs+16,T%      ' GadgetType¶
POKEL bs+18,i       ' GadgetRender¶
POKEL bs+26,Txt     '*GadgetText¶
POKEL bs+34,si      ' SpecialInfo¶
POKEW bs+38,n%      ' GadgetID¶
END SUB¶
SUB IntuiText(bs, c1%, x%, y%, T$, nx) STATIC¶
  Size=20+LEN(T$)+1 ' IntuiText-Structure length + Text
length+ Nullbyte¶
  DefChip bs,Size¶
  POKE  bs  ,c1%      ' FrontPen¶
  POKE  bs+ 2,1      ' DrawMode¶
  POKEW bs+ 4,x%     ' Left corner¶
  POKEW bs+ 6,y%     ' Top corner¶
  POKEL bs+12,bs+20 ' IText¶
  POKEL bs+16,nx     ' NextText¶
  FOR i%=1 TO LEN(T$)¶
    POKE bs+19+i%,ASC(MID$(T$,i%,1))¶
  NEXT¶
END SUB¶
SUB Border(bs, x%, y%, c%, b%, h%) STATIC¶
  DefChip bs,48&    ' Border-Structure length+
coordinates¶
  POKEW bs  ,x%     ' Left corner¶
  POKEW bs+2,y%     ' Top corner¶
  POKE  bs+4,c%     ' FrontPen¶
  POKE  bs+7,8      ' Count¶
  POKEL bs+8,bs+16 '*XY¶
  FOR i%=0 TO 1¶
    POKEW bs+22+i%*4,h%-1¶
    POKEW bs+24+i%*4,b%-1¶
    POKEW bs+32+i%*4,1¶
    POKEW bs+38+i%*4,h%-1¶
    POKEW bs+40+i%*4,b%-2¶
  NEXT¶
END SUB¶
SUB PropInfo(bs, Flags%, HPot%, VPot%, HBody%, VBody%)
STATIC¶
  DefChip bs,22&¶
  POKEW bs  ,Flags%¶
  POKEW bs+ 2,HPot%¶
  POKEW bs+ 4,VPot%¶
  POKEW bs+ 6,HBody%¶
  POKEW bs+ 8,VBody%¶
END SUB¶
¶

```

*Looking  
toward the  
future*

From the information in this section, you should be able to develop many user-friendly programs. We have tried to develop procedures that allow easy access to the operating system, especially Intuition. All the programs presented here are in modular form. This makes it easy for you to add these modules to your own programs. This is done the following way:

Write each subprogram in a directory as an ASCII file (save "program\_name",A). Put comments listing the required parameters before each routine. When you need to use an Intuition call, then load the subprogram using Merge and put the Intuition call in the OpenAll subroutine.

---

## 4.2 Rubberbanding

Earlier in this chapter you learned about the most important elements of professional program design. You shouldn't be afraid of hunting for new ways to do things. Every new problem has a new solution.

This section discusses a function that you've used any number of times. The function is called *rubberbanding*. Rubberbanding occurs when you change the size of a window. *Intuition* lets you change a window's size by grabbing onto the sizing gadget at the lower right corner of most windows. This section, however, shows how to program rubberbanding in BASIC.

The trick is in creating lines in complement mode instead of simply drawing lines. Complement mode allows you to move a line or set of lines around on the screen without redrawing the background.

You'd normally use rubberbanding for determining window size on the screen. However, this process also makes it easier to draw rectangles in graphic programs.

---

### 4.2.1 Rectangles in rubberbanding

The purpose of the following program is to show you how this function is used in a program. You can adapt the mouse control techniques to your own applications.

When you start the program an empty window appears with a mouse pointer in it. Press and hold the left mouse button from any position in the window and drag the pointer down and to the right. A rubberbanded rectangle appears and changes size as you move the pointer. When you release the left mouse button, the rectangle stays on the screen and changes to character color 1.

```
' Drawing Rectangles with Rubberbanding¶
'¶
' by Wgb in June '87¶
'¶
¶
LIBRARY "T&T2:bmaps/graphics.library"¶
¶
ON MOUSE GOSUB SetPoint¶
MOUSE ON¶
¶
```

```

WHILE INKEY$<>" "¶
  SLEEP¶
WEND¶
¶
MOUSE OFF¶
END¶
¶
¶
SetPoint:¶
¶
MStat=MOUSE(0)¶
IF MStat<>-1 THEN RETURN¶
¶
xStart=MOUSE(3)¶
yStart=MOUSE(4)¶
CALL SetDrMd&(WINDOW(8),2)¶
¶
NewPosition:¶
¶
mx=MOUSE(1)¶
my=MOUSE(2)¶
¶
LINE (xStart,yStart)-(mx,my),,b¶
¶
WHILE MOUSE(0)=-1¶
  IF mx<>MOUSE(1) OR my<>MOUSE(2) THEN¶
    LINE (xStart,yStart)-(mx,my),,b¶
    GOTO NewPosition¶
  END IF¶
WEND ¶
¶
CALL SetDrMd&(WINDOW(8),1)¶
LINE (xStart,yStart)-(mx,my),,b¶
RETURN¶
¶

```

**Variables**

MStat	mouse status
mx, my	mouse coordinates
xStart	starting X-coordinate of rectangle
yStart	starting Y-coordinate of rectangle

**Program description**

The `graphics.library` opens. The program draws the guidelines in complement mode and this library file transfers the necessary graphic routines to the program.

The `SetPoint` subroutine sets the mouse reading at the beginning of the program. The program waits for a keypress. The keypress turns off and ends the mouse reading routine.

The mouse reader is the central point of the program; take a good look at those program lines. The mouse status goes into a variable. The subroutine exits when it notes that the user hasn't pressed the left mouse key. Otherwise, the program marks the pointer position as the



starting value, and the drawing mode changes to complement mode. The routine then draws the rectangle and waits for you to move the mouse. The program then deletes the rectangle and redraws it to fit the new mouse position.

When the user releases the left mouse button, the program exits the loop. The program then returns to normal character mode and displays the final rectangle.

### 4.2.2 Creating shapes

Rubberbanding can be used for much more than changing window sizes and drawing rectangles. This program draws lines between two points selected by the user. This routine also uses rubberbanding. When you start the program and press the left mouse button, you'll soon see two pixels connected by a rubberband.

```
' Connections with Rubberbanding
'
' by Wgb in June '87
'
LIBRARY "T&T2:bmaps/graphics.library"
BaseGraphic:
LINE (100,180)-(540,180)
FOR i=100 TO 540
  x=(i-100)/2.444444
  y=SIN(x*3.1415/180)*100
  LINE -(i,180-y)
NEXT i
ON MOUSE GOSUB SetPoint
MOUSE ON
WHILE INKEY$<>" "
  SLEEP
WEND
MOUSE OFF
END
SetPoint:
MStat=MOUSE(0)
IF MStat<>-1 THEN RETURN
```

```

CALL SetDrMd&(WINDOW(8),2)¶
¶
NewPosition:¶
¶
mx=MOUSE(1)¶
CALL Connect(mx)¶
¶
WHILE MOUSE(0)=-1¶
  IF mx<>MOUSE(1) THEN¶
    CALL Connect(mx)¶
    GOTO NewPosition¶
  END IF¶
WEND ¶
¶
CALL SetDrMd&(WINDOW(8),1)¶
CALL Connect(mx)¶
RETURN¶
¶
SUB Connect (x) STATIC¶
¶
IF x<100 THEN x=100¶
IF x>540 THEN x=540¶
¶
xw=(x-100)/2.444444¶
yw=SIN(xw*3.1415/180)*100¶
¶
LINE (100,180)-(x,180-yw)¶
LINE -(540,180)¶
PSET (x,180-yw)¶
¶
END SUB¶
¶

```

**Variables**

MStat	mouse status
i	floating variable
mx	mouse position
x, y	graphic coordinates
xw, yw	coordinates in SUB

**Program description**

The basic design is similar to the first listing. There is an additional routine for the banding based on a short sine equation followed by the same delay loop.

The major changes appear in the SUB programs. The mouse control routine now checks the X-position of the pointer. This position controls the call of a subroutine. The routine then draws the connecting line, while reading the pointer's X-movement. Like the previous program, the old lines are deleted and redrawn at the new position.

Try the program. The X-value goes into a specific range because not all X-coordinates have a graphic equivalent. The program then computes the coordinates and draws the line.

### 4.2.3 Object positioning

This last routine came from the idea of a drawing program for two-dimensional grid graphics. When you draw multiple objects in such a program, you may find that you run out of room on the screen. The simplest way to move objects would be to select them with the mouse pointer and drag the objects to new screen locations. The following program performs a function similar to this. First it computes the imaginary corner points of a circle. Obviously circles do not have corners, but using imaginary points makes the coding simple.

The circle is displayed as long as you press and hold the left mouse button; it disappears when you release the left mouse button.

```
' Objects with Rubberbanding¶
'¶
' by Wgb in June '87¶
'¶
¶
LIBRARY "T&T2:bmaps/graphics.library"¶
¶
ObjectDefinition:¶
¶
DIM SHARED Ob%(10,1)¶
Pi=3.141593¶
¶
FOR i=0 TO 360 STEP 36¶
  x=COS(i*Pi/180)*30¶
  y=SIN(i*Pi/180)*15¶
  Ob%(i/36,0)=x¶
  Ob%(i/36,1)=y¶
NEXT i¶
¶
¶
ON MOUSE GOSUB SetObject¶
MOUSE ON¶
¶
  WHILE INKEY$<>" "¶
    SLEEP¶
  WEND¶
¶
MOUSE OFF¶
END¶
¶
¶
SetObject:¶
¶
MStat=MOUSE(0)¶
IF MStat<>-1 THEN RETURN¶
¶
```

```

CALL SetDrMd&(WINDOW(8),2)¶
¶
NewPosition:¶
¶
mx=MOUSE(1)¶
my=MOUSE(2)¶
CALL DrawObject(mx,my)¶
¶
WHILE MOUSE(0)=-1¶
  IF mx<>MOUSE(1) OR my<>MOUSE(2) THEN¶
    CALL DrawObject(mx,my)¶
    GOTO NewPosition¶
  END IF¶
WEND ¶
¶
CALL SetDrMd&(WINDOW(8),1)¶
CALL DrawObject(mx,my)¶
¶
RETURN¶
¶
SUB DrawObject(x,y) STATIC¶
¶
PSET (Ob%(0,0)+x,Ob%(0,1)+y)¶
¶
FOR i=1 TO 10¶
  LINE -(Ob%(i,0)+x,Ob%(i,1)+y) ¶
NEXT i¶
¶
LINE -(Ob%(10,0)+x,Ob%(10,1)+y)¶
¶
END SUB¶

```

**Arrays**            Ob            circle point array

**Variables**

MStat	mouse status
Pi	3.141593
i	floating variable
mx, my	mouse coordinates
x, y	circle coordinates

**Program description**    The graphics.library opens and the Ob% array reads the X- and Y-coordinates. A loop computes the 11-pixel offset from the circle's "corner" to the circle's border. The rest of the program should look familiar to you.

The most important changes occur in the mouse reader routine. If the left mouse button was not pressed, the mouse reader branches back to the main program. However, if you did press the left mouse button, the the program sets the drawing mode and draws the object at the current position.

Then the program goes into a delay loop again, and exits when you release the left mouse button. The program branches again to the point before the loop where you change the mouse position. This is because the grid must be erased and the object drawn at its new position.

The subroutine for drawing the object takes the 11 coordinate pairs from the `Ob%` array. The first point is drawn, then the others through `LINE` commands. All points drawn join to form a circle.

---

## 4.3 Status lines & animation

Invisible status lines are part of a new screen organization which offer you many new special effects. For example, it allows you to create a color bar that lets you move the entire screen up and down. This bar has its own foreground and background colors, and it can also contain movable text. With the same program, it's possible to fill the screen background with a pattern or graphic if you wish. This pattern stays intact, even when you use PRINT commands, draw or scroll. You can even scroll your background independently of the foreground drawing.

You need only two applications for doing all this. Before listing the program, let's look at the individual SUB programs. The first is `CreateStatus`. This command turns on the new screen organization. The next is `Copy`. This command copies the current screen contents in the background. This is where only colors 0 and 1 appear (only one bitplane is available in background memory). Once the screen contents are copied, a new background pattern appears. You can clear the "normal" screen with the `CLS` command; the background pattern stays on. The closing is the `Move` SUB program. This command scrolls the background pattern up or down. The command syntax needs two values:

```
Move dir%,speed%
```

The `dir%` variable gives the number of pixels the background graphic should move. A positive value scrolls the graphic down; a negative number scrolls it up. The `speed%` variable sets the scrolling speed. Zero is the top speed. Here's a sample call:

```
Move 100, 40
```

This call moves the background 100 pixels down at a delay rate of 40.

As you'll see when you test the following programs, the `Move` command does more than just move the background. When you move the background graphic up or down, the opposite side of the page stays visible. The routine acts as an endless scroll routine, which can produce some very interesting effects. Try this version of the `Move` command:

```
Move 0,0
```

This call appears to do nothing (moving the background graphic 0 pixels), but it has a special function: It clears the background graphic.

The `EndStatus` SUB reactivates the normal screen display. This command must be at the end of your programs to remove the

CreateStatus command's effects. This command also returns the entire user memory range.

```
'#####
'#                               #
'# Program: Dual BitMap         #
'# Author:  tob                 #
'# Date:    May 8, 1987         #
'# Version: 2.0                 #
'#                               #
'#####

DECLARE FUNCTION AllocMem& LIBRARY
DECLARE FUNCTION BltBitMap% LIBRARY

LIBRARY "T&T2:bmaps/graphics.library"
LIBRARY "T&T2:bmaps/intuition.library"
LIBRARY "T&T2:bmaps/exec.library"

demo:  '* Open Screen
        SCREEN 1, 640, 240, 3, 2
        WINDOW 1,"DualBitmap", (0,0)-(610,217),1,1
        WINDOW OUTPUT 1
        #
        '* Draw Circle
        CreateStatus
        LINE (0,0) - (620,10),,bf
        Copy
        CLS
        #
        '* Color
        PALETTE 1,1,1,1
        PALETTE 4,1,0,0
        PALETTE 5,1,.5,.5
        #
        GOSUB text
        #
        '* Move Scroll Circle
        Move 166, 0
        PRINT "Please Press any Key.":PRINT " "
        WHILE INKEY$ = "": WEND
        Move 0,0
        #
        '* 2nd Experiment
        CLS
        CIRCLE (140,100), 120, 1
        CIRCLE (140,100), 100, 1
        CIRCLE (140,100), 80, 1
        CIRCLE (140,100), 50, 1
        CIRCLE (140,100), 25, 1
        PAINT (250,100), 1, 1
        PAINT (210,100), 1, 1
        PAINT (140,100), 1, 1
        Copy
        CLS
        #
```

```

* Color
PALETTE 0,0,0,1
PALETTE 1,1,0,0
PALETTE 4,0,1,1
PALETTE 5,0,1,0

GOSUB text

LOCATE 22,1
PRINT "Please Press any Key."
  WHILE INKEY$ = ""
    Move -3, 0
  WEND

* 3rd Experiment
Move 0,0
CLS
WIDTH "scrn:", 85
text$ = "* Amiga Tricks and Tips"
FOR loop% = 1 TO 56
  LOCATE loop%,5
  PRINT text$
NEXT loop%

Copy
CLS

* Color
PALETTE 0,.1,.1,.8
PALETTE 1,1,1,1
PALETTE 4,.3,.3,.3
PALETTE 5,1,1,1

GOSUB text

* Animation
WHILE INKEY$ = ""
  Move 1,0
WEND

Move 0,0

EndStatus
WINDOW 1,"Dual-Bitmap",,-1
SCREEN CLOSE 1

LIBRARY CLOSE
END
text: * Print Text
CLS
LOCATE 5,1
PRINT "This is the new 'Dual-Bitmap'."
LOCATE 6,1
PRINT "You can control two bitplanes,"
LOCATE 7,1
PRINT "one completely independent of"

```



```

LOCATE 8,1¶
PRINT "the display."¶
LOCATE 9,1¶
PRINT "The level helps"¶
LOCATE 10,1¶
PRINT "determine the color"¶
LOCATE 11,1¶
PRINT "registers using the bitplanes:"¶
LOCATE 12,1¶
PRINT "Level          Color register"¶
LOCATE 13,1¶
PRINT "-----"¶
LOCATE 14,1¶
PRINT "  1          not functional"¶
LOCATE 15,1¶
PRINT "  2          2, 3"¶
LOCATE 16,1¶
PRINT "  3          4, 5"¶
LOCATE 17,1¶
PRINT "  4          8, 9"¶
LOCATE 18,1¶
PRINT "  5          16, 17"¶
RETURN¶
¶
SUB Copy STATIC¶
  SHARED bitmap&, bitmap2&¶
  l% = PEEK (WINDOW(7) + 54)¶
  r% = PEEK (WINDOW(7) + 56)¶
  u% = PEEK (WINDOW(7) + 57)¶
  o% = PEEK (WINDOW(7) + 55)¶
  w% = PEEKW (WINDOW(7) + 8) - r% - l%¶
  h% = PEEKW (WINDOW(7) + 10) - u% - o%¶
  x% = PEEKW (WINDOW(7) + 4) + l%¶
  y% = PEEKW (WINDOW(7) + 6) + o%¶
  ¶
  plc% = BltBitMap% (bitmap&, x%, y%, bitmap2&, x%, y%,
w%, h%, 200, 255, 0)¶
END SUB¶
¶
SUB Move (dir%, speed%) STATIC¶
  SHARED bitmap2&¶
  l% = PEEK (WINDOW(7) + 54)¶
  r% = PEEK (WINDOW(7) + 56)¶
  u% = PEEK (WINDOW(7) + 57)¶
  o% = PEEK (WINDOW(7) + 55)¶
  w% = PEEKW (WINDOW(7) + 8) - r% - l%¶
  h% = PEEKW (WINDOW(7) + 10) - u% - o%¶
  x% = PEEKW (WINDOW(7) + 4) + l%¶
  y% = PEEKW (WINDOW(7) + 6) + o%¶
  ¶
  spd% = 10*speed%¶
  u% = y% + h% - 2¶
  IF dir% = 0 THEN¶
    bitplane& = PEEKL (bitmap2& + 8)¶
    m% = PEEKW (bitmap2&)¶
    n% = PEEKW (bitmap2& + 2)¶

```

```

        s& = (m%*n%)¶
        CALL BltClear(bitplane&, s&, 0)¶
        EXIT SUB¶
    END IF¶
    FOR z% = 1 TO ABS(dir%)¶
        IF dir% > 0 THEN¶
            plc% = BltBitMap% (bitmap2&, x%, u%, bitmap2&,
x%, y%, w%, 1, 200, 255, 0)¶
            plc% = BltBitMap% (bitmap2&, x%, y%, bitmap2&,
x%, y% + 1, w%, h% - 1, 200, 255, 0)¶
        ELSE¶
            plc% = BltBitMap% (bitmap2&, x%, y%, bitmap2&,
x%, u%, w%, 1, 200, 255, 0)¶
            plc% = BltBitMap% (bitmap2&, x%, y% + 1,
bitmap2&, x%, y%, w%, h% - 1, 200, 255, 0)¶
        END IF¶
        FOR del% = 1 TO spd%: NEXT del%¶
    NEXT z%¶
END SUB¶
¶
SUB EndStatus STATIC¶
    SHARED rasInfo&¶
    rasInfo2& = PEEKL (rasInfo&)¶
    bitmap& = PEEKL (rasInfo& + 4)¶
    bitmap2& = PEEKL (rasInfo2& + 4)¶
    level% = PEEK (bitmap& + 5)¶
    POKEL bitmap& + 8 + level%*4, PEEKL (bitmap2& + 8)¶
    POKE bitmap& + 5, level% + 1¶
    POKEL rasInfo&, 0¶
    CALL FreeMem(rasInfo2&, 10)¶
    CALL FreeMem(bitmap2&, 40)¶
END SUB¶
¶
SUB CreateStatus STATIC¶
    SHARED rasInfo&, bitmap&, bitmap2&¶
    '* Get System Addresses¶
    wind& = WINDOW(7)¶
    rastport& = WINDOW(8)¶
    bitmap& = PEEKL (rastport& + 4)¶
    level% = PEEK (bitmap& + 5)¶
    scr& = PEEKL (wind& + 46)¶
    vp& = PEEKL (scr& + 44)¶
    rasInfo& = PEEK (vp& + 36)¶
    ¶
    IF level% < 2 THEN¶
        PRINT "A Screen with 2 levels is needed!"¶
        EXIT SUB¶
    END IF¶
    ¶
    '* Establish Structure¶
    opt& = 2^1 + 2^16¶
    rasInfo2& = AllocMem&(10, opt&)¶
    IF rasInfo2& = 0 THEN ERROR 7¶
    bitmap2& = AllocMem&(40, opt&)¶
    IF bitmap2& = 0 THEN¶
        CALL FreeMem(rasInfo2&, 10)¶

```

```

        ERROR 7¶
    END IF¶
    ¶
    CALL CopyMem(rasInfo&, rasInfo2&, 10)¶
    CALL CopyMem(bitmap&, bitmap2&, 40)¶
    ¶
    POKE  bitmap&  + 5, level% - 1¶
    POKE  bitmap2& + 5, 1¶
    POKEL bitmap2& + 8, PEEKL (bitmap& + 4 + 4*level%)¶
    POKEL bitmap&  + 4 + 4*level%, 0¶
    POKEL rasInfo2& + 4, bitmap2&¶
    ¶
    POKEL rasInfo& , rasInfo2&¶
END SUB¶

```

***Program  
description***

Once you enter this program, be sure to save it to diskette before you try running it for the first time. The first experiment displays a red bar. It moves around the text page, and can pass behind text in the window. The second experiment is similar. Transparent circles move around on the screen. The third experiment fills the background with a text pattern.

Now we'll discuss the technical basics of what you're doing. The Amiga recognizes a special mode called the *Dual Playfield* mode. This mode can divide individual bitplanes in screen memory into two groups, and make these two groups independent of each other. These two groups are like independent screens; each one is visible through the other in the background. This graphic mode isn't used in these examples. Only one item which is actually counted as *Dual Playfield* mode is used. The *RasInfo* data structure, which assigns a pointer in the viewport to the selected screen, lets you detach individual bitplanes from each other. The *RasInfo* structure connects one of its own bit-map structures contained in the disconnected bitmap.

The *CreateStatus* SUB reads the corresponding system addresses and tests for a screen with a depth of 2 or more. The system can't use the screen if it has only one bitplane. Two *Bitmap* and *RasInfo* structures are created if two or more bitplanes are available (*AllocMem()* allocates the needed memory). The original bitplane takes on the named bitplane (incremented in depth by 1). The second bitmap receives a depth number of 1. It's inserted into the first bitmap. Finally, a pointer to the new bitmap must be inserted in the *RasInfo* structure.

The *Copy* SUB copies the contents of the first bitplane (colors 0 and 1) to the coupled bitplane (*bitplane2&*). Only window contents are copied. You might think that it would be easier to copy the entire screen contents. However, the window borders would also be copied. Using the *Move* routine under these conditions would scroll the window borders as well as the background. This probably would result in a system error. If you reduced the size of your window after the copy

process, the background would keep its full size. You can avoid this by either not changing window size or clearing the background with `Move 0,0`.

The `Move SUB` scrolls the background up or down. This affects only the window contents, nothing else. The system handles this as an endless scroll routine, which can scroll one line of pixels up or down at a time. Larger increments move through multiple looping.

Calling `Move 0,0` activates the `BitClear()` function, which clears the entire background (not just the window's contents). This also clears any window section hidden beyond the edges of the screen.

`EndStatus` restores the original bitmap and clears the dual structures.

Now that you have some background information, let's take a closer look at the program itself. When mixing bitplanes, the user doesn't have eight colors with a screen that has a depth of 3 planes (normally  $2^3=8$ ). Instead, since two of those planes are merged, only four colors are available ( $2^2=4$ ). However, you still get 8 colors in combination with the background. A screen with a depth of 3 appears in background memory with the color of color register 4. This command sets the color of the background graphic:

```
PALETTE 4, 1, .6, .9
```

The combined color between background graphic and normal foreground drawing color comes from color register 5. This command sets the color shared by the background and normal foreground:

```
PALETTE 5, 1, 1, .7
```

The color selections are up to you—you can get some nice effects. For example, you can combine the normal background and color register 4 to set a combined shade of red:

```
PALETTE 0, 0, 0, 0
PALETTE 4, 0, 0, 0
PALETTE 5, 1, 0, 0
```

The result: The background is invisible. When the foreground color runs into the background (through `PRINT`, etc.), the text turns red.

Another is the transparent effect. Color register 4 must be assigned to different colors, like red. The best combined color should be a mixture of foreground color (register 1) and register 5:

```
PALETTE 1, 1, 1, 1 'White foreground color'
PALETTE 4, 1, 0, 0 'Red background graphic'
PALETTE 5, 1, .5, .5 'Combined pink color'
```

When you want to put text or a pattern in the background (see the third program above), make sure that the window height allows enough room for the entire graphic or text. You do not want to split the text or pattern in the window. However, if this does happen, after being scrolled, the line will reappear as broken lines.



**5**  
**AmigaBASIC**  
**internals**





---

## 5. AmigaBASIC Internals

AmigaBASIC has a very powerful command set. The manual that comes with it, however, contains many unclear descriptions of commands. Those of you who may have owned another computer before buying an Amiga probably had a number of *utility* programs. Utilities help programmers to program better. Some utilities help users change programs, create new program code or extract old program code. Others allow you to load any program at another starting address.

Since memory manipulation is so complex on the Amiga, there are no memory handling programs in this chapter. However, there are a number of other utilities here to let you change program code. The authors have devised a diskette configuration so that you can load a program into a utility, change the program and save the program back in its edited form. This configuration uses internal drive `df0:`, the RAM disk `ram:` and any external drives (optional). We'll discuss diskette configuration later.

Before continuing with the utilities, you must know about the file types supported by AmigaBASIC. Section 5.2 gives detailed information about Amiga file structures. This information will help you later on with adapting these utilities to your own uses.

### ***Workbench 2.0***

The Workbench 2.0 FD files were not available at the time this book was published, so the following programs have only been tested on Workbench 1.2 and 1.3. When the new 2.0 library FD files are available, the 2.0 `bmap` file can be created. The following programs may require minor changes to operate using the 2.0 `bmap` files.

## 5.1 File Monitor

Now that we know the fundamentals of programming gadgets and accessing Intuition, we'd like to show you a program that uses even more Intuition calls. Accessing the screen displays using the operating system is 10 times faster than in BASIC. Not only that, displaying data on the screen through the operating system is many times faster than in BASIC.

The file monitor in this section permits you to view any disk file in hexadecimal and ASCII text formats. It also allows you to change or edit the file. The file monitor uses an Intuition screen so gadgets control the program.

The following program contains a few BASIC lines that must be entered on one line in AmigaBASIC even though they appear on two lines in this book. This is because formatting the program listings to fit into this book has split some long BASIC lines into two lines of text. To show where a BASIC line actually ends, we added an end-of-paragraph marker (§). This character shows when you should press the (↵) key at the end of a line. For example, the following line appears as two lines below but must be entered as one line in AmigaBASIC:

```
WinDef NWindow, 100, 50, 460, 150, 32+64+512&, 15&+4096&, 0&,
Title$§
```

The § shows the actual end of the BASIC line. Here's the file monitor program listing:

```
'REM DISKMON§
  OPTION BASE 1§
  DEFNG a-z§
' ON ERROR GOTO FAILED ;REM remove after testing§
  DECLARE FUNCTION ALLOCMEM LIBRARY§
  DECLARE FUNCTION GETMSG LIBRARY§
  LIBRARY"T&T2:bmaps/exec.library"§
  LIBRARY"T&T2:bmaps/graphics.library"§
  DECLARE FUNCTION OPENSREEN LIBRARY§
  DECLARE FUNCTION OPENWINDOW LIBRARY§
  LIBRARY"T&T2:bmaps/intuition.library"§
  DECLARE FUNCTION LOCK LIBRARY§
  DECLARE FUNCTION EXAMINE LIBRARY§
  DECLARE FUNCTION EXNEXT LIBRARY§
  DECLARE FUNCTION IOERR LIBRARY§
  DECLARE FUNCTION XOPEN LIBRARY§
  DECLARE FUNCTION XREAD LIBRARY§
  DECLARE FUNCTION XWRITE LIBRARY§
  DECLARE FUNCTION SEEK LIBRARY§
```

```

LIBRARY"T&T2:bmaps/dos.library"¶
' LPRINT :REM used to load printer driver at startup¶
PRINT"-----FILE MONITOR-V1.0-----"¶
PRINT") '88 by DATA BECKER (w)'88 by S. M."¶
PRINT ¶
PRINT"Program starts in a few seconds."¶
PRINT"Please stand by...(no Multitasking"¶
PRINT"during initialization!)"¶
DIM SHARED borders(14),itxt(25),gadgets(24),sinfo(2)¶
bfec01=12577793&¶
clearentry$=SPACE$(30)¶
clearstring$=STRING$(80,0)¶
INITIALIZE¶
DIRECTORY¶
start%=-1¶
blocked%=0¶
WHILE (-1)¶
    qualifier%=PEEK(bfec01)¶
    IF (qualifier%>&H60)AND(qualifier%<&H68)THEN¶
        IF qualifier%AND 1 THEN GOSUB keypressed¶
        END IF ¶
        intuimsg=GETMSG(userport)¶
        IF intuimsg>0 THEN GOSUB IntuitionMsg¶
    WEND ¶
IntuitionMsg:¶
MsgTyp=PEEKL(intuimsg+20)¶
IF MsgTyp=2097152& THEN¶
    IF start% THEN RETURN¶
    ascii.i%=PEEKW(intuimsg+24)¶
    IF ascii.i%>0 GOTO keypressed¶
END IF¶
Item=PEEKL(intuimsg+28)¶
GadgetNr%=PEEK(Item+39)¶
IF MsgTyp=32 THEN¶
    IF (GadgetNr%=10)OR(GadgetNr%=14)THEN blocked%=-1¶
    RETURN¶
END IF ¶
IF MsgTyp<>64 THEN RETURN¶
blocked%=0¶
IF GadgetNr%<0 THEN ERROR 255¶
IF GadgetNr%<5 THEN¶
    COPYMEM SADD(clearstring$),sinfo(1)+36,80¶
    POKEL sinfo(1)+36,CVL("DF"+CHR$(47+GadgetNr%)+":")¶
    lasttype%=0¶
    DIRECTORY¶
ELSEIF GadgetNr%<10 THEN¶
    SETFILEACTDIR¶
    IF lasttype%=1 THEN¶
        STATUS "Loading Block"¶
        OPENFILE¶
        IF oldhandle>0 THEN :STATUS "Edit"¶
        RETURN¶
    END IF¶
    DIRECTORY¶
ELSEIF GadgetNr%=10 THEN¶
    blocked%=0¶

```

```

    DIRECTORY¶
    RETURN¶
ELSEIF GadgetNr%=11 THEN¶
    IF dirstart%>4 THEN dirstart%=dirstart%-5:DISPLAYDIR¶
ELSEIF GadgetNr%=12 THEN¶
    IF number%>dirstart%+5 THEN¶
        dirstart%=dirstart%+5¶
        DISPLAYDIR¶
    END IF ¶
ELSEIF GadgetNr%=13 THEN¶
    DIRECTORY¶
ELSEIF GadgetNr%=14 THEN¶
    blocked%=0¶
    newoffset=PEEKL(sinfo(2)+28)*488¶
    IF (newoffset>=newflen)OR(newoffset<0) THEN¶
        POKEL sinfo(2)+36,CVL("0"+MKI$(0)+CHR$(0))¶
        STATUS "illegal Input"¶
        DISPLAYBEEP scrbase¶
        RETURN¶
    END IF¶
    oldpos=SEEK(oldhandle,newoffset,-1)¶
    currentoffset=newoffset¶
    STATUS "reading Block"¶
    READBLOCK¶
    RETURN¶
ELSEIF oldhandle=0 THEN¶
    POKEL sinfo(2)+36,CVL("0"+MKI$(0)+CHR$(0))¶
    STATUS "no File selected"¶
    DISPLAYBEEP scrbase¶
    RETURN¶
ELSEIF GadgetNr%=15 THEN¶
    COPYMEM fundo,fbuffer,488¶
    DISPLAYBUFFER¶
ELSEIF GadgetNr%=16 THEN¶
    STATUS "reading again"¶
    oldpos=SEEK(oldhandle,-amtread,0)¶
    READBLOCK¶
ELSEIF GadgetNr%=17 THEN¶
    IF currentoffset<newflen-488 THEN¶
        STATUS "reading next Sec"¶
        currentoffset=currentoffset+488¶
        READBLOCK¶
    END IF¶
ELSEIF GadgetNr%=18 THEN¶
    IF currentoffset>487 THEN¶
        STATUS "reading last Sec"¶
        currentoffset=currentoffset-488¶
        oldpos=SEEK(oldhandle,-amtread-488,0)¶
        READBLOCK¶
    END IF¶
ELSEIF GadgetNr%=19 THEN¶
    STATUS "writing Buffer"¶
    oldpos=SEEK(oldhandle,-amtread,0)¶
    wr=XWRITE(oldhandle,fbuffer,amtread)¶
ELSEIF GadgetNr%=20 THEN¶
    DUMPFIL ¶

```

```

ELSEIF GadgetNr%=21 THEN
  DUMPBUFFER
ELSEIF GadgetNr%=22 THEN
  edmode%=0
  STATUS "switched to HEX"
  DISPLAYBUFFER
ELSEIF GadgetNr%=23 THEN
  edmode%=1
  STATUS "switched to ASCII"
  DISPLAYBUFFER
ELSEIF GadgetNr%=24 THEN
  STATUS "ARE YOU SURE? Y/N"
  t%=0
  WHILE (t%<&HD4)AND(t%<&H93)
    t%=PEEK(bfec01)
  WEND
  IF t%=&HD4 THEN
    STATUS "You ARE sure! BYE"
    GOTO FAILED
  END IF
END IF
STATUS "OKAY"
RETURN
keypressed:
  ascii$=UCASE$(CHR$(ascii.i))
  IF edmode%=1 GOTO ASCIImode
  value%=INSTR("0123456789ABCDEF",ascii$)-1
  IF qualifier%=&H67 THEN
    offset%=offset%-32 : 'REM PAL uses 24
    IF offset%<0 THEN offset%=amtread-1:nibble%=1
    CURSOROFF
    CURSORON
    RETURN
  ELSEIF qualifier%=&H65 THEN
    offset%=offset%+32 : REM PAL uses 24
    IF offset%>=amtread THEN offset%=0:nibble%=0
    CURSOROFF
    CURSORON
    RETURN
  ELSEIF qualifier%=&H63 THEN
    IF nibble%=0 THEN
      nibble%=1
    ELSE
      nibble%=0
      offset%=offset%+1
      IF offset%>=amtread THEN offset%=0
    END IF
    CURSOROFF
    CURSORON
    RETURN
  ELSEIF qualifier%=&H61 THEN
    IF nibble%=1 THEN
      nibble%=0
    ELSE
      nibble%=1
      offset%=offset%-1
    END IF
  END IF

```

```

        IF offset%<0 THEN offset%=amtread-1
        END IF
        CURSOROFF
        CURSORON
        RETURN
    END IF
    IF value%>=0 THEN
        IF nibble%=0 THEN andi%=15:muls%=16:GOTO mk
        andi%=240
        muls%=1
mk: a%=(PEEK(fbuffer+offset%)AND andi%)+value%*muls%
        POKE fbuffer+offset%,a%
        CURSOROFF
        MOVE rastport,o.x%,o.y%+6
        SETAPEN rastport,1
        SETBPEN rastport,0
        TEXT rastport,SADD("0123456789ABCDEF")+value%,1
    
```

PAL systems can display both ascii and hex

```

        MOVE rastport,(o.b%+54)*8,o.y%+6
        SETAPEN rastport,0
        SETBPEN rastport,1
        TEXT rastport,fbuffer+offset%,1
    
```

```

        IF nibble%=0 THEN nibble%=1:GOTO mk2
        nibble%=0
        offset%=offset%+1
        IF offset%>=amtread THEN offset%=0
mk2:
        CURSORON
        RETURN
    END IF
    RETURN
ASCIImode:
    IF qualifier%=&H67 THEN
        offset%=offset%-32 :'PAL uses 24
        IF offset%<0 THEN offset%=amtread-1:nibble%=1
        CURSOROFF
        CURSORON
        RETURN
    ELSEIF qualifier%=&H65 THEN
        offset%=offset%+32 :'PAL uses 24
        IF offset%>=amtread THEN offset%=0:nibble%=0
        CURSOROFF
        CURSORON
        RETURN
    ELSEIF qualifier%=&H63 THEN
        offset%=offset%+1:IF offset%>=amtread THEN offset%=0
        CURSOROFF
        CURSORON
        RETURN
    ELSEIF qualifier%=&H61 THEN
        offset%=offset%-1
        IF offset%<0 THEN offset%=amtread-1
        CURSOROFF
        CURSORON
    
```

```

RETURN
END IF
IF ascii$ <> CHR$(0) THEN
    value%=ascii.i
    POKE fbuffer+offset%,value%
    CURSOROFF
    ' PAL Systems can be adapted to display both ASCII and hex
    ' MOVE rastport,o.x%+(o.m%=0)*nibble%,o.y%+6
    ' SETAPEN rastport,1
    ' SETBPEN rastport,0
    ' TEXT rastport,SADD(RIGHT$("0"+HEX$(value%),2)),2
    SETAPEN rastport,0
    SETBPEN rastport,1
    ' REM PAL original: MOVE rastport,(o.b%+54)*8,o.y%+6
    MOVE rastport,o.b%*8,o.y%+6
    TEXT rastport,fbuffer+offset%,1
    offset%=offset%+1
    IF offset%>amtread THEN offset%=0
    CURSORON
    RETURN
END IF
RETURN
' FAILED:
UNDEF
IF scrbase>0 THEN
    IF winbase>0 THEN
        CLOSEWINDOW winbase
        IF oldhandle>0 THEN :XCLOSE oldhandle
    END IF
    CLOSESCREEN scrbase
END IF
LIBRARY CLOSE
END
' SYSTEM
SUB DUMPBUFFER STATIC
    SHARED fbuffer,HEXBUFF,currentlongs,currentoffset
    outstring$=SPACE$(1134)
    HEXBUFF currentlongs-1,fbuffer,SADD(outstring$)
    STATUS "printing"
    FOR i%=0 TO 20
        LPRINT RIGHT$("
+STR$(currentoffset+i%*20),8);": " ";
        LPRINT MID$(outstring$,i%*54+1,54)
    NEXT
    LPRINT
END SUB
SUB DUMPFILE STATIC
    SHARED amtread,oldhandle,currentoffset
    savedoffset=currentoffset
    oldpos=SEEK(oldhandle,0,-1)

```

```

    currentoffset=0¶
df.loop:¶
    READBLOCK¶
    DUMPBUFFER¶
    currentoffset=currentoffset+488¶
    IF amtread=488 GOTO df.loop¶
    currentoffset=savedoffset¶
    oldpos=SEEK(oldhandle,currentoffset,-1)¶
    READBLOCK¶
END SUB¶
¶
SUB CURSORON STATIC¶
    SHARED o.x%,o.y%,edmode%,o.m%,rastport,offset%,nibble%¶
    SHARED o.b%¶
    z%=INT(offset%/32)¶
    o.b%=offset%-z%*32¶
    l%=INT(o.b%/4)¶
    o.x%=(o.b%*2+l%-(edmode%=0)*nibble%)*8¶
    o.y%=z%*8+2¶
    SETAPEN rastport,3¶
    SETDRMD rastport,3¶
    ¶
    IF edmode%=0 THEN RECTFILL rastport,o.x%,o.y%,o.x%+7-
(edmode%=1)*8,o.y%+7¶
'REM original PAL :RECTFILL rastport,o.x%,o.y%,o.x%+7-
(edmode%=1)*8,o.y%+7¶
    ¶
    IF edmode%=1 THEN RECTFILL
rastport,(o.b%)*8,o.y%,(o.b%)*8+7,o.y%+7¶
'REM original PAL: RECTFILL
rastport,(o.b%)*8,o.y%,(o.b%)*8+7,o.y%+7¶
    ¶
    SETDRMD rastport,1¶
    o.m%=edmode%¶
END SUB¶
¶
SUB CURSOROFF STATIC¶
    SHARED o.x%,o.y%,o.m%,o.b%,rastport¶
    SETAPEN rastport,3¶
    SETDRMD rastport,3¶
    ¶
    IF o.m%=0 THEN RECTFILL rastport,o.x%,o.y%,o.x%+7-
(o.m%=1)*8,o.y%+7¶
'REM original PAL: RECTFILL rastport,o.x%,o.y%,o.x%+7-
(o.m%=1)*8,o.y%+7¶
    ¶
    IF o.m%=1 THEN RECTFILL
rastport,(o.b%)*8,o.y%,(o.b%)*8+7,o.y%+7¶
'REM original PAL: RECTFILL
rastport,(o.b%+54)*8,o.y%,(o.b%+54)*8+7,o.y%+7¶
    ¶
    SETDRMD rastport,1¶
END SUB ¶
¶
SUB OPENFILE STATIC¶
    SHARED oldhandle,scrbase,currentoffset,actdir,newflen ¶

```



```

SHARED numblocks¶
IF oldhandle>0 THEN :XCLOSE oldhandle¶
oldhandle=XOPEN(actdir,1005)¶
IF oldhandle=0 THEN¶
    STATUS "File Open Error"¶
    DISPLAYBEEP scrbase¶
    EXIT SUB¶
END IF¶
numblocks=newflen/488¶
w=CVL(RIGHT$(" "+STR$(numblocks),4))¶
POKEL itxt(12)+20,w¶
currentoffset=0¶
READBLOCK¶
END SUB¶
SUB READBLOCK STATIC¶
SHARED oldhandle,fbuffer,fundo,amtread,currentlongs¶
SHARED currentoffset¶
amtread=XREAD(oldhandle,fbuffer,488)¶
IF amtread<488 THEN¶
    v$=STRING$(488-amtread,0)¶
    COPYMEM SADD(v$),fbuffer+amtread,LEN(v$)¶
END IF¶
x=currentoffset/488¶
w=CVL(LEFT$(MID$(STR$(x),2)+MKL$(0),4))¶
POKEL sinfo(2)+36,w¶
currentlongs=(amtread+3)/4¶
COPYMEM fbuffer,fundo,488 ¶
DISPLAYBUFFER¶
END SUB ¶
SUB DISPLAYBUFFER STATIC¶
SHARED HEXBUFF,currentlongs,fbuffer,rastport,amtread¶
SHARED start%,offset%,nibble%,edmode%¶
ASCIIbuffer$=SPACE$(1134)¶
HEXBUFF currentlongs-1,fbuffer,SADD(ASCIIbuffer$)¶
SETAPEN rastport,0¶
RECTFILL rastport,0,0,639,140¶
SETAPEN rastport,1¶
SETBPEN rastport,0¶
IF edmode%=0 THEN¶
    l%=72¶
    FOR i%=0 TO 15¶
        MOVE rastport,0,i%*8+8¶
        IF i%=15 THEN l%=17¶
        TEXT rastport,SADD(ASCIIbuffer$)+i%*72,l%¶
    NEXT¶
END IF¶
SETAPEN rastport,0¶
SETBPEN rastport,1¶
l%=32¶
IF edmode%=1 THEN¶
    FOR i%=0 TO 15¶
        MOVE rastport,0,i%*8+8¶
        IF i%=15 THEN l%=8¶
        TEXT rastport,fbuffer+i%*32,l%¶
    NEXT¶
END IF¶

```

```

start%=0¶
offset%=0¶
nibble%=0¶
CURSORON¶
END SUB ¶
SUB SETFILEACTDIR STATIC¶
  SHARED GadgetNr%,actdir,scrbase,clearstring$,newflen¶
  SHARED dirstart%,dirbuff,lasttype%¶
  compare1$=STRING$(31,0)¶
  compare2$=STRING$(80,0)¶
  COPYMEM actdir,SADD(compare2$),79¶
  COPYMEM itxt(GadgetNr%)+20,SADD(compare1$),30¶
  12%=INSTR(compare2$,CHR$(0))-1¶
  11%=INSTR(compare1$,CHR$(0))-1¶
  IF lasttype%=1 THEN¶
    12%=INSTR(compare2$,":")¶
  path.loop:¶
    13%=INSTR(12%+1,compare2$,"/")¶
    IF 13%>12% THEN 12%=13%:GOTO path.loop¶
  END IF ¶
  IF (11%+12%)>78 THEN¶
    STATUS "FileName Too Long"¶
    DISPLAYBEEP scrbase¶
    EXIT SUB¶
  END IF¶
  v$=LEFT$(compare2$,12%)¶
  IF (lasttype%>1)AND(RIGHT$(v$,1)<>":") THEN v$=v$+"/"¶
  lasttype%=PEEK(dirbuff+(dirstart%+GadgetNr%-5)*36+31)¶
  v$=LEFT$(v$+compare1$+clearstring$,79)¶
  COPYMEM SADD(v$),actdir,79¶
  newflen=PEEKL(dirbuff+(dirstart%+GadgetNr%-5)*36+32)¶
END SUB ¶
SUB DISPLAYDIR STATIC¶
  SHARED dirstart%,number%,cleareentry$,dirbuff,winbase¶
  FOR i%=5 TO 9¶
    COPYMEM SADD(cleareentry$),itxt(i%)+20,30¶
  NEXT¶
  i%=0¶
  IF number%<=dirstart% GOTO displaydir.show¶
  REFRESHGADGETS gadgets(23),winbase,0¶
displaydir.loop:¶
  a=dirbuff+(i%+dirstart%)*36¶
  COPYMEM a,itxt(i%+5)+20,30¶
  POKE itxt(i%+5),PEEK(a+31)¶
  i%=i%+1¶
  IF (i%<5)AND(number%>(dirstart%+i%))GOTO displaydir.loop¶
displaydir.show: ¶
  REFRESHGADGETS gadgets(23),winbase,0¶
END SUB¶
SUB DIRECTORY STATIC¶
  SHARED number%,dirstart%,actdir,lasttype%¶
  SHARED fileinfo,cleareentry$,dirbuff,newflen¶
  STATUS "Examining Entry"¶
  dirlock=LOCK(actdir,-2)¶
  IF dirlock=0 THEN¶
    STATUS "File not found"¶

```

```

EXIT SUB¶
END IF¶
e=EXAMINE(dirlock,fileinfo)¶
IF e=0 THEN¶
    UNLOCK dirlock¶
    STATUS "Examine Error"¶
    EXIT SUB¶
END IF¶
IF PEEKL(fileinfo+120)<0 THEN¶
    newflen=PEEKL(fileinfo+124)¶
    UNLOCK dirlock¶
    OPENFILE¶
    lasttype%=1¶
    EXIT SUB¶
END IF¶
lasttype%=3¶
number%=0¶
dirstart%=0¶
FOR i%=5 TO 9¶
    COPYMEM SADD(clearentry$),itxt(i%)+20,30¶
NEXT¶
STATUS "reading Directory" ¶
directory.loop:¶
e=EXNEXT(dirlock,fileinfo)¶
IF e=0 THEN¶
    e=IOERR¶
    IF e<>232 THEN¶
        STATUS "Directory invalid"¶
        number%=0¶
    ELSE¶
        STATUS "Okay"¶
    END IF¶
    UNLOCK dirlock¶
    DISPLAYDIR¶
    EXIT SUB¶
END IF¶
a=dirbuff+number%*36¶
COPYMEM fileinfo+8,a,30¶
IF PEEKL(fileinfo+120)<0 THEN c%=1 ELSE c%=3¶
POKE a+31,c%¶
POKEL a+32,PEEKL(fileinfo+124)¶
number%=number%+1¶
IF number%<72 GOTO directory.loop¶
UNLOCK dirlock¶
STATUS "Okay"¶
DISPLAYDIR¶
END SUB¶
SUB INITIALIZE STATIC¶
SHARED HEXBUFF,fbuffer,fundo,nscreen,dirbuff,fileinfo¶
SHARED actdir,scrbase,winbase,viewport,rastport¶
SHARED userport¶
FORBID¶
DEFCHIP HEXBUFF,60&¶
DEFCHIP fbuffer,488&¶
DEFCHIP fundo,488&¶
DEFCHIP nscreen,88&¶

```

```

DEFCHIP dirbuff,2592&¶
DEFCHIP fileinfo,252&¶
DEFCHIP shows,68&¶
borders(13)=shows+28¶
borders(14)=shows+48¶
FOR i%=0 TO 14¶
  READ i$¶
  POKEW HEXBUFF+i%*4,VAL("&H"+LEFT$(i$,4))¶
  POKEW HEXBUFF+i%*4+2,VAL("&H"+RIGHT$(i$,4))¶
NEXT¶
FOR i%=0 TO 6¶
  READ i$¶
  POKEW shows+i%*4,VAL("&H"+LEFT$(i$,4))¶
  POKEW shows+i%*4+2,VAL("&H"+RIGHT$(i$,4))¶
NEXT¶
POKE shows+29,10¶
POKE shows+31,3¶
POKE shows+33,7¶
POKE shows+35,7¶
POKE shows+37,1¶
POKEW shows+42,256¶
COPYMEM shows+28,shows+48,20¶
FOR i%=0 TO 1¶
  POKEL shows+i%*20+38,shows+i%*14¶
NEXT¶
FOR i%=1 TO 12¶
  READ a%,b%,c%,d%,e%,f%¶
  BORDER borders(i%),a%,b%,c%,d%,e%¶
  IF f%>0 THEN POKEL borders(i%)+12,borders(f%)¶
NEXT¶
FOR i%=1 TO 4¶
  INTUITEXT itxt(i%),1,6,3,"DF"+CHR$(47+i%)+":",0&¶
NEXT¶
FOR i%=5 TO 9¶
  INTUITEXT itxt(i%),1,8,0,SPACE$(30),0&¶
NEXT¶
FOR i%=10 TO 25¶
  READ a%,b%,c%,d$,e%¶
  IF e%>0 THEN f=itxt(e%) ELSE f=0¶
  INTUITEXT itxt(i%),a%,b%,c%,d$,f¶
NEXT¶
STRINGINFO sinfo(1),79,"DF0:"¶
STRINGINFO sinfo(2),4,"0"+STRING$(15,0)¶
actdir=sinfo(1)+36¶
d=0¶
FOR i%=1 TO 24¶
  READ e%,f%,G%,h%,j%,k%,l%,m%,n%,o%¶
¶
f%=f%-56 :REM NTSC FIX *****¶
¶
IF o%>0 THEN a=sinfo(o%) ELSE a=0¶
IF n%>0 THEN b=itxt(n%) ELSE b=0¶
IF m%>0 THEN c=borders(m%) ELSE c=0¶
GADGET gadgets(i%),d,e%,f%,G%,h%,j%,k%,l%,c,b,a,i%¶
d=gadgets(i%)¶
NEXT¶

```

```

POKEL nscreen+4,41943296&¶
POKE nscreen+9,2¶
POKE nscreen+12,192¶
POKEW nscreen+14,&H10F¶
nwindow=nscreen+32¶
POKEL nwindow+4,41943296&¶
POKEW nwindow+8,259¶
POKE nwindow+11,32¶
POKE nwindow+13,96¶
POKE nwindow+15,1¶
POKE nwindow+16,24¶
POKEL nwindow+18,d¶
POKE nwindow+47,15¶
POKEW nscreen+82,&HFFF¶
POKE nscreen+84,15¶
POKEW nscreen+86,&HFD0¶
PERMIT¶
scrbase=OPENSSCREEN(nscreen)¶
IF scrbase=0 THEN ERROR 7¶
POKEL nwindow+30,scrbase¶
winbase=OPENWINDOW(nwindow)¶
IF winbase=0 THEN ERROR 7¶
rastport=PEEKL(winbase+50)¶
viewport=scrbase+44¶
userport=PEEKL(winbase+86)¶
LOADRGB4 viewport,nscreen+80,4¶
END SUB¶
SUB STATUS(t$)STATIC¶
  SHARED winbase¶
  t$=LEFT$(t$+SPACE$(17),17)¶
  COPYMEM SADD(t$),itxt(22)+20,17¶
  REFRESHGADGETS gadgets(23),winbase,0¶
END SUB ¶
SUB DEFCHIP(Buffer,size)STATIC¶
  SHARED MList¶
  size=size+8¶
  Buffer=ALLOCMEM(size,65538&)¶
  IF Buffer>0 THEN¶
    POKEL Buffer,MList¶
    POKEL Buffer+4,size¶
    MList=Buffer¶
    Buffer=Buffer+8¶
  ELSE¶
    ERROR 7¶
  END IF¶
END SUB¶
SUB UNDEF STATIC¶
  SHARED MList¶
  undef.loop:¶
  IF MList>0 THEN¶
    Buffer=PEEKL(MList)¶
    size=PEEKL(MList+4)¶
    FREEMEM MList,size¶
    MList=Buffer¶
    GOTO undef.loop¶
  END IF¶

```

```

END SUB      ¶
SUB GADGET(bs,nx,x%,y%,b%,h%,f%,a%,t%,i,txt,si,n%) STATIC¶
DEFCHIP bs,44&¶
POKEL bs,nx¶
POKEW bs+4,x%¶
POKEW bs+6,y%¶
POKEW bs+8,b%¶
POKEW bs+10,h%¶
POKEW bs+12,f%¶
POKEW bs+14,a%¶
POKEW bs+16,t%¶
POKEL bs+18,i¶
POKEL bs+26,txt¶
POKEL bs+34,si¶
POKEW bs+38,n%¶
END SUB¶
SUB INTUITEXT(bs,c1%,x%,y%,t$,nx) STATIC¶
size=20+LEN(t$)+1¶
DEFCHIP bs,size¶
POKE bs,c1%¶
POKE bs+2,1¶
POKEW bs+4,x%¶
POKEW bs+6,y%¶
POKEL bs+12,bs+20¶
POKEL bs+16,nx¶
COPYMEM SADD(t$),bs+20,LEN(t$) ¶
END SUB¶
SUB BORDER(bs,x%,y%,c%,b%,h%) STATIC¶
DEFCHIP bs,48&¶
POKEW bs,x%¶
POKEW bs+2,y%¶
POKE bs+4,c%¶
POKE bs+7,8¶
POKEL bs+8,bs+16¶
FOR i%=0 TO 1¶
POKEW bs+22+i%*4,h%-1¶
POKEW bs+24+i%*4,b%-1¶
POKEW bs+32+i%*4,1¶
POKEW bs+38+i%*4,h%-1¶
POKEW bs+40+i%*4,b%-2¶
NEXT¶
END SUB¶
SUB STRINGINFO(bs,max%,buff$) STATIC¶
IF LEN(buff$)>max% THEN nmax%=LEN(buff$) ELSE nmax%=max%¶
IF (nmax%AND 1) THEN nmax%=nmax%+1¶
size=36+2*(nmax%+4) ¶
DEFCHIP bs,size¶
POKEL bs,bs+36¶
POKEL bs+4,bs+40+nmax%¶
POKEW bs+10,max%+1¶
IF buff$<>" " THEN¶
COPYMEM SADD(buff$),bs+36,LEN(buff$) ¶
END IF¶
END SUB      ¶
DATA 48E7F0C0,4CEF0308,001C5303,22187407,E9991001,
0200000F¶

```

```

DATA 06000030,0C00003A,65040600,000712C0,51CAFFE6,
12FC0020
DATA 51CBFFDA,4CDF030F,4E750000,10003800,7C00FE00,
38003800
DATA 38003800,38003800,FE007C00,38001000
DATA 0,0,2,43,13,0,-6,-3,2,268,45,0,-6,-3,3,268,13,0
DATA 0,0,2,28,13,0,0,-45,2,28,13,4,0,-15,2,28,13,5
DATA -62,-3,2,172,13,0,0,0,2,65,13,0,0,0,2,109,13,0
DATA 0,15,2,218,13,9,0,0,2,60,13,0,0,2,43,28,0
DATA 3,-56,0,"Block:",0,3,40,0,"of:",10,1,72,0," 0",11
DATA 3,6,3,"OK",0,1,6,3,"UNDO",0,1,6,3,"PRINT BUFFER",0
DATA 1,6,3," PRINT FILE",0,1,17,3,"READ",0,1,17,3,
"NEXT",0
DATA 1,17,3,"BACK",0,1,13,3,"WRITE",0,3,6,18,
"Status:",15
DATA 1,70,18,"reading Directory",21,1,9,3,"ASCII",0
DATA 1,9,3," HEX",0,1,6,10,"QUIT",0
DATA 0,198,43,13,0,3,1,1,1,0,0,213,43,13,0,3,1,1,2,0
DATA 0,228,43,13,0,3,1,1,3,0,0,243,43,13,0,3,1,1,4,0
DATA 52,201,256,8,0,3,1,2,5,0,52,209,256,8,0,3,1,0,6,0
DATA 52,217,256,8,0,3,1,0,7,0,52,225,256,8,0,3,1,0,8,0
DATA 52,233,256,8,0,3,1,0,9,0,52,246,256,8,0,3,4,3,0,1
DATA 317,198,28,13,4,3,1,13,0,0
DATA 317,228,28,13,4,3,1,14,0,0
DATA 317,243,28,13,0,3,1,6,13,0
DATA 416,201,40,8,0,2051,4,7,12,2
DATA 529,198,43,13,1,3,1,1,14,0
DATA 575,198,65,13,0,3,1,8,17,0
DATA 575,213,65,13,0,3,1,8,18,0
DATA 575,228,65,13,0,3,1,8,19,0
DATA 575,243,65,13,0,3,1,8,20,0
DATA 354,213,109,13,0,3,1,9,16,0
DATA 354,228,109,13,0,3,1,10,22,0
DATA 466,213,60,13,0,3,1,11,24,0
DATA 466,228,60,13,0,3,1,11,23,0
DATA 529,213,43,28,128,3,1,12,25,0

```

---

### 5.1.1 Using the file monitor

This monitor uses a large amount of chip RAM. This means that you should only run one task when on a monitor session. This one task is the file monitor. If you run a second program while the file monitor is running, you may run out of chip RAM. The LPRINT at the beginning of the program ensures that the printer driver loads into memory before the program's memory allocation takes place. If you aren't using a printer you may delete this line.

The four gadgets which list the most frequently accessed drives may be changed by editing the corresponding DATA statements. These gadgets are used by the directory routine. To select a drive simply click on the proper gadget.

The default drives for the program range from drive DF0: through drive DF3:. If you want to enter other drives in the gadgets, change the DATA statements with the corresponding names and make sure that the name is no longer than four characters (including the ending colon). You can also assign the desired drives with the drive labels DF0: through DF3: by using `Assign` before loading this monitor.


### *Using the gadgets*

The four gadgets on the left border of the screen help speed up the selection of the drive and directories. Simply click on the DF0: gadget to see the main directory of the internal drive.

The file list displays up to five directory entries. Files and programs appear in white text and directories are shown as yellow text. Clicking on a directory name opens and displays the contents of that directory. When you click on a file, the first data block of the desired file loads into memory and then appears on the screen. This data block can be edited in hexadecimal or ASCII form.

The string gadget under the file list displays the current directory or filename. A cursor appears when you click on it. You can now enter your own paths/filenames from the keyboard. This is useful when you want to enter a long path or if you want to access a drive not listed in the four disk drive gadgets.

The scroll arrows to the right of the file list let you scroll up and down the file list and view all the available names. The directories scroll by five entries at a time.

The OK gadget updates the entry in the string gadget (this is the same as pressing the  key when you're done editing the string gadget).

The line `Block #### of: ####` shows you the current data block number on display of the active file and the total number of blocks in that file. The first block number is handled as an integer gadget. It allows you to enter the desired data block number by clicking on the gadget. This displays the desired block.

Both Print gadgets allow you to output either the editor buffer or the entire file on a printer in hexadecimal format. After the printing process ends, the last block edited reappears on the screen. The Status display shows all of the errors and the current operations. If all is well, the Status display says OK.

The ASCII and HEX gadgets make it possible to select hexadecimal display or ASCII display. This is a valuable option for changing text



(e.g., customizing the AmigaBASIC menus). The Quit gadget, with a confirming requester, ends this program.

The READ, NEXT and BACK gadgets let you read the current block, next block and previous block of the file. The Write gadget writes the editor's buffer to the disk. No requester appears (this increases the operation speed). If you write a block by mistake, select the Undo gadget and select the Write gadget again.

The Undo buffer contains the original contents of the data currently on display. The Undo gadget takes the contents of the Undo buffer and places it in the editor buffer (the buffer containing the data currently displayed).

The editor accepts any characters that can be entered from the keyboard, including the cursor keys. PAL system users can display both hexadecimal and ASCII modes on the screen at once, since the PAL screen has a larger display. The program code above contains comments on what must be changed to run the full display on a PAL system.

Although the program can multitask, we don't recommend it (see the beginning of this segment). The key combinations left <Amiga> **M** and left <Amiga> **N** toggle between the file monitor and Workbench screen.

One last item: The file monitor can only read disk paths up to two directory levels deep. Should you desire more flexibility here, you must dimension the directory buffer correspondingly and adjust the directory subprogram. You can access each file with direct input into the string gadget.

---

## 5.1.2 Patching files with the monitor

Patching means changing an existing program by manipulating certain bytes of that program. This makes it possible to customize any program to suit your own needs.

There is one thing you should bear in mind, however: Changing copyright messages or copying commercial programs, patched or otherwise, is against the law. To stay on the side of law and order, patch any commercial programs for your own use. Don't alter copyright messages or use this file monitor for illegal purposes.

### 5.1.3 Patching AmigaBASIC

Using the file monitor you can customize your copy of AmigaBASIC. You can change the menus and error messages to give your AmigaBASIC interpreter a personal touch.

**Warning:**

Whenever you patch any program or edit any file using the file monitor, make sure you patch a copy of a program or file. Never patch the original program or file.

To patch AmigaBASIC, start by copying AmigaBASIC to another disk. Run the file monitor program and select the ASCII mode. Next insert the disk that contains the copy of AmigaBASIC (NOT the original). Select AmigaBASIC and press the **(←)** key.

Once the file is done loading, click on the Block gadget, enter 28 and press **(←)**. Now use the Next and Back gadgets to page through AmigaBASIC until you find the menus. Use the cursor keys to position the cursor, then edit the file to customize AmigaBASIC. Click on the Write gadget to save your changes. If you made a mistake, click Undo, fix the problem and click the Write gadget again. Click on the Quit gadget and press **(Y)** to quit the program.

You may need a few clues on what to do to change your menus. Here are some examples of what we did to change our AmigaBASIC menus using the file monitor program:

**Original menus:**

Project	Edit	Run	Windows
New	Cut	Start	Show List
Open	Copy	Stop	Show Output
Save	Paste	Continue	
Save as		Suspend	
Quit		Trace on	
		Step	

**Edited menus:**

Stuff	Edit	Run	Screens
Oops it	Cut	Go	List
Load it	Copy	Break	BASIC
Save it	Paste	Keep on	
Save as		Whoa	
System		Trace on	
		Step	

---

## 5.2 AmigaBASIC file structure

AmigaBASIC's `SAVE` command lets users save programs in three different ways:

```
SAVE "Test",a stores the program Test as an ASCII file.  
SAVE "Test",b stores the program in binary form.  
SAVE "Test",p stores the program in protected form.
```

Before you save a program, you should know what you want done with this file later on. That is, the purpose of a file, and the situations in which it is used later.

### *ASCII files*

ASCII files are necessary when you want to combine files using `MERGE` or `CHAIN`. When you want to store a program as an ASCII file, you can reload it later and save it out again as an ASCII, binary (normal) or protected file.

The disadvantage of ASCII files is the amount of memory they consume, especially when many variable names are used (more on this later). This disadvantage also applies to the entire concept of modular programming.

### *Binary files*

Binary files are shorter; the computer converts commands and variables into *tokens*. A binary file can be saved out later in ASCII, binary or protected form.

### *Protected files*

Protected files cannot be corrected or changed in any way. A file cannot be changed once you save a file in protected form. Unlike the other file forms, you can't resave a protected file in ASCII or binary form. Before saving a file as a protected program, make sure you have at least one backup copy of the file in either ASCII or binary form.

---

### 5.2.1 Determining filetype

Now you may want to manipulate AmigaBASIC programs, whether they are on diskette or in buffer memory. As soon as you know the structure of an AmigaBASIC file, there should be no problem with this.

There is one glitch: Say you wrote a program that generates a new AmigaBASIC program from a program already on diskette. This program waits for you to tell it which program you want modified (let's assume that this program is on the diskette currently in the drive). You must know whether this file is an AmigaBASIC file.

### 5.2.1.1 Checking for a BASIC file

This program examines a file and informs you if the file is an AmigaBASIC program.

```
GOTO start¶
¶
REM #####¶
REM # B A S I C - C H E C K #¶
REM #-----#¶
REM # (W) 1987 by Stefan Maelger #¶
REM #####¶
¶
REM SUB-Routine to check whether a File¶
REM is a AmigaBASIC-Program¶
¶
start:¶
¶
DECLARE FUNCTION xOpen% LIBRARY¶
DECLARE FUNCTION xRead% LIBRARY¶
DECLARE FUNCTION Seek% LIBRARY¶
¶
LIBRARY "T&T2:bmaps/dos.library"¶
¶
main:¶
¶
CLS¶
LOCATE 2,2¶
PRINT "Name of AmigaBASIC-Program:"¶
LOCATE 4,1¶
PRINT ">";LINE INPUT Filename$¶
BASICcheck Filename$,Flag%¶
LOCATE 6,2¶
IF Flag% THEN ¶
PRINT "It is an AmigaBASIC-Program!"¶
ELSE¶
PRINT "No, it's not an AmigaBASIC-Program..."¶
END IF¶
LIBRARY CLOSE¶
END¶
¶
SUB BASICcheck (Filename$,ok%) STATIC¶
¶
File$ = Filename$+".info"+CHR$(0)¶
Default.Tool$ = SPACE$(12)¶
```

```

    OpenOldFile% = 1005¶
    OffsetEOF% = 1¶
    Offset% = -12¶
¶
OpenFile:¶
¶
    File.handle& = xOpen&(SADD(File$),OpenOldFile%)¶
    IF File.handle& = 0 THEN¶
        CLS¶
        LOCATE 2,2¶
        PRINT "I can't find ";Filename$;"!"¶
        BEEP¶
        EXIT SUB¶
    ELSE¶

OldPosition%=Seek%(File.handle&,Offset%,OffsetEOF%)¶

GotThem%=xRead%(File.handle&,SADD(Default.Tool$),12)¶
    IF GotThem%<12 THEN¶
        CLS¶
        LOCATE 2,2¶
        PRINT "READ-ERROR"¶
        BEEP¶
        EXIT SUB¶
    ELSE¶
        IF INSTR(Default.Tool$,":AmigaBASIC")>0 THEN¶
            ok%=-1¶
        ELSE¶
            ok%=0¶
        END IF¶
    END IF¶
    CALL xClose(File.handle&)¶
END IF¶
END SUB¶

```

**Variables**

Filename\$	name of the potential AmigaBASIC program
Flag%	=-1: the file is an AmigaBASIC program
ok%	SUB variable indicator from flag%
File\$	name of the info file from Filename\$+CHR\$(0)
Default.Tool\$	12-byte string, taken from the last 12 bytes of file\$
OpenOldFile%	parameter used when file opens (1006=new file open)
OffsetEOF%	sets cursor to end of file during file read routine (-1=beginning, 0=present position)
File.handle&	file handle address (0=file not open)
OldPosition%	old file cursor offset
GotThem%	number of bytes read so far

**Program  
description**

If you've tried out the `Info` item from the `Workbench` pulldown menu, you've seen the `Default Tool` string gadget in the `Info` window. `Default Tool` is the main program that loads when you double-click a program's icon. For example, if you double-click an AmigaBASIC program's icon, AmigaBASIC loads first, then the program loads and runs. So, the `Default Tool` gadget of an AmigaBASIC program contains the entry `:AmigaBASIC`. Every AmigaBASIC program (and most programs) have a companion file called an *info* file. This file has the same name as the program with an added file extension of `.info`. This info file holds the bitmap of the program's icon, as well as the `Default Tool` designation.

To determine whether a file is an AmigaBASIC program, this program opens the matching info file, moves the cursor to a location 12 bytes from the end of the file and reads the `Default Tool` gadget. Why 12 bytes? The entry only has 11 bytes and AmigaDOS only accepts names ended with `CHR$(0)`.

**Note:**

Some programs that allow icon editing and creation may not work quite right. These program errors can result in a misplaced `Default Tool`. You can get around this error by raising the number of bytes you want read.

### 5.2.1.2 Checking the program header

Now you know how to identify a file as an AmigaBASIC program. You still can't change the program yet; you have to determine the program type before any changes can be made. The AmigaBASIC interpreter must know the program type.

**Header bytes**

The first byte of an AmigaBASIC program conveys the program type. This byte is called the *header byte*. Programs stored in binary (normal) form and protected form attach this header byte to the beginning of the file. ASCII files contain no header bytes, since they don't need header bytes (see Section 5.2.2 below for details on ASCII file structure).

The header byte assignments are as follows:

<code>\$F5</code>	binary program
<code>\$F4</code>	protected program
no header byte	ASCII file

The program below performs this function. This program requires the `dos.library` routines `xRead` and `xWrite`. Remember to have this library file available on the diskette currently in the drive.

```

GOTO start¶
¶
' #####¶
' # H E A D E R - C H E C K #¶
' #-----#¶
' # (W) 1987 by Stefan Maelger #¶
' #####¶
'¶
' SUB-Routine to determine the File-Type¶
' of an AmigaBASIC-Program from the¶
' File-Headers.¶
' -----¶
'¶
start:¶
¶
  DECLARE FUNCTION xOpen& LIBRARY¶
  DECLARE FUNCTION xRead% LIBRARY¶
  ¶
  LIBRARY "T&T2:bmaps/dos.library"¶
  ¶
main:¶
  ¶
  ProgramType$(0)="n ASCII-File"¶
  ProgramType$(1)=" Binary-File"¶
  ProgramType$(2)=" Protected-Binary-File"¶
  ¶
  LINE INPUT "Filename: >";Filename$¶
  ¶
  HeaderCheck Filename$,Result%¶
  ¶
  LOCATE 10,1¶
  ¶
  PRINT "The Program ";CHR$(34);¶
  PRINT Filename$;CHR$(34);¶
  PRINT " is a";ProgramType$(Result%)¶
  ¶
  LOCATE 15,1¶
  ¶
  LIBRARY CLOSE¶
  END¶
  ¶
SUB HeaderCheck (Filename$,Result%) STATIC¶
  ¶
  File$=Filename$+CHR$(0)¶
  OpenOldFile%=1005¶
  handle&=xOpen& (SADD (File$), OpenOldFile%)¶
  IF handle&=0 THEN ERROR 53¶
  s$="1"¶
  Byte&=1¶
  Count&=xRead% (handle&, SADD (s$), Byte&)¶
  CALL xClose (handle&)¶
  Result%=0¶
  d%=ASC (s$)¶
  IF d%=&HF5 THEN¶
    Result%=1¶
  ELSEIF d%=&HF4 THEN¶

```

```

        Result%=2
    END IF
    1
END SUB

```

<b>Variables</b>	ProgramType\$	program type
	Filename\$	name of the AmigaBASIC program
	Result%	0=ASCII; 1=binary; 2=protected
	File\$	Filename\$+CHR\$(0)
	OpenOldFile%	parameter used for open file
	handle&	file handle address
	s\$	string from which first byte is read
	Byte&	number of bytes to be read
	Read&	number of bytes read so far
	d%	ASCII value from s\$

### 5.2.2 ASCII files

ASCII file structure is really quite simple. Load AmigaBASIC and enter the following program code:

```

a=1
PRINT a

```

Save this program using the following syntax:

```

SAVE "Test",A

```

Now quit AmigaBASIC and load up the file analyzer program from Section 5.1 (or use some other file monitor if you have one available). When the file analyzer finishes loading, select the Open item from the menu and enter the name of the program you just saved.

The program code appears on the right hand side of the screen:

```

a=1.PRINT a..

```

And the hex dump of the program appears on the left hand side of the screen:

```

61 3D 31 0A 50 52 49 4E 54 20 61 0A 0A

```

If you convert these hex numbers to decimal notation, they look like this:

```

97 61 49 10 80 82 73 78 84 32 97 10 10

```



Look in Appendix A of your AmigaBASIC manual for a list of ASCII character codes. You'll see that these numbers match the text. Character code 10 executes a linefeed (next line).

If you want to read a program saved as an ASCII file, use the following program in AmigaBASIC:

```
LINE INPUT File$¶
OPEN File$ FOR INPUT AS 1¶
  WHILE NOT EOF(1)¶
    PRINT INPUT$(1,1);
  WEND¶
CLOSE 1¶
```

Insert your Workbench diskette.

- Start up the Shell.
- Enter the following:

```
ed Diskname:Test
```

Diskname is the name of the diskette on which you saved the Test program. You can edit ASCII programs using Ed (the editor) from the Workbench diskette. The main disadvantage to Ed is that you cannot test programs using it.

If you thought of simply creating a new program using OPEN name FOR OUTPUT, you had a good idea. The problem with that, though, comes up when you try loading the new program into the directory. The filename .info has no :AmigaBASIC listed as its Default Tool. Just do the following to create a new info file:

```
SAVE "Dummy":KILL File$+".info"¶
NAME "Dummy.info" AS File$+".info"¶
KILL "Dummy"¶
```

See Section 5.3 for practical applications using ASCII files.

### 5.2.3 Binary files

Binary file structure is extremely important since this is the usual file format directly accessible from the AmigaBASIC interpreter. All other filetypes must be converted to binary format before AmigaBASIC can execute them.

Binary programs have a header byte containing \$F5.

The first program line begins at the second byte of the program. This would be a good time to examine the structure of an AmigaBASIC line.

### 5.2.3.1 Structure of an AmigaBASIC line

**Line header** The first byte of a line is the line header. This byte can have one of two values: 0 or 128 (\$80 hexadecimal). If the line begins with 0, the line is handled as if it has no line number. If the line begins with 128, then it has a line number. Labels do not apply to this header (more on this later).

**Line offset** The second byte of a line is the offset to the next line. It would be pretty complicated to try figuring out pointers to the next line every time an AmigaBASIC program loads and runs at different memory locations. Instead, AmigaBASIC counts the total length of the current line. The interpreter then figures out the address at which the line begins, and takes the number of occupied bytes from it. If the interpreter must jump a number of lines forward (e.g., during a jump command), it just adds the line length of the current line to the starting address.

Line length is represented in only one byte. This is why a program line can be no longer than 255 bytes.

Indenting program lines can make your program code more easily readable for debugging or when trying to read a program for its flow of execution. A program might look something like this:

multiple.FOR.NEXT.loops:	0
FOR FirstLoop=1 TO 100	2
FOR secondLoop=1 TO 10	4
FOR thirdLoop=1 TO 50	6
LPRINT FNstepon (x,y,z)	8
NEXT thirdLoop,secondLoop,FirstLoop	2

The numbers at the right of the lines above don't belong to the program itself. These are the numbers taken up by the third byte of the matching program line. Take a look at these with the file monitor. Only LIST and editing commands make use of this byte. It gives the spacing of the first command from the left margin. This answers the question as to whether the program length or execution speed are affected by indentation. The only change is in the value of the third byte.

**Line numbers** Now look at the difference between the structures of a line containing a line number and a line without a line number. Up to now, you've seen how a line without line numbers is handled. Here's a review:

Byte	Value	Definition
1	00	Line without line number follows
2	xx	Line length in bytes (with head and end)
3	xx	Spacing from left margin to first command (for LISTing programs only)

Lines with line numbers have two additional bytes, making the line header a total of five bytes long. Bytes four and five give the line number in high byte/low byte format. For example, if the line number is 10000, the fourth and fifth byte return \$27 and \$10 respectively (39 and 16 decimal:  $39*256+16=\text{line number}$ ). The structure looks like this:

Byte number	Value	Definition
1	128	Line with line number follows
2	xx	Line length in bytes (with head and end)
3	xx	Spacing from left margin to first command (for LISTing programs only)
4	xx	Line number (high byte)
5	xx	Line number (low byte)

Both line structures are similar. The bytes following are the *tokens* (commands coded into two-byte numbers).

BASIC lines end with the value 0 (an extra byte). To summarize, a program line consists of:

1. a program header with or without line numbers
2. tokens (commands, labels, variables and values)
3. end byte of 0

### ***Blank lines***

Now that you know about line storage, you may already know how blank spaces are stored. The blanks discussed here are those spaces between one line and the next.

Here's the problem: The first byte must contain a zero, so no line number follows. The third byte (indentation) is also zero most of the time). The fourth byte starts the token list. If this line is blank, the end-of-line code (another zero) follows. The line ends and the total line length (four bytes) goes to the second byte of the line.

A blank line looks something like this:

\$00 - \$04 - \$xx - \$00

It's obvious here that every blank line takes up four bytes of memory and slows down the computer's execution time, since the interpreter checks these blank lines for commands. You should remove blank lines from your programs, especially programs that are time-critical. You

know the old saying—little things add up. See Section 5.3 for a program that removes blank lines.

### *The last line*

The last line of a program begins with a null byte. There is no line number offset. The next byte is the line length byte, which is also set to null, then the end-of-line code (again, a zero).

Other bytes could follow, say when a program has been edited. These bytes can have some strange values.

### *Variable tables*

Variable names can be up to 40 characters long in AmigaBASIC. The problem comes up every time access occurs on a variable stored under its full name. In order to use long-named variables without slowing the computer down, the programmer must do the following in this BASIC dialect:

When a variable occurs, the interpreter reads a special token. This token always has the value \$01. Following this token is a number in high byte/low byte format. The interpreter simply numbers each variable and continues program execution based upon variable numbers. These variables must be stored under their full names so that LIST lists these variables under their full names. The end of the program contains a variable table to accomplish this. An entry in this table appears in the following format:

1st byte	Length of the variable name in bytes
successive bytes	Variable names in ASCII code.

For example, if you use the variables `a%`, `String$` and `Address&` in your program, the variable table would look something like this:

Hexadecimal	ASCII
01	61 .a
06	5A 74 72 69 6E 67 .String
07	41 64 64 72 65 73 73 .Address

The last byte of your program would then be \$65. It doesn't matter what type the variable is to the table—these follow the variable number set by the token \$01. If you look at the above example, the `a%` variable lies in the program as follows:

Byte number	Value	Definition
1	1	Variable number follows
2	0	High byte of variable number
3	0	Low byte of variable number
4	37	ASCII code of % character

The above table shows you that the first variable is assigned the number zero.

Unfortunately, the variables in AmigaBASIC aren't as simple as all that. The order of the variables in the variable table is the order in which you first typed them in. To see this bug in action, do the following:

- Load AmigaBASIC.
- Enter the following:

```
The.big.error%=0
Blahblahblah%=The.error%
Hello%=0
```

- Change Blahblahblah% to read:

```
Blahblahblah%=The.big.error%
```

- Save the program in binary form, and look at it with the file monitor.

The program itself no longer contains the `error%` variable. However, the variable table still has this variable. If you write a long program and mistype or change some variable names, you're still stuck with the original errors/variable names in the variable table regardless if you use them or not. Your program could end up several kilobytes longer than you need, and execution time suffers as well.

See Section 5.3.6 for a solution to this problem.

Another AmigaBASIC bug is that all SUB programs, their calls and all operating system routines called by LIBRARY and/or DECLARE FUNCTION are set up as *variables* in the table and the program text. AmigaBASIC can only recognize these names in complete syntax checking as functions or SUB extensions. This makes no difference to the BASIC interpreter, which goes through a complete check of the program before starting it. This means that some delay can occur between a program loading and eventually starting.

### ***Label handling***

Labels are similar to variables. The developers of AmigaBASIC had some problems dealing with long label names. The solution is as follows: Labels are treated as special variables—different from other variables in that they are used for program branching.

This means that labels are sorted out in the variable table like a normal variable. Now the BASIC interpreter must be able to recognize a label, since no memory is set aside for labels. A special token (`$02`) marks labels in program code. When the interpreter encounters a `$02`, the number immediately following is the high byte/low byte number of a label. For example:

Byte number	Value	Definition
1	2	Label number follows
2	xx	High byte of label number
3	xx	Low byte of label number

If the interpreter finds \$02 \$00 \$09 in the program, it knows that there is a label here whose name is at the tenth place in the variable table (this table begins its numbering at 0).

### *Label branching*

You can jump to any label you want, especially useless ones like REMARKS. This section talks about GOTO and labels, but the same applies to GOSUB.

Example:     GOTO division

Let's assume that `division` stands at the third place in the variable table. The interpreter finds the following in the program:

Byte number	Value	Definition
1	151	Token for GOTO (see Appendices)
2	32	Space
3	3	Token=label that should be branched to
4	0	Always 0
5	0	High byte of number in variable table
6	2	Low byte of number

You've just learned a new token—\$03. The interpreter looks for a \$02-\$00-\$02 and continues program execution at that point.

### *Line number branching*

Line number branches are very different from label branches. The reason is that line numbers aren't stored in the variable table. A new token is required:

Example:     GOTO 10000

Byte number	Value	Definition
1	151	Token for GOTO (see Appendices)
2	32	Space
3	14	Token=branch to following line number
4	0	Always 0
5	39	High byte of line number (39*256)
6	16	Low byte of line number (+16=10000)

The \$0E token means that in all lines containing header bytes of \$80, bytes 4 and 5 must be compared with bytes 5 and 6 to find the branch line.

*Values in AmigaBASIC*

AmigaBASIC has another big difference from other versions of BASIC: AmigaBASIC uses its own methods of handling values in its program codes. For example, take a simple variable assignment like the one listed below:

```
Amiga=1
```

The item of interest here is the way the "1" is stored in the program. Unlike the methods used in other BASIC dialects, in which numbers are converted to their ASCII equivalents (which takes time during program execution), AmigaBASIC stores numbers and values in the necessary format. For every format (e.g., floating-point or octal), a new token must exist. Let's go through this process step by step.

The process used to differentiate the format selection is a stupid one; it's not dependent upon the needs of the variable. Look at the above example. It goes without saying that the number 1 would be handled as an integer. The next important fact is that the number is a single-digit number. When it comes down to the leading character of the number (positive or negative), the following occurs:

Positive integers from 0 to 9 go into the program without tokens. The ASCII code is unused. Direct storage in memory is impossible, since the numbers can be interpreted as other values (e.g., "0" means end-of-line and "1" means "Variable number"). The values are coded as follows:

Hex	Dec	Value (decimal)
\$11	17	0
\$12	18	1
\$13	19	2
...	...	...
\$19	25	8
\$1A	26	9

When the interpreter finds a byte between 17 and 26, it replaces the value 17 with the proper value.

Now take a look at positive integer values between 10 and 255. One byte is enough for storing these numbers. Again, a token is required so that the interpreter cannot mistake the number for a command token or other token. The format is:

Byte number	Value	Definition
1	15	A positive integer from 10 to 255 follows
2	xx	Value between 10 and 255

Integer values can also be larger than 255, and positive or negative. These numbers use this format:

Byte number	Value	Definition
1	28	A 2-byte integer with leading character follows
2	xx	High byte (bit 7=leading character bit)
3	xx	Low byte

Integers larger than 32767 are represented in long-integer format:

Byte number	Value	Definition
1	30	A 4-byte integer with leading character follows
2-5	xx	4-byte integer (bit 7 in byte 2=leading character bit)

If the value should be handled as a floating-point number, use the following format:

Byte number	Value	Definition
1	29	A 4-byte floating-point number follows
2-5	xx	4-byte floating-point (7-place accuracy)

Double-length floating-point numbers:

Byte number	Value	Definition
1	31	An 8-byte floating-point number follows
2-9	xx	8-byte floating-point (16-place accuracy)

### Notation

The Amiga has ways to recognize and fix incorrect numerical notation. Enter the following into a program from AmigaBASIC:

```
a=&hff
```

When you exit the line, the Amiga corrects the error:

```
a=&HFF
```

Tokens help the Amiga recognize the number system used:

Byte number	Value	Definition
1	12	Hexadecimal number follows
2	xx	High byte
3	xx	Low byte

Then there are the larger octal numbers like &O123456. These must be converted into 2-byte format:

Byte number	Value	Definition
1	11	Octal number follows
2 + 3	xx	Octal number (accuracy to 6 places)



Assigning values to strings has one major change from the other variables: Strings are stored in ASCII. To save memory, no new memory is set aside for a direct value assignment. In the program, the pointer is set to the starting address of the string.

For example, type this in AmigaBASIC and run it:

```
a$="-----"
b$="These lines I am a'changing."
FOR i=1 TO LEN(b$)
  POKE SADD(a$)+i-1,ASC(MID$(b$,i,1))
NEXT
LIST
```

SADD may be an unfamiliar command to you. It returns the starting address of the string contained in a variable (in this case a\$).

After you run this program, compare the listing above with the program you entered and ran. It looks like this:

```
a$="These lines I am a'changing."
b$="These lines I am a'changing."
FOR i=1 TO LEN(b$)
  POKE SADD(a$)+i-1,ASC(MID$(b$,i,1))
NEXT
LIST
```

You can see from this small example, there is potential for self-modifying programs. For example, you could put the name of a window in a\$. The user could enter a new name while the program runs. The program then POKES the name into the system and saves the altered program to diskette.

### *Command tokens*

Command tokens (characters having ASCII codes higher than 127) have their own peculiarities that you should know about. These tokens are stored by AmigaBASIC as single- or double-character codes. They represent direct commands, but require less memory than if the Amiga stored commands by their full names.

\$8E (ELSE) never happens in program code by itself. The interpreter can only determine the end of a command when it either finds code \$00 (end-of-line) or code \$3A (colon). If the interpreter finds IF and THEN without an ELSE, then IF/THEN are handled by the interpreter as one command. If ELSE follows, you can see that the BASIC interpreter adds a colon before the \$8E (you can't see this colon when you call LIST). If you put your own colons in preceding the ELSES in your programs, the file monitor shows two colons. The colon originally added by the interpreter itself is invisible to LIST.

REMARKS cause a similar problem—the interpreter adds a colon. This is strange, since it happens even when REM is the only command in the line. A line can look like this:

```
'*1.*
```

Its structure can look like this:

00	0E	00	3A	AF	E8	20	2A	20	31	2E	20	2A	00
Header	:	'	*	1	.	*	End						

Another strange thing happens when you create a program and use the token \$BE for the WHILE command. Under certain circumstances, the Amiga stops the program and returns ERROR 22 (Missing operand). If you write a program in AmigaBASIC, once in a while the interpreter places an \$EC after the visible single-byte token \$BE.

**Important:**

There is one token that you can't list and you almost never use. You know that you can only call SUB routines directly through THEN or ELSE with the CALL command. You can use BASIC commands as well as SUB programs. The SUB program has one purpose alone: It allows the programming of command extensions in BASIC. Those who know this never use the CALL command, aside from calling operating system routines. Instead they use this token. Unlike CALL, this token goes after the pointers to the variable table. The token is the double token \$F8-\$D1.

In closing, a few words about the DATA command. DATA statements are placed in ASCII text, like the data following a REMARK. This data can be read into variables, and can be of any type:

```
DATA &hffe2,123,&06666
```

**SUB programs**

Why were the SUB programs implemented in AmigaBASIC? The first reason is that they allow modular programming. Also, SUB programs allow the retention of variable names, even when programs are combined through CHAIN and MERGE. Any of these variables can be shared with other routines by stating the names with STATIC.

It's a good idea to edit each and every SUB program separately and store them as ASCII files. Then combine the SUBS with the BASIC program currently in memory using MERGE in direct mode or program mode (the syntax check requires a lot of time). The call convention (e.g., which operating system routines must be declared as functions, etc.) should be declared and archived with a file manager. The second point of interest was that unlike earlier computers with incomplete command sets, SUB programs allow extension of the command set:

```
PRINTAT 10,20,"Sample text"

SUB PRINTAT (x,y,Text$) STATIC
  LOCATE y,x
```

```
    PRINT Text$
END SUB
```

The third point is the pressure on the programmer to learn Pascal or another language. Why learn more complex languages, when BASIC can do it just as well and just as fast? Unlike Pascal, SUB programs cannot call themselves. However, a command can be called multiple times by using a label at the beginning of a routine made up of SUB programs.

Programs handle SUB routines like variables. This is the only way the Amiga recognizes these routines.

### *Important details*

What would the make-believe manipulation program do when it encounters the code sequence \$20-\$F8-\$8F-\$20? Turn to the token list in the Appendix. The code stands for the \$F8 double token END, placed between two spaces. The program hasn't ended, though. What about this \$F8-\$BE? That's the double code for SUB. You see, a token by itself can cause trouble. First the connection in which the token is compared to other tokens sets the type of execution. This also goes for PRINT# and ?#—the token numbers are the same.

### *Other tokens*

No time has been spent discussing tokens below 128. These tokens are used, though. There are occasions when you try saving an edited program in direct mode when the Amiga displays an error requester instead. Apparently the Amiga gets stuck in the error checking routine, and keeps registering an error. Clicking on the OK gadget eventually gets you past the error, but you may have to click it a few times over.

A simple program check can change commands around. An occasional gap in the token list can control the program. For example, \$8, which acts as the branch offset of the IF/THEN construction that may not be in the same place in another program. In order to make life with manipulation programs as simple as possible, try to follow these ground rules:

1. Manipulation programs or programs for reading data from other programs which require binary file format should:
  - Allow storage of the modified file as an ASCII file.
  - Allow you to save the file back in binary file format after loading.
2. ASCII files require no special treatment, as long as the program control codes aren't saved as well.

## 5.3 Utility programs

The following section presents programs that let you change AmigaBASIC program code.

### 5.3.1 DATA generator

This program demonstrates how you can create an AmigaBASIC program saved in ASCII format. This program makes DATA statements out of any file on diskette. This can be used to include sprites, bobs and machine language directly in your AmigaBASIC programs. This could be used to produce program listing for inclusion in a book. The ASCII file created can be appended to a program using MERGE.

To keep the DATA list short, the DATA statements are displayed in hexadecimal notation. You may recognize the reader routine from the AmigaBASIC manual program for converting hex to decimal numbers. The reverse routine can be found anywhere, although it's not standard to AmigaBASIC. Just type:

```
stuff: DATA ff,ec,0,1,f
RESTORE stuff:FOR i=1 TO 5:READ a$:x(i)=VAL("&H"+a$):NEXT
```

Now for the listing:

```
GOTO Start¶
' #####
' # D A T A - G E N E R A T O R   A M I G A   #¶
' #-----#¶
' #           (W) 1987 by Stefan Maelger     #¶
' #####
'¶
' "dos.bmap" and "exec.bmap" must be on¶
' Disk or in LIBS: !¶
' -----¶
' Declare System Routines and Functions¶
'¶
Start: ¶
  DECLARE FUNCTION xOpen&      LIBRARY¶
  DECLARE FUNCTION xRead&     LIBRARY¶
  DECLARE FUNCTION AllocMem&  LIBRARY¶
  DECLARE FUNCTION Examine&   LIBRARY¶
  DECLARE FUNCTION Lock&      LIBRARY¶
' -----¶
' Open Libraries¶
```

```

'  ¶
LIBRARY "T&T2:bmaps/exec.library"¶
LIBRARY "T&T2:bmaps/dos.library"¶
' -----¶
' Input¶
' ¶
sourcefile:¶
CLS¶
LINE INPUT "Name of Source-File: ";source$¶
PRINT¶
PRINT "Insert Diskette and Press <RETURN>"¶
WHILE A$<>CHR$(13)¶
  A$=INKEY$¶
WEND¶
LOCATE 3,1:PRINT "Checking File..."¶
CHDIR "df0:"¶
CheckFile source$,Bytes&¶
¶
¶
IF Bytes&=0 THEN¶
  LOCATE 3,1:PRINT "File not found...":BEEP¶
  A=TIMER+3 :WHILE A>TIMER:WEND¶
  GOTO sourcefile¶
ELSEIF Bytes&=-1 THEN¶
  LOCATE 3,1:PRINT "I can't find the Directory..."¶
  BEEP :A=TIMER+3:WHILE A>TIMER:WEND¶
  GOTO sourcefile¶
END IF¶
LOCATE 3,1:PRINT "File Found. Length=";Bytes&," Byte"¶
' -----¶
' Setup Buffer¶
' ¶
PublicRAM&=65537&¶
Buffer&=AllocMem&(Bytes&,PublicRAM&)¶
IF Buffer&=0 THEN¶
  LOCATE 5,1:PRINT "Not enough memory."¶
  LOCATE 7,1¶
  PRINT "Program can re-started with RUN."¶
  BEEP :END¶
END IF¶
' -----¶
' Load File in Buffer¶
' ¶
source$=source$+CHR$(0)¶
Opened&=xOpen&(SADD(source$),1005)¶
IF Opened&=0 THEN¶
  LOCATE 5,1:PRINT "I can not open the File!"¶
  BEEP :A=TIMER+3:WHILE A>TIMER:WEND¶
  GOTO sourcefile¶
END IF¶
sofar%=xRead%(Opened&,Buffer&,Bytes&)¶
CALL xClose(Opened&)¶
' -----¶
' Input Target-File¶
' ¶
targetfile: ¶

```

```

LOCATE 9,1:PRINT "Name of BASIC-ASCII-File"¶
¶
FOR i=11 TO 17 STEP 2¶
  LOCATE i,1:PRINT SPACES(80)¶
NEXT¶
LOCATE 11,1:LINE INPUT "to be produced: ";target$¶
LOCATE 13,1:PRINT "Insert Target-Disk and Press
<RETURN>"¶
A$="" :WHILE A$<>CHR$(13):A$=INKEY$:WEND¶
CHDIR "df0:"¶
LOCATE 15,1:PRINT "Checking Disk..."¶
CheckFile target$,exist&¶
IF exist&=-1 THEN¶
  LOCATE 15,1:PRINT "This is the Name of a Directory!"¶
  BEEP :A=TIMER+3:WHILE A>TIMER:WEND¶
  GOTO targetfile¶
ELSEIF exist&<>0 THEN¶
  LOCATE 15,1:PRINT "A File with that name already"¶
  LOCATE 17,1:PRINT "exists! Replace File? (Y/N)"¶
pause: ¶
A$=INKEY$ :IF A$<>" " THEN A$=UCASE$(A$)¶
IF A$="Y" GOTO continue¶
IF A$<>"N" GOTO pause¶
GOTO targetfile¶
END IF¶
continue:¶
' -----¶
' Produce DATA-ASCII-File¶
'¶
LOCATE 19,1:PRINT "Producing ASCII-File."¶
LOCATE 21,1:PRINT "Please be Patient..."¶
OPEN target$ FOR OUTPUT AS 1¶
  Number&=0¶
  PRINT#1,"RESTORE datas";CHR$(10);¶
  PRINT#1,"datastring$=";CHR$(34);CHR$(34);CHR$(10);¶
  PRINT#1,"FOR i=1 TO ";STR$(Bytes&);CHR$(10);¶
  PRINT#1,"READ a$";CHR$(10);¶
  PRINT#1,"a$=";CHR$(34);"&H";CHR$(34);"a$";CHR$(10);¶
  PRINT#1,"datastring$=datastring$+CHR$(VAL(a$))";¶
  PRINT#1,CHR$(10);¶
  PRINT#1,"NEXT";CHR$(10);¶
  PRINT#1,"datas:";CHR$(10);¶
¶
Loop:¶
  PRINT#1,"DATA ";¶
  BCount=0¶
Value: ¶
  PRINT#1,HEX$(PEEK(Buffer&+Number&));¶
  BCount=BCount+1 :Number&=Number&+1¶
  IF Number&<Bytes& THEN¶
    IF BCount<20 THEN ¶
      PRINT#1,",";¶
      GOTO Value¶
    ELSE¶

```

```

        PRINT#1,CHR$(10);¶
        GOTO Loop¶
    END IF¶
    END IF¶
    PRINT#1,CHR$(10);CHR$(10);¶
    CLOSE 1¶
'-----¶
' Alter .info-file¶
'¶
    SAVE "DATA-GENINFO"¶
    tmp$=target$+".info"¶
    KILL tmp$¶
    NAME "DATA-GENINFO.info" AS target$+".info"¶
    KILL "DATA-GENINFO"¶
    CLS¶
    PRINT "finished."¶
    CALL FreeMem(Buffer&,Bytes&)¶
    END¶
'-----¶
' SUBROUTINE¶
'¶
SUB CheckFile(FileName$,Length&) STATIC¶
    ChipRAM&=65538&¶
    InfoBytes&=252¶
    Info&=AllocMem&(InfoBytes&,ChipRAM&)¶
    IF Info&=0 THEN ERROR 7¶
    File$=FileName$+CHR$(0)¶
    ¶
    DosLock&=Lock&(SADD(File$),-2)¶
    IF DosLock&=0 THEN¶
        Length&=0¶
    ELSE¶
        Dummy&=Examine&(DosLock&,Info&)¶
        Length&=PEEKL(Info&+4)¶
        IF Length&>0 THEN ¶
            Length&=-1¶
        ELSE¶
            Length&=PEEKL(Info&+124)¶
        END IF¶
    END IF¶
    CALL UnLock(DosLock&)¶
    CALL FreeMem(Info&,InfoBytes&)¶
END SUB¶

```

**Variables**

A	string, help variable
AllocMem	EXEC routine; reserves memory
Buffer	address of reserved memory
Bytes	length of file being edited
CheckFile	SUB routine; tests for file availability: if yes, then it checks for directory; if not, it checks for length
ChipRAM	option for AllocMem; 2 <sup>16</sup> (65536)=clear range, 2 <sup>1</sup> (2)=chip RAM range
DosLock	file handle for Checkfile routine
Dummy	unused variable

Examine	DOS routine; looks for file
File	filename with concluding 0 for DOS
Filename	name of file being edited
FreeMem	EXEC routine; frees memory range
Info	address of file info structure
InfoBytes	length of file info structure
Length	file length
Lock	DOS routine; blocks access from other programs and provides handle
Opened	address of file handle for source file
PublicRAM	option for AllocMem; 2^16 (65536)=clear range, 2^1(2)=public range
UnLock	DOS routine; releases Lock
Number&	counter for DATA values written
sofar&	number of bytes read so far
i	loop variable
source	source file
target	target file in ASCII format for DATA
exist&	flag:does file exist?
Tmp	help variable - temporary file
xClose	DOS routine; closes file
xOpen	DOS routine; opens file
xRead	DOS routine; reads file
BCount	byte counter for a line of DATA

### 5.3.2 Cross-reference list

This program demonstrates a method of reading values from AmigaBASIC programs stored in binary format. However, you must first remove the onboard program control codes, any program "garbage" that can occur between the program body and the variable table of the program you wish to read. Do the following to clean up the program code:

- Load the file you want to check
- SAVE "Filename",A
- Quit AmigaBASIC
- Reload AmigaBASIC
- LOAD "Filename"
- SAVE "Filename",B



Once you do this, you can now send a cross-reference list of this program to a printer using the program below. It displays labels as well as line numbers in the output. Places where branches are set (e.g., GOTO place) are marked by "<- -". If a branch goes to a section of a program not set by a branch marker (e.g., the beginning of a program), a pseudo label appears in parentheses (e.g., "(Program start)"). A "->" marks the destination of the branch. Bear in mind that operating system calls and SUB routines are viewed by the AmigaBASIC interpreter as variables. Aside from that, this program is a great method of documenting your programs.

```
' #####
' # C r o s s R e f e r e n c e   A m i g a
' #-----#
' #           (W) 1987 by Stefan Maelger           #
' #####
'
' This program creates a Cross-Reference
' of a program on your Printer.
' It allows every BINARY format
' AmigaBASIC-Program to be documented.
' -----
' How the AmigaBASIC programmer handled
' SUB-Routines and System calls is still
' not well known.
' -----
'
' ----Reserve Memory, load PrinterDriver, ----
' ----Open Library and Variables----
CLEAR,45000&
LPRINT
DECLARE FUNCTION xOpen& LIBRARY
DECLARE FUNCTION xRead% LIBRARY
DECLARE FUNCTION Seek% LIBRARY
LIBRARY "T&T2:bmaps/dos.library"
DIM Cross$(5000),names$(1000)
'
LOCATE 2,2
PRINT CHR$(187);" Cross Reference Amiga ";CHR$(171)
LOCATE 5,2
PRINT "Name of the binary AmigaBASIC-Program:"
LOCATE 7,2
LINE INPUT Filename$
CHDIR "df0:"
'
BASICcheck Filename$,Result%
'
LOCATE 10,2
IF Result%=-1 THEN
PRINT "I can not find any Info-File."
ELSEIF Result%=0 THEN
PRINT "Read-Error!"
ELSEIF Result%=1 THEN
PRINT "This is Not an AmigaBASIC-Program."
END IF
```

```

IF Result%<>2 THEN¶
  BEEP¶
  WHILE INKEY$=""¶
    WEND¶
  RUN¶
END IF¶
PRINT CHR$(34);Filename$;".info";CHR$(34)¶
PRINT ¶
PRINT " made with this Program as AmigaBASIC-File."¶
¶
OpenFile Filename$,handle&¶
¶
LOCATE 14,2¶
IF handle&=0 THEN¶
  PRINT "AAAaargh! I can't find ";CHR$(34);¶
  PRINT Filename$;CHR$(34);"!!!"¶
  BEEP¶
  WHILE INKEY$="" :WEND :RUN¶
ELSE¶
  PRINT "File opened."¶
END IF¶
LOCATE 16,2¶
¶
HeaderCheck handle&,Header$¶
¶
IF ASC(Header$)<>&HF5 THEN¶
  PRINT "Sorry, I can only Cross-Reference binary-Files"¶
  BEEP¶
  WHILE INKEY$="" :WEND :RUN¶
ELSE¶
  PRINT "File has binary Format"¶
  PRINT :PRINT "Please be patient. ";¶
  PRINT "I'll report on my status..."¶
END IF¶
pointer%=-1¶
¶
main:¶
¶
GetLine handle&,Current$¶
¶
IF LEN(Current$)<4 THEN¶
  PRINT ¶
  PRINT " Reached the end of Binary-Codes"¶
  PRINT :PRINT " getting Variable Table."¶
  GOTO Vartab¶
END IF¶
IF ASC(Current$)=128 THEN¶
  pointer%=pointer%+1¶
  Cross$(pointer%)=CHR$(128)+MID$(Current$,4,2)¶
  Current$=MID$(Current$,6)¶
ELSE¶
  Current$=MID$(Current$,4)¶
END IF¶
¶
GetToken:¶
¶

```

```

Token%=ASC(Current$+CHR$(0)) ¶
IF Token%=0 GOTO main¶
¶
'----Command Token?---- ¶
IF Token%>127 THEN¶
  IF Token%=175 OR Token%=141 GOTO main¶
  IF Token%=190 OR Token%>247 THEN¶
    Current$=MID$(Current$,3) ¶
  ELSE¶
    Current$=MID$(Current$,2) ¶
  END IF¶
  GOTO GetToken¶
END IF¶
¶
'----String?---- ¶
IF Token%=34 THEN¶
  Byte%=INSTR(2,Current$,CHR$(34)) ¶
  IF Byte%=0 GOTO main¶
  Current$=MID$(Current$,Byte%+1) ¶
  GOTO GetToken¶
END IF¶
¶
'---- 2-Byte-Value Sequence?---- ¶
IF Token%=1 OR Token%=11 OR Token%=12 OR Token%=28 THEN¶
  Current$=MID$(Current$,4) ¶
  GOTO GetToken¶
END IF¶
¶
'---- 1-Byte-Value Sequence?---- ¶
IF Token%=15 THEN Current$=MID$(Current$,3):GOTO
GetToken¶
¶
'---- 4-Byte-Value Sequence?---- ¶
IF Token%=29 OR Token%=30 THEN¶
  Current$=MID$(Current$,6) ¶
  GOTO GetToken¶
END IF¶
¶
'---- 8-Byte-Value Sequence?---- ¶
IF Token%=31 THEN Current$=MID$(Current$,10):GOTO
GetToken¶
¶
'---- Is it a Label?---- ¶
IF Token%=2 THEN¶
  pointer%=pointer%+1¶
  Cross$(pointer%)=LEFT$(Current$,3) ¶
  Current$=MID$(Current$,4) ¶
  GOTO GetToken¶
END IF¶
¶
'---- Is it a Branch Statement?---- ¶
IF Token%=3 OR Token%=14 THEN¶
  pointer%=pointer%+1¶
  Cross$(pointer%)=CHR$(Token%)+MID$(Current$,3,2) ¶
  Current$=MID$(Current$,5) ¶
  GOTO GetToken¶

```

```

END IF¶
Current$=MID$(Current$,2)¶
GOTO GetToken¶
¶
Vartab:¶
¶
p2%=-1¶
¶
notforever:¶
¶
  GetLength handle&,bytes%¶
¶
  IF bytes%=0 GOTO GoOn¶
¶
  GetName handle&,Current$,bytes%¶
¶
  p2%=p2%+1¶
  names$(p2%)=Current$¶
  GOTO notforever¶
¶
GoOn:¶
¶
  IF pointer%=-1 THEN¶
    PRINT ¶
    PRINT "I have no Label or Line Number"¶
    PRINT ¶
    PRINT "that I can discover!"¶
    BEEP¶
    WHILE INKEY$="" :WEND:RUN¶
  ELSEIF p2%=-1 THEN¶
    PRINT ¶
    PRINT "Hmm - no Variable Table"¶
    BEEP¶
    WHILE INKEY$="" :WEND:RUN¶
  ELSE ¶
    PRINT :PRINT " Getting Data."¶
  END IF¶
¶
LPRINT ">>> CrossReference Amiga <<<"¶
LPRINT "-----"¶
LPRINT "Program: ";Filename$¶
LPRINT¶
FOR i=0 TO pointer%¶
  ascii%=ASC(Cross$(i))¶
  IF ascii%=2 THEN¶
    LPRINT names$(CVI(MID$(Cross$(i),2)));" : "¶
    FOR j=0 TO pointer%¶
      IF ASC(Cross$(j))=3 THEN¶
        IF CVI(MID$(Cross$(j),2))=CVI(MID$(Cross$(i),2))
THEN¶
          k=j¶
          WHILE k>-1¶
            k=k-1¶
            IF k>-1 THEN¶
              IF ASC(Cross$(k))=2 THEN¶
                LPRINT " <-- ";¶

```

```

        LPRINT names$(CVI(MID$(Cross$(k),2))) ¶
        k=-2¶
        ELSEIF ASC(Cross$(k))=128 THEN¶
            LPRINT " <-- ";CVI(MID$(Cross$(k),2)) ¶
            k=-2¶
        END IF¶
    END IF¶
WEND ¶
IF k=-1 THEN LPRINT " <-- (Program-Start)"¶
END IF¶
END IF¶
NEXT j¶
ELSEIF ascii%=3 THEN¶
    LPRINT " --> ";names$(CVI(MID$(Cross$(i),2))) ¶
ELSEIF ascii%=14 THEN¶
    LPRINT " --> ";CVI(MID$(Cross$(i),2)) ¶
ELSEIF ascii%=128 THEN¶
    LPRINT CVI(MID$(Cross$(i),2)) ¶
    FOR j=0 TO pointer%¶
        IF ASC(Cross$(j))=14 THEN¶
            IF CVI(MID$(Cross$(j),2))=CVI(MID$(Cross$(i),2))
THEN¶
                k=j¶
                WHILE k>-1¶
                    k=k-1¶
                    IF k>-1 THEN¶
                        IF ASC(Cross$(k))=2 THEN¶
                            LPRINT " <-- ";¶
                            LPRINT names$(CVI(MID$(Cross$(k),2))) ¶
                            k=-2¶
                        ELSEIF ASC(Cross$(k))=128 THEN¶
                            LPRINT " <-- ";CVI(MID$(Cross$(k),2)) ¶
                            k=-2¶
                        END IF¶
                    END IF¶
                WEND ¶
                IF k=-1 THEN LPRINT " <-- (Program-Start)"¶
                END IF¶
            END IF¶
        NEXT j¶
    END IF¶
NEXT i¶
PRINT :PRINT "Finished."¶
BEEP¶
WHILE INKEY$="":WEND:RUN¶
¶
SUB GetName(handle%,Current$,bytes%) STATIC¶
    Current$=SPACE$(bytes) ¶
    Length%=xRead$(handle$,SADD(Current$),bytes) ¶
END SUB¶
¶
SUB GetLength(handle%,bytes%) STATIC¶
    Current$=CHR$(0) ¶
readit: ¶
    Length%=xRead$(handle$,SADD(Current$),1) ¶
    IF Length%=0 THEN¶

```

```

        CALL xClose(handle&)¶
        bytes%=0¶
        EXIT SUB¶
    END IF¶
    bytes%=ASC(Current$)¶
    IF bytes%=0 THEN readit¶
    IF bytes%>60 THEN readit¶
    ¶
END SUB¶
    ¶
SUB GetLine(handle&,Current$) STATIC¶
    Current$=STRING$(3,0)¶
    Length%=xRead%(handle&,SADD(Current$),3)¶
    OldPos%=Seek%(handle&,-3,0)¶
    LoL%=ASC(MID$(Current$,2,1))¶
    IF LoL%=0 THEN¶
        EXIT SUB¶
    ELSE¶
        Current$=STRING$(LoL%,0)¶
        Length%=xRead%(handle&,SADD(Current$),LoL%)¶
    END IF¶
END SUB¶
    ¶
SUB HeaderCheck(handle&,Header$) STATIC¶
    Header$="1"¶
    OldPos%=Seek%(handle&,0,-1)¶
    gotit%=xRead%(handle&,SADD(Header$),1)¶
END SUB¶
    ¶
SUB OpenFile(FileName$,handle&) STATIC¶
    file$=FileName$+CHR$(0)¶
    handle&=xOpen$(SADD(file$),1005)¶
END SUB¶
    ¶
SUB BASICcheck(FileName$,Result%) STATIC¶
    file$=FileName$+".info"+CHR$(0)¶
    Default.Tool$=SPACE$(20)¶
    handle&=xOpen$(SADD(file$),1005)¶
    IF handle&=0 THEN¶
        Result%=-1¶
    ELSE¶
        OldPos%=Seek%(handle&,-20,1)¶
        gotit%=xRead%(handle&,SADD(Default.Tool$),20)¶
        IF gotit%<20 THEN¶
            Result%=0¶
        ELSE¶
            IF INSTR(Default.Tool$,"AmigaBASIC")>0 THEN¶
                Result%=2¶
            ELSE¶
                Result%=1¶
            END IF¶
        END IF¶
        CALL xClose(handle&)¶
    END IF¶
END SUB¶

```

<b>Variables</b>	BASICcheck	SUB routine; test for Default Tools
	Byte	pointer to byte in string
	Bytes	length of file being edited
	Cross	string array; buffer for branch markers and jumps
	Current	string; BASIC line read
	Default.Tool	string; reads Default Tool
	Filename	string; name of file to be edited
	GetLength	SUB routine; reads label length
	GetLine	SUB routine; reads line
	GetName	SUB routine; reads label name
	Header	string; file header byte
	HeaderCheck	SUB routine; checks for header type
	Length	file length
	LoL	line length
	OldPos	old pointer position in file
	OpenFile	SUB routine; opens file
	Result	flag; result of search
	Seek	DOS routine; moves read/write pointer in file
	Token	address of file handle for source file
	ascii	code value in Cross\$
	File	string; filename ended with 0 for DOS routines
	gotit	bytes read so far
	handle	file handle address
	i	loop variable
	j	loop variable
	k	loop variable
	names	string array; branch marker names
	p2	help variable
	pointer	help variable
	xClose	DOS routine; closes file
	xOpen	DOS routine; opens file
	xRead	DOS routine; reads file

### 5.3.3 Blank line killer

Now that you know how to make blank lines, you should know how to get rid of them. The following program removes these lines for you. Before using this program, any control codes and garbage must be removed (see the preceding section for instructions on doing this).

**Note:**

When you type in this program, you could create small errors that can ruin the programs being modified. Use *copies* of the program you want to modify only, and test the main program with these copies to make sure that it runs properly. This program alters the file and saves it out again. The current window closes to save memory. If there are small errors in the line killer program, such as an endless loop, you won't be able to recover the program. If the program seems as if it's taking a while at first, don't panic—the time factor depends on the file being modified.

```
' #####
' #          Blank Line-Killer  Amiga          #
' #-----#
' #          (W) 1987 by Stefan Maelger        #
' #####
'
' "dos.bmap" and "exec.bmap" must be on
' Disk or in LIBS:
'-----
'
DECLARE FUNCTION AllocMem& LIBRARY
DECLARE FUNCTION Lock&      LIBRARY
DECLARE FUNCTION Examine&  LIBRARY
DECLARE FUNCTION xOpen&    LIBRARY
DECLARE FUNCTION xRead&    LIBRARY
DECLARE FUNCTION xWrite&   LIBRARY
LIBRARY "T&T2:bmaps/exec.library"
LIBRARY "T&T2:bmaps/dos.library"
WINDOW CLOSE WINDOW(0)
WINDOW 1,"Blank Line-Killer", (0,0)-(250,50),16
Allocation.1:
COLOR 3,1:CLS
info&=AllocMem&(252&,65538&)
IF info&=0 THEN
  ALLOCERR
  GOTO Allocation.1
END IF
Source:
REQUEST "SOURCE"
SELECT box%
IF box% THEN CALL FreeMem(info&,252):SYSTEM
CHDIR "df0:"
GetFilename:
LINPUT Filename$
GETINFO Filename$,info&,Length&
IF Length&<1 THEN
  IF Length&=-1 THEN
    DIRERR
  ELSEIF Length&=0 THEN
    FILEERR
  END IF
  GOTO GetFilename
END IF
Allocation.2:
COLOR 3,1:CLS
```



```

buffer&=AllocMem&(Length&,65537&)¶
IF buffer&=0 THEN¶
  ALLOCERR¶
  GOTO Allocation.2¶
END IF¶
LOADFILE Filename$,buffer&,Length&¶
IF Filename$="" THEN¶
  CALL FreeMem(buffer&,Length&)¶
  LOADERR¶
  GOTO GetFilename¶
END IF¶
IF PEEK(buffer&)<>&HF5 THEN¶
  CALL FreeMem(buffer&,Length&)¶
  FORMERR¶
  GOTO GetFilename¶
END IF¶
NEWFILE Filename$,handle&¶
IF handle&=0 THEN¶
  CALL FreeMem(buffer&,Length&)¶
  CALL FreeMem(info&,252&)¶
  OPENERR¶
  SYSTEM¶
END IF¶
Bytes&=1¶
DWRITE handle&,buffer&,Bytes&¶
IF Bytes&=0 THEN¶
  CALL xClose(handle&)¶
  CALL FreeMem(buffer&,Length&)¶
  CALL FreeMem(info&,252&)¶
  WRITEERR¶
  SYSTEM¶
END IF¶
pointer&=buffer&+1¶
GetLength:¶
Bytes&=PEEK(pointer&+1)¶
IF Bytes&=4 THEN¶
  pointer&=pointer&+4¶
  GOTO GetLength¶
ELSEIF Bytes&>4 THEN¶
  DWRITE handle&,pointer&,Bytes&¶
  IF Bytes&=0 THEN¶
    CALL xClose(handle&)¶
    CALL FreeMem(buffer&,Length&)¶
    CALL FreeMem(info&,252&)¶
    WRITEERR¶
    SYSTEM¶
  END IF¶
  pointer&=pointer&+Bytes&¶
  GOTO GetLength¶
ELSE¶
  Bytes&=Length&-(pointer&-buffer&+1)¶
  DWRITE handle&,pointer&,Bytes&¶
  IF Bytes&=0 THEN¶
    CALL xClose(handle&)¶
    CALL FreeMem(buffer&,Length&)¶
    CALL FreeMem(info&,252&)¶

```

```

        WRITEERR¶
        SYSTEM¶
    END IF¶
    END IF¶
    CALL xClose(handle&)¶
    CALL FreeMem(buffer&,Length&)¶
    CALL FreeMem(info&,252&)¶
    LIBRARY CLOSE¶
    COLOR 3,1:CLS:LOCATE 2,2:PRINT "Ready."¶
    WHILE INKEY$="" :WEND¶
    SYSTEM¶
SUB WRITEERR STATIC¶
    COLOR 1,3:CLS:LOCATE 2,2:PRINT "ERROR: Write-error."¶
    ShowCont¶
END SUB ¶
SUB DWRITE(handle&,adr&,Length&) STATIC¶
    written&=xWrite&(handle&,adr&,Length&)¶
    IF written&<>Length& THEN Length&=0¶
END SUB¶
SUB OPENERR STATIC¶
    COLOR 1,3:CLS:LOCATE 2,2:PRINT "ERROR: Can't open
File."¶
    ShowCont¶
END SUB ¶
SUB NEWFILE(Filename$,handle&) STATIC¶
    File$=Filename$+CHR$(0)¶
    handle&=xOpen&(SADD(File$),1005)¶
END SUB ¶
SUB FORMERR STATIC¶
    COLOR 1,3:CLS:LOCATE 2,2:PRINT "ERROR: Not a binary
File."¶
    ShowCont¶
END SUB ¶
SUB LOADERR STATIC¶
    COLOR 1,3:CLS:LOCATE 2,2:PRINT "ERROR: Load-error."¶
    ShowCont¶
END SUB¶
SUB LOADFILE(Filename$,buffer&,Length&) STATIC¶
    File$=Filename$+CHR$(0)
:handle&=xOpen&(SADD(File$),1005)¶
    IF handle&=0 THEN¶
        Filename$=""¶
    ELSE ¶
        inBuffer&=xRead&(handle&,buffer&,Length&)¶
        CALL xClose(handle&)¶
        IF inBuffer&<>Length& THEN Filename$=""¶
    END IF¶
END SUB¶
SUB FILEERR STATIC¶
    COLOR 1,3:CLS:LOCATE 2,2:PRINT "ERROR: File not
found."¶
    ShowCont¶
END SUB ¶
SUB DIRERR STATIC¶
    COLOR 1,3:CLS:LOCATE 2,2¶
    PRINT "ERROR: File is a Directory."¶

```

```

    ShowCont¶
END SUB¶
SUB GETINFO(Filename$,info&,Length&) STATIC¶
    File$=Filename$+CHR$(0) :DosLock&=Lock&(SADD(File$),-
2)¶
    IF DosLock&=0 THEN          ¶
        Length&=0¶
    ELSE¶
        Dummy&=Examine&(DosLock&,info&)¶
        IF PEEKL(info&+4)>0 THEN¶
            Length&=-1¶
        ELSE¶
            Length&=PEEKL(info&+124)¶
        END IF¶
    END IF¶
    CALL UnLock(DosLock&)¶
END SUB¶
SUB LINPUT(Filename$) STATIC¶
    COLOR 3,1:CLS:WINDOW 2,"Filename:",(0,0)-(250,10),0¶
    WINDOW OUTPUT 1:LOCATE 5,2¶
    PRINT "Name of a binary saved File";¶
    LINE INPUT Filename$:WINDOW CLOSE 2¶
END SUB¶
SUB SELECT(box%) STATIC¶
Check: ¶
    WHILE MOUSE(0)=0:WEND:x=MOUSE(1):y=MOUSE(2)¶
    IF y>27 AND y<43 THEN¶
        IF x>9 AND x<38 THEN box%=0:EXIT SUB¶
        IF x>177 AND x<238 THEN box%=-1:EXIT SUB¶
    END IF¶
    GOTO Check¶
END SUB¶
SUB ALLOCERR STATIC¶
COLOR 1,3:CLS:LOCATE 2,2:PRINT "ERROR: Allocation
denied."¶
    ShowCont¶
END SUB¶
SUB ShowCont STATIC¶
    LOCATE 4,2:PRINT "Press SPACE to continue,"¶
    LOCATE 5,7:PRINT "ESCAPE to exit.";¶
    WHILE a$<>CHR$(32) AND a$<>CHR$(27)¶
        a$=INKEY$¶
    WEND¶
    IF a$=CHR$(27) THEN SYSTEM¶
END SUB¶
SUB REQUEST(disk$) STATIC¶
    COLOR 3,1:CLS¶
    LOCATE 2,2:PRINT "INSERT ";disk$;" DISK INTO DRIVE"¶
    LOCATE 3,14:PRINT "DF0:";LOCATE 5,3:PRINT "OK";¶
    LOCATE 5,24:PRINT "CANCEL";:LINE(10,28)-(37,42),3,b¶
    LINE(178,28)-(237,42),3,b¶
END SUB¶

```

<i>Variables</i>	ALLOCERR	SUB routine; memory reservation error
	AllocMem	EXEC routine; reserves memory
	Bytes	length of file being edited
	DIERR	SUB routine; error—no file
	DWRITE	SUB routine; write to file
	DosLock	file handle of Lock
	Dummy	unused variable
	Examine	DOS routine; looks for file
	FILEERR	SUB routine; error
	FORMERR	SUB routine; error
	File	filename with concluding 0 for DOS
	Filename	name of file being edited
	FreeMem	EXEC routine; frees memory range
	GETINFO	SUB routine; file check
	LINPUT	SUB routine; input
	LOADERR	SUB routine; error
	LOADFILE	SUB routine; load program
	Length	file length
	Lock	DOS routine; blocks access from other programs and provides handle
	NEWFILE	SUB routine; create new file
	OPENERR	SUB routine; error
	REQUEST	SUB routine; draw primitive requester
	SELECT	SUB routine; select through mouse click
	ShowCont	SUB routine; show options
	UnLock	DOS routine; releases Lock
	WRITEERR	SUB routine; error
	a	help variable
	adr	address
	b	help variable
	box	help variable
	buffer	address of reserved memory
	disk	diskette
	handle	address of file handle
	inBuffer	bytes read
	info	address of file info structure
	pointer	help variable
	written	bytes written
	x	help variable
	xClose	DOS routine; closes file
	xOpen	DOS routine; opens file
	xRead	DOS routine; reads file
	xWrite	DOS routine; writes to file
	y	help variable

### 5.3.4 REM killer

This program has a lot of the same code as the line killer in Section 5.3.3. Load that program, change the necessary text and save the new program under a different name from the name you assigned in Section 5.3.3.

```
' #####
' #   K i l l - R e m a r k   A m i g a   #
' #-----#
' #           (W) 1987 by Stefan Maelger   #
' #####
'
' "dos.bmap" and "exec.bmap" must be on
' Disk or in LIB:
' -----
'
  DECLARE FUNCTION AllocMem& LIBRARY
  DECLARE FUNCTION Lock&      LIBRARY
  DECLARE FUNCTION Examine&  LIBRARY
  DECLARE FUNCTION xOpen&    LIBRARY
  DECLARE FUNCTION xRead&    LIBRARY
  LIBRARY "T&T2:bmaps/exec.library"
  LIBRARY "T&T2:bmaps/dos.library"
  WINDOW CLOSE WINDOW(0)
  WINDOW 1,"Kill-Remark", (0,0)-(250,50),16
Allocation.1:
  COLOR 3,1:CLS
  info&=AllocMem&(252&,65538&)
  IF info&=0 THEN
    ALLOCERR
    GOTO Allocation.1
  END IF
Source:
  REQUEST "SOURCE"
  SELECT box%
  IF box% THEN CALL FreeMem(info&,252):SYSTEM
  CHDIR "df0:"
GetFilename:
  LINPUT filename$
  GETINFO filename$,info&,Length&
  IF Length&<1 THEN
    IF Length&=-1 THEN
      DIRERR
    ELSEIF Length&=0 THEN
      FILEERR
    END IF
    GOTO GetFilename
  END IF
Allocation.2:
  COLOR 3,1:CLS
```

```

buffer&=AllocMem&(Length&,65537&)&
IF buffer&=0 THEN&
  ALLOCERR&
  GOTO Allocation.2&
END IF&
LOADFILE filename$,buffer&,Length&&
IF filename$="" THEN&
  CALL FreeMem(buffer&,Length&)&
  LOADERR&
  GOTO GetFilename&
END IF&
IF PEEK(buffer&)<>&HF5 THEN&
  CALL FreeMem(buffer&,Length&)&
  FORMERR&
  GOTO GetFilename&
END IF&
NEWFILE filename$&
Bytes&=1&
DWRITE buffer&,Bytes&&
pointer&=buffer&+1&
GetLength:&
Bytes&=PEEK(pointer&+1)&
IF Bytes&=4 THEN&
  pointer&=pointer&+4&
  GOTO GetLength&
ELSEIF Bytes&>4 THEN&
  IF PEEK(pointer&)=128 THEN offs&=6 ELSE offs&=4&
  IF PEEK(pointer&+offs&)<>175 THEN&
    DWRITE pointer&,Bytes&&
  END IF &
  pointer&=pointer&+Bytes&&
  GOTO GetLength&
ELSE&
  IF ((pointer&-buffer&+1)MOD 2)=1 THEN&
    pointer&=pointer&-1&
  END IF&
  Bytes&=Length&-(pointer&-buffer&+1)+1&
  DWRITE pointer&,Bytes&&
END IF&
CLOSE 1&
OPEN filename$+"-RL.info" FOR OUTPUT AS 1&
OPEN filename$+".info" FOR INPUT AS 2&
PRINT#1,INPUT$(LOF(2),2);&
CLOSE 2,1&
KILL filename$+"-RL.info.info"&
&
CALL FreeMem(buffer&,Length&)&
CALL FreeMem(info&,252&)&
LIBRARY CLOSE&
COLOR 3,1:CLS:LOCATE 2,2:PRINT "Ready."&
WHILE INKEY$="":WEND&
SYSTEM&

```

```

SUB WRITEERR STATIC¶
  COLOR 1,3:CLS:LOCATE 2,2:PRINT "ERROR: Write-error."¶
  ShowCont¶
END SUB ¶
SUB DWRITE(adr&,Length&) STATIC¶
  FOR i&=1 TO Length&¶
    PRINT#1,CHR$(PEEK(adr&-1+i&));¶
  NEXT¶
END SUB¶
SUB OPENERR STATIC¶
  COLOR 1,3:CLS:LOCATE 2,2:PRINT "ERROR: Can't open
File."¶
  ShowCont¶
END SUB ¶
SUB NEWFILE(filename$) STATIC¶
  File$=filename$+"-RL"¶
  OPEN File$ FOR OUTPUT AS 1 ¶
END SUB ¶
SUB FORMERR STATIC¶
  COLOR 1,3:CLS:LOCATE 2,2:PRINT "ERROR: Not a binary
File."¶
  ShowCont¶
END SUB ¶
SUB LOADERR STATIC¶
  COLOR 1,3:CLS:LOCATE 2,2:PRINT "ERROR: Load-error."¶
  ShowCont¶
END SUB¶
SUB LOADFILE(filename$,buffer&,Length&) STATIC¶
  File$=filename$+CHR$(0)
:handle&=xOpen&(SADD(File$),1005)¶
  IF handle&=0 THEN¶
    filename$=""¶
  ELSE ¶
    inBuffer&=xRead&(handle&,buffer&,Length&)¶
    CALL xClose(handle&)¶
    IF inBuffer&<>Length& THEN filename$=""¶
  END IF¶
END SUB¶
SUB FILEERR STATIC¶
  COLOR 1,3:CLS:LOCATE 2,2:PRINT "ERROR: File not
found."¶
  ShowCont¶
END SUB ¶
SUB DIRERR STATIC¶
  COLOR 1,3:CLS:LOCATE 2,2¶
  PRINT "ERROR: File is a Directory."¶
  ShowCont¶
END SUB¶
SUB GETINFO(filename$,info&,Length&) STATIC¶
  File$=filename$+CHR$(0) :DosLock&=Lock&(SADD(File$),-
2)¶
  IF DosLock&=0 THEN ¶
    Length&=0¶
  ELSE¶
    Dummy&=Examine&(DosLock&,info&)¶
    IF PEEKL(info&+4)>0 THEN ¶

```

```

        Length&=-1 ¶
    ELSE ¶
        Length&=PEEKL(info&+124) ¶
    END IF¶
END IF¶
CALL UnLock(DosLock&) ¶
END SUB¶
SUB LINPUT(filename$) STATIC¶
    COLOR 3,1:CLS:WINDOW 2,"Filename:",(0,0)-(250,10),0¶
    WINDOW OUTPUT 1:LOCATE 5,2¶
    PRINT "Name of a binary saved File";¶
    LINE INPUT filename$:WINDOW CLOSE 2¶
END SUB¶
SUB SELECT(box%) STATIC¶
Check: ¶
    WHILE MOUSE(0)=0:WEND:x=MOUSE(1):y=MOUSE(2) ¶
    IF y>27 AND y<43 THEN¶
        IF x>9 AND x<38 THEN box%=0:EXIT SUB¶
        IF x>177 AND x<238 THEN box%=-1:EXIT SUB¶
    END IF¶
    GOTO Check¶
END SUB¶
SUB ALLOCERR STATIC¶
    COLOR 1,3:CLS:LOCATE 2,2:PRINT "ERROR: Allocation
denied."¶
    ShowCont¶
END SUB¶
SUB ShowCont STATIC¶
    LOCATE 4,2:PRINT "Press SPACE to continue,"¶
    LOCATE 5,7:PRINT "ESCAPE to exit.";¶
    WHILE a$<>CHR$(32) AND a$<>CHR$(27) ¶
        a$=INKEY$¶
    WEND¶
    IF a$=CHR$(27) THEN SYSTEM¶
END SUB¶
SUB REQUEST(disk$) STATIC¶
    COLOR 3,1:CLS¶
    LOCATE 2,2:PRINT "INSERT ";disk$;" DISK INTO DRIVE"¶
    LOCATE 3,14:PRINT "DF0:";LOCATE 5,3:PRINT "OK";¶
    LOCATE 5,24:PRINT "CANCEL";:LINE(10,28)-(37,42),3,b¶
    LINE(178,28)-(237,42),3,b¶
END SUB¶

```

**Variables**

ALLOCERR	SUB routine; memory reservation error
AllocMem	EXEC routine; reserves memory
Bytes	length of file being edited
DIERR	SUB routine; error—no file
DWRITE	SUB routine; write to file
DosLock	file handle of Lock
Dummy	unused variable
Examine	DOS routine; looks for file
FILEERR	SUB routine; error
FORMERR	SUB routine; error
File	filename with concluding 0 for DOS



FreeMem	EXEC routine; frees memory range
GETINFO	SUB routine; file check
LINPUT	SUB routine; input
LOADERR	SUB routine; error
LOAD LE	SUB routine; load program
Length	file length
Lock	DOS routine; blocks access from other programs and provides handle
NEWFILE	SUB routine; create new file
OPENERR	SUB routine; error
REQUEST	SUB routine; draw primitive requester
SELECT	SUB routine; select through mouse click
ShowCont	SUB routine; show options
UnLock	DOS routine; releases Lock
WRITEERR	SUB routine; error
a	help variable
adr	address
b	help variable
box	help variable
buffer	address of reserved memory
disk	diskette
filename	name of file
handle	address of file handle
i	help variable
inBuffer	bytes read
info	address of file info structure
offs	offset
pointer	help variable
written	bytes written
x	help variable
xClose	DOS routine; closes file
xOpen	DOS routine; opens file
xRead	DOS routine; reads file
y	help variable

### 5.3.5 Listing variables

You may look at a listing for an older BASIC program and wonder how you can solve any of its problems. Part of human nature lies in doing no more work than necessary. You want to avoid detailed documentation, and at the same time, keep from being buried in a stack of program printouts.

Thanks to modular programming, you can store a collection of short routines on diskette, and merge them into programs as needed. Documenting these short routines is indispensable. Also, many

magazines from which you get program listings usually supply detailed documentation.

The program here gives variable lists and label names. These items are vital to documenting program code. For example, you could check out the variable lists of two files before MERGEing one to the other. This avoids any major rewrites on both programs for changing variables to match/conflict. Keep in mind that the variable list program can view SUB programs and operating system routines as variables, even if the variable types are different. This can occur in other aspects of BASIC with DEFINT xxx (e.g., DEFINT a-c). For example, if you use a variable named Ant on\$, this variable appears in the list under Ant on. If you want the program to ignore uppercase and lowercase during sorting, remove the four UCASE\$( ) statements after the display label.

**Note:**

The loading and saving conventions used in the two preceding programs apply to this section as well.

```
' #####
' # Variable - List Amiga #
' #-----#
' # (W) 1987 by Stefan Maelger #
' #####
'
' "dos.bmap" and "exec.bmap" must be on
' Disk of in LIB:
'-----
'
'
CLEAR,50000&
DECLARE FUNCTION AllocMem& LIBRARY
DECLARE FUNCTION Lock& LIBRARY
DECLARE FUNCTION Examine& LIBRARY
DECLARE FUNCTION xOpen& LIBRARY
DECLARE FUNCTION xRead& LIBRARY
LIBRARY "T&T2:bmaps/exec.library"
LIBRARY "T&T2:bmaps/dos.library"
WINDOW CLOSE WINDOW(0)
DIM varname$(2000),var%(2000),er$(5)
FOR i=0 TO 5:READ er$(i):NEXT
'
DATA "File contains no binary."
DATA "Read-Error.,""File open error."
DATA "File is a directory.,""File not found."
DATA "Allocation denied."
'
nextTry:
REQUEST "Place Disk into Drive df0.",1,"OK","",flag%
WINPUT filename$
CHECKFILE filename$,buffer&
IF buffer<<0 THEN
e%=6+buffer&
REQUEST er$(e%),2,"CANCEL","QUIT",flag%
IF flag%=2 THEN LIBRARY CLOSE:SYSTEM
```

```

        GOTO nextTry¶
    END IF¶
    pointer&=buffer&+1¶
¶
ReadLine:¶
    SETPOINTER pointer&,flag¶
    IF flag%=1 GOTO ReadNames¶
¶
ReadToken:¶
    CHECKTOKEN pointer&,number%¶
    IF number%<0 GOTO ReadLine¶
    var%(number%)=1:GOTO ReadToken¶
¶
ReadNames:¶
    current%=0¶
¶
searching:¶
    IF PEEK(pointer&)=0 OR PEEK(pointer&)>=H60 THEN¶
        pointer&=pointer&+1:GOTO searching¶
    END IF¶
¶
getlength:¶
    length%=PEEK(pointer&)¶
    IF length%=0 GOTO display¶
    FOR i%=1 TO length%¶
        pointer&=pointer&+1¶
¶
varname$(current%)=varname$(current%)+CHR$(PEEK(pointer&
) )¶
    NEXT¶
    current%=current%+1¶
    pointer&=pointer&+1:GOTO getlength¶
¶
display:¶
    flag%=1:first%=0:last%=current%-2¶
    WHILE flag%=1¶
        flag%=0¶
        FOR i%=first% TO last%¶
            IF UCASE$(varname$(i%))>UCASE$(varname$(i%+1))
THEN¶
                SWAP varname$(i%),varname$(i%+1)¶
                SWAP var%(i%),var%(i%+1)¶
                flag%=1¶
            END IF¶
        NEXT¶
        start%=start%+1:flag%=0¶
        FOR i%=last% TO first% STEP -1¶
            IF UCASE$(varname$(i%))<UCASE$(varname$(i%-1))
THEN¶
                SWAP varname$(i%),varname$(i%-1)¶
                SWAP var%(i%),var%(i%-1)¶
                flag%=1¶
            END IF¶
        NEXT¶
        last%=last%-1¶
    WEND¶

```

```

┌
Display2: ┌
  BEEP┌
  REQUEST "List to Screen?",2,"YES","NO",sflag%┌
  REQUEST "List to Printer?",2,"YES","NO",pflag%┌
  REQUEST "Save as ASCII-File?",2,"YES","NO",fflag%┌
  IF sflag%=2 AND pflag%=2 AND fflag%=2 GOTO Display2┌
  IF sflag%=1 THEN WINDOW 2,"Variables:",(0,0)-
(240,180),31┌
  IF fflag%=1 THEN┌
    OPEN filename$+".V" FOR OUTPUT AS 1┌
    PRINT#1,CHR$(10);"Variable-List:";┌
    PRINT#1,CHR$(10);"-----";CHR$(10);CHR$(10);┌
  END IF┌
  IF pflag%=1 THEN┌
    LPRINT "Variable-List from:"┌
    LPRINT filename$:LPRINT┌
  END IF┌
  FOR i%=0 TO current%-1┌
    IF var%(i%)=1 THEN┌
      IF sflag%=1 THEN PRINT varname$(i%)┌
      IF pflag%=1 THEN LPRINT varname$(i%)┌
      IF fflag%=1 THEN PRINT#1,varname$(i%);CHR$(10);┌
    END IF┌
  NEXT┌
  IF fflag%=1 THEN CLOSE 1┌
  REQUEST "Ready.",1,"OK","",flag%┌
  LIBRARY CLOSE┌
  SYSTEM┌
  ┌
SUB CHECKTOKEN(a&,n%) STATIC┌
PeekToken:┌
  t%=PEEK(a&):a&=a&+1┌
  IF t%=0 THEN strflag%=0:n%=-1:EXIT SUB┌
  IF strflag%=1 AND t%<>34 GOTO PeekToken┌
  IF t%>127 THEN┌
    IF t%>247 THEN a&=a&+1┌
    GOTO PeekToken┌
  ELSEIF t%=1 THEN┌
    n%=CVI(CHR$(PEEK(a&))+CHR$(PEEK(a&+1))):a&=a&+2:EXIT
SUB┌
  ELSEIF t%=2 OR t%=11 OR t%=12 OR t%=28 THEN┌
    a&=a&+2:GOTO PeekToken┌
  ELSEIF t%=15 THEN┌
    a&=a&+1:GOTO PeekToken┌
  ELSEIF t%=29 OR t%=30 THEN┌
    a&=a&+4:GOTO PeekToken┌
  ELSEIF t%=31 THEN┌
    a&=a&+8:GOTO PeekToken┌
  ELSEIF t%=3 OR t%=14 THEN┌
    a&=a&+3:GOTO PeekToken┌
  ELSEIF t%=34 THEN┌
    IF strflag%=1 THEN strflag%=0 ELSE strflag%=1┌
    GOTO PeekToken┌
  ELSE┌
    GOTO PeekToken┌

```

```

    END IF¶
END SUB¶
¶
SUB SETPOINTER(a$,f%) STATIC¶
    IF PEEK(a&+1)=0 THEN f%=1 ELSE f%=0¶
    IF PEEK(a&)=0 THEN a&=a&+3 ELSE a&=a&+5¶
END SUB¶
¶
SUB CHECKFILE(a$,f%) STATIC¶
    i&=AllocMem&(252&,65538&)¶
    IF i&=0 THEN ¶
        f&=-1:EXIT SUB¶
    ELSE¶
        b$=a$+CHR$(0):l&=Lock&(SADD(b$),-2)¶
        IF l&=0 THEN¶
            f&=-2:EXIT SUB¶
        ELSE¶
            s&=Examine&(l&,i&)¶
            IF PEEKL(i&+4)>0 THEN¶
                f&=-3:CALL UnLock(l&):EXIT SUB¶
            ELSE¶
                f&=PEEKL(i&+124):CALL UnLock(l&)¶
                CALL FreeMem(i&,252&):v&=f&+3¶
                c&=AllocMem&(v&,65537&)¶
                IF c&=0 THEN¶
                    f&=-1:EXIT SUB¶
                ELSE¶
                    h&=xOpen&(SADD(b$),1005)¶
                    IF h&=0 THEN¶
                        f&=-4:EXIT SUB¶
                    ELSE¶
                        r&=xRead&(h&,c&,f&):CALL xClose(h&)¶
                        IF r&<>f& THEN¶
                            f&=-5:EXIT SUB¶
                        ELSE¶
                            f&=c&¶
                            IF PEEK(f&)<>&HF5 THEN f&=-6:EXIT SUB¶
                        END IF¶
                    END IF ¶
                END IF¶
            END IF¶
        END IF¶
    END IF¶
END SUB¶
¶
SUB WINPUT(a$) STATIC¶
    WINDOW 1,"Input: Filename",(0,0)-(240,8),0¶
    LINE INPUT a$¶
    WINDOW CLOSE 1¶
END SUB¶
¶
SUB REQUEST(a$,m%,b$,c$,b%) STATIC¶
    WINDOW 1,"System Request",(0,0)-(240,40),22¶
    COLOR 0,1:CLS:LOCATE 2,(30-LEN(a$))\2:PRINT a$;:COLOR
    1,0¶
    IF m%=1 THEN¶

```

```

1%=LEN(b$)/2:LOCATE 4,15-1%:PRINT " ";b$;" ";
ELSEIF m%=2 THEN
  LOCATE 4,2:PRINT " ";b$;" ";:LOCATE 4,27-LEN(c$)
  PRINT " ";c$;" ";
END IF
mouse1:
WHILE MOUSE(0)<>0:WEND
WHILE MOUSE(0)=0:WEND
x%=(MOUSE(1)+8)\8:y%=(MOUSE(2)+8)\8:b%=0
IF y%=4 THEN
  IF m%=1 THEN
    IF x%>14-1% AND x%<17+1% THEN b%=1
  ELSEIF m%=2 THEN
    IF x%>1 AND x%<LEN(b$)+4 THEN b%=1
    IF x%>26-LEN(c$) AND x%<30 THEN b%=2
  END IF
END IF
IF b%>0 THEN
  WINDOW CLOSE 1
  EXIT SUB
END IF
GOTO mouse1
END SUB

```

This program created many of the variable lists in this book.

### 5.3.6 Removing "extra" variables

Maybe you've wondered why a binary format BASIC program becomes longer, instead of shorter, when you load, shorten and resave it. Or you've noticed when your BASIC program stops with an error, the orange error box surrounds a couple of blank lines. You find that there's garbage in the program that you can only see with the file monitor. Why does the big program you've been working on run slower and slower every time you edit it? And how can you manipulate internal errors in a binary program?

There is a solution to these problems. As you repeatedly save programs from the AmigaBASIC interpreter, the interpreter adds bits of extraneous data to the file (garbage). Like a garbage can, the program can only hold so much of this garbage. This also goes for the entire memory range assigned to the variable table. When you save a program, the interpreter saves it without checking which variables still belong to the program and which don't. The final problem is that important pointers remain uninitialized—especially if these pointers stay unset before saving or reloading a program.

There is, as always, a loophole. When you save a program in ASCII format, what you get in the file is what you see on the screen: Plain text separated by linefeeds (`CHR$(10)`).

- Save your program once with the extension `.A`.
- Quit AmigaBASIC (if you just type `new`, the garbage still stays on the screen, and the pointers stay unchanged).
- Restart AmigaBASIC's interpreter.
- Load the program.
- Save the program with the extension of `.B` (binary format—very important).

Remember the following rules when trying this resaving:

1. This process works best when you save incomplete programs as ASCII files in the first place. Save the program out in binary form when you wish to try running the program and/or debugging it.
2. When a program runs into a problem you may not be able to see, the logical solution is to save the file in ASCII format. Then you might be able to recover the program.
3. The worst thing you can do is saving a program in binary format after a test run that resulted in an error message. This causes the most garbage sent from the interpreter.
4. If your program doesn't run after all, it may be due to a programmer error or memory error, or an error in AmigaBASIC itself.

---

### 5.3.7 Self-modifying programs

There are methods that allow changing program code as a program runs. The two programs listed below can bring this about.

The first method of program modifying is direct access through `POKE`. The principal is simple: You assign a set of characters to a string variable. This may be at any point in the program. It is important that you make no changes to the string itself, such as `A$=A$+CHR$(0)`. You can point the variable pointer direct to the string in your program.

This first example lets you change strings within a program. This routine opens the window named in the string. Selecting the `CHANGE`

item from the menu lets you insert a new window title, after which the new program loads and starts.

```

REM *****
REM *   S e l f   M o d i f y i n g   I   *
REM *-----*
REM *   (W) 1987 by Stefan Maelger, Hamburg *
REM *****
REM *
REM * The new Title String will be changed here:
REM *
Title$="Self Modifying I"
REM *
SCREEN 1,320,200,2,1
WINDOW 2,Title$,,16,1
MENU 1,0,1,"CHANGE"
MENU 1,1,1,"TITLE"
REM *
ON MENU GOSUB checkmenu
MENU ON
REM *
WHILE Maelger=0
SLEEP
WEND
REM *
MENU RESET
WINDOW CLOSE 2
SCREEN CLOSE 1
END
REM *
checkmenu:
IF MENU(1)=1 AND MENU(0)=1 GOTO newtitle
RETURN
REM *
newtitle:
PRINT "Please enter new Title"
PRINT LEN(Title$);"Characters Long."
LINE INPUT newt$
newt$=LEFT$(newt$+SPACE$(LEN(Title$)),LEN(Title$))
REM *
REM * Here is where the String is changed:
REM *
FOR i=1 TO LEN(newt$)
POKE SADD(Title$)+i-1,ASC(MID$(newt$,i,1))
NEXT i
REM *
REM * Start Program again (with the new Title)
REM *
PRINT "Program with new title being saved."
SAVE "Programname"
PRINT "New Program is saved."
PRINT "Re-Load or start this"
PRINT "program over again."
t=TIMER+15:WHILE t>TIMER:WEND
Maelger=1
RETURN

```



You can see how simple it is. Replace "Programname" with your own program name.

This method lets you change commands in a binary format program. However, it also allows changes to files saved in protected format.

Now we come to the second method—the ASCII file method. Here, too, you can completely change a program. The clincher to this method is the ease in changing entire program sections.

Using POKE to change parameters in a binary format program can have serious consequences: It isn't that easy to change commands. The ASCII file route makes this replacement much simpler.

### *How it works*

Here's the principle behind it. First the program section must be found for replacement. User input works with a syntax check to find the area that needs changing. The running program deletes the program lines you want changed (DELETE from-to). The program then saves to diskette as an ASCII file. While the change waits under its own name, the RAM disk supplies the most speed. Now the saved ASCII program opens for appending (OPEN x\$ FOR APPEND AS y), and the DATA generator creates the new program segment.

In order to get this program into memory, all you need to enter is RUN filename\$ or LOAD filename\$. The program starts all over again, so that you can create the new program section as an ASCII file in the RAM disk, then join the programs with CHAIN MERGE. You can also restart the altered program with a starting label, and merge a series of program segments (e.g., CHAIN stuff,lines,ALL).

```

REM *****
REM *      S e l f M o d i f y i n g   I I      *
REM *-----*
REM * (W) 1987 by Stefan Maelger, Hamburg *
REM *****
REM *
REM * Get the Screens Resolution
REM *
GOSUB VariableLabel
REM *
SCREEN 1, SWidth%, Height%, Depth%, Mode%
WINDOW 2, "Hello!", , 0, 1
REM *
PRINT "Width in Pixels: "; SWidth%
PRINT "Height in Pixels: "; Height%
PRINT "Depth in Planes: "; Depth%
REM *
PRINT
PRINT "Please enter the"
PRINT "New Width: ";
INPUT NewWidth%
IF NewWidth% < 20 OR NewWidth% > 640 THEN
  NewWidth% = SWidth%

```

```

END IF¶
INPUT "New Height:";NewHeight¶
IF NewHeight%<10 OR NewHeight%>512 THEN¶
  NewHeight%=Height%¶
END IF¶
INPUT "New Depth:";NewDepth¶
IF NewDepth%<1 OR NewDepth%>5 THEN¶
  NewDepth%=Depth%¶
END IF¶
PRINT¶
Mode%=1¶
IF NewWidth%>320 THEN Mode%=2¶
IF NewHeight%>256 THEN Mode%=Mode%+2¶
IF Mode%=4 AND NewDepth%>2 THEN¶
  NewDepth%=2¶
ELSEIF Mode%>1 AND NewDepth%>4 THEN¶
  NewDepth%=4¶
END IF¶
OPEN "Programname.t" FOR OUTPUT AS 1¶
PRINT#1,"VariableLabel:";CHR$(10);¶
PRINT#1,"SWidth=";STR$(NewWidth%);CHR$(10);¶
PRINT#1,"Height=";STR$(NewHeight%);CHR$(10);¶
PRINT#1,"Depth=";STR$(NewDepth%);CHR$(10);¶
PRINT#1,"Mode=";STR$(Mode%);CHR$(10);¶
PRINT#1,"RETURN";CHR$(10);¶
PRINT#1,"VariableLabelEnd:";CHR$(10);¶
CLOSE 1¶
¶
DELETE VariableLabel-VariableLabelEnd¶
SAVE "Programname",A¶
OPEN "Programname.t" FOR INPUT AS 1¶
OPEN "Programname" FOR APPEND AS 2¶
PRINT#2,INPUT$(LOF(1),1);¶
CLOSE 2¶
CLOSE 1¶
KILL "Programname.t"¶
WINDOW CLOSE 2¶
SCREEN CLOSE 1¶
LOAD "Programname",R¶
END¶
¶
VariableLabel:¶
SWidth= 320¶
Height= 200¶
Depth= 2¶
Mode= 1¶
RETURN¶
VariableLabelEnd:¶

```

This procedure is particularly good for any kind of graphic program. For example, you could enter user-defined functions in a function plot, palette values in a drawing program, etc.

**6**

**The Workbench**



---

## 6. The Workbench

The Amiga's user interface leaves nothing to the imagination. All important operations can be easily performed using icons. These icons make text input almost unnecessary, thus removing the barriers so often caused by language. Workbench 2.0 is a major improvement over the previous Workbench versions.

### *Workbench Versions*

There are different versions of the Amiga Workbench. These versions are designated by a version number. The two most current versions of the Workbench are Workbench 1.3 and Workbench 2.0. Check the label of your Workbench diskette to make sure you are using one of these versions.

Since Workbench 2.0 is an upgrade to Version 1.3, it greatly enhances the performance of the Amiga. Unfortunately this upgrade requires certain hardware and memory configurations to run in an Amiga. Some Amiga 500 and 2000 models may require hardware modifications in order to run Workbench 2.0. These modifications include installing an enhanced and improved custom chip set or additional memory expansion. See your dealer for information on these upgrades for early Amiga models.

Don't despair if your Amiga will not run Workbench 2.0. Workbench 1.3 is an excellent version of the Amiga operating system and all of your work can be performed using this version of the operating system.

This book will describe Workbench Version 1.3 and 2.0. We'll point out the differences between the two versions so Amiga users of either version will find this book useful.

There are some Workbench functions that few users even know about. These users can form easy solutions to tough problems. This chapter shows how effectively these functions can be used, with a minimum of time and effort.

---

## 6.1 Using the Workbench

The Workbench is the one part of the Amiga that the user sees most often. With that in mind, here are some helpful hints for making your Workbench maintenance and use easier and more efficient.

---

### 6.1.1 Keyboard tricks

Do you know what a *string gadget* is? It's essentially a miniature input window. The Amiga uses string gadgets whenever it needs some form of keyboard input (e.g., for renaming a diskette). Instead of pressing the **Del** key to delete the old name, press and hold the right <Amiga> key and press the **X** key. Presto, the string gadget clears.

In most cases, right <Amiga> **Q** acts as an Undo function, restoring the last item changed.

When you want to move the cursor to the first character of the input line, press **Shift** **+**. Press **Shift** and **→** to move to the end of the input line.

Now we come to the icons. Suppose you want to select more than one icon. Hold down the **⇧** key and click on every icon you want selected. Whatever option is performed with the last icon that is selected will apply to all the selected icons. For example, if you want to throw the multiselected icons into the Trashcan, just drag the last icon to the Trashcan (you can release the **⇧** key).

When Shell output flashes by on the screen (e.g., directory listings), you can stop the listing by pressing the **←** key. Continue the listing by pressing the **↵** (Backspace) key.

If you want to go to the beginning of a screen, or just open a fresh window, press **Ctrl** **L** to clear the screen.

Now and again a prompt may not appear. **Ctrl** **O** and **←** returns the prompt to the screen.

**Ctrl** **D** interrupts the Startup-sequence and **Ctrl** **C** interrupts any currently executing command.

### 6.1.2 The Trashcan

Not everything you make when computing is worthwhile. The developers of the operating system created the Trashcan for disposing of garbage. It's easy to use:

- Select the icon you want to get rid of.
- Drag it to the Trashcan.
- Click on the Trashcan icon.
- Select the Empty Trash item from the Icons (Disk in 1.3) pulldown menu.

There's an even simpler way to do it. The above process works well, on the condition that you remember to empty the trash. However, if you don't the diskette keeps the data placed in the Trashcan in disk memory. Since diskettes only have a capacity of about 880K, this can take up a great deal of disk memory.


Now for the simpler method:

- Click once on the file icon you want disposed of.
  - Select the Delete (Discard in 1.3) item from the Icons (Workbench in 1.3) pulldown menu.
  - Click on the OK (ok to discard in 1.3) gadget in the system requester.
- 



### 6.1.3 Extended selection

Have you ever wondered about how to organize icons in every window? If you put your Extras diskette in the drive and open the BASICdemos drawer, you'll see 25 icons. Most of these icons have such long names that the Clean Up item doesn't put most of them in neat order.

You could conceivably select and move each icon, then execute the Snapshot item from the Special pulldown menu each time you get an icon into position. This takes time, though.

There's a simpler way out. Every icon you click stays active while you hold down one of the  keys. Most of the functions you can perform

on single icons work with multiple icons (assuming that these functions match the icons). For example, you can't use `Delete` (`Discard` in 1.3) on a disk icon.

- Move each icon into the desired position.
- Press and hold the  key.
- Click on all the icons you want organized.
- Release the  key.
- Select the `Snapshot` item from the `Icons` (`Special` in 1.3) pulldown menu.

If you wish to copy several programs, this *extended selection* helps you to do this copying quickly and easily. You can drag a set of icons across the screen, and onto the windows in other diskettes. The only disadvantage is that diskette exchanges must be made for every program.

If you wish to avoid this constant diskette switching, here's a quick method of getting around this:

- Select `New Drawer` from the `Window` pulldown menu or copy the `Empty` drawer of the `Workbench 1.3` diskette onto the formatted source diskette.
- Move all icons you want copied into this drawer using extended selection.
- Drag the drawer to the target diskette icon.



## 6.2 Information

The `Information` (`Info` in 1.3) item from the `Icons` (`Workbench` in 1.3) pulldown menu allows the user to look at information in programs and data files. But which information can you change? These are questions that the Amiga manual doesn't discuss. Here are some answers.

---

### 6.2.1 The Information screen

This screen appears after selecting any kind of icon and selecting the `Information` (`Info` in 1.3) item from the `Icons` (`Workbench` in 1.3) pulldown menu. The `Information` screen lists all the vital information about the file.

The `Information` screen has several areas. The name and type of the file are displayed. The upper left corner lists common data about the file and/or diskette. This information includes the size in two different measurements, the number of disk blocks and the number of bytes the file uses in memory is listed.

`Type` describes the type of icon for a file or diskette. The normal icon types are `Disk`, `Drawer`, `Tool`, `Project` and `Garbage`:

***Disk*** Disks are the diskette icons which lie outside of directory windows. Double-clicking on a disk opens the disk window (the diskette's main directory).

The only item of interest about this icon type is that, like the other icons, you can change its shape. Computer owners who are into nostalgia can change the disk icons to look like 5-1/4" diskettes.

***Drawers*** Drawers are the icons which represent subdirectories. Moving programs into a drawer easily lets you find programs on the same diskette. This operation takes a lot of time, though. Here's a suggestion: Use the `RENAME` command from AmigaDOS. First, enter the first name with the full path specification. Then enter the new name with the path under which you want the program placed (don't forget to copy the `.info` file to the new path as well).

***Tools*** Any executable program is called a tool. Tools can lie in drawers, windows and on the `Workbench` screen. They have their own icons which execute programs when you double-click on them.

AmigaBASIC, DiskCopy and BeckerText Amiga from Abacus are tools.

**Projects**

Amiga projects are any files that contain data saved from a tool (program).

Notepad texts, word processing files, BASIC programs and .bmap files are projects.

**Trashcan**

This last type is actually another form of drawer. Normally you can place drawers inside of drawers. The Trashcan drawer can only lie in the main directory. It also can't be moved onto the Workbench screen. Whenever you need a Trashcan icon, look in the main directory.

On the right side of the screen you'll see a box which lists the Status of the file. This refers to the access options offered to the user (see the AmigaDOS manual under PROTECT). When the write protect is set on the diskette, you can't change the read, write or executable attributes. When you get information from a diskette, this area lists whether the diskette is write-protected or write-enabled.

Notes about the file appear in the Comment line. Amiga-DOS's FILENOTE lets you write a text of up to 80 characters long. What you put under Comment has no effect on other parts of the system—it's just commentary. This function is suppressed by diskettes, since a diskette cannot be supplied with a comment.

The Information screen does more than give information about programs or diskettes. They also supply details about projects (text and data files). Default Tool tells the user which tool created the project, or which diskette has the copy. The Workbench knows which project to load when you double-click a tool's icon.

The last line displays Tool Types. This information is used by the main program that created the file. It stores important information that is passed to the program when you double click on the file.

# **7**

# **Icons**



---

## 7. Icons

The Amiga's Workbench user interface uses *icons* to help the user easily identify programs, data file, directories and diskettes. These icons appear as pictures that quickly indicate their purposes to the user. You start programs by double-clicking on their icons, instead of typing in the program name as you would from the Shell.

Clicking icons saves the trouble of typing in disk paths to open directories and subdirectories to the file you want. All you have to do is click on a drawer; click on the drawer inside the drawer that opens; and so on, until you get to the file icon you need.

This chapter gives detailed information on icon design, drawer structure and image structure. Programs are included that let you edit icons and examine the structure of an icon from AmigaBASIC. You'll also find information about icon structure and creating multiple graphics for one icon (before double-clicking and after double-clicking).

## 7.1 Icon types

There's a problem with this title: All icon symbols can stand for different objects. You have to be able to differentiate between directories and diskettes, and between programs. So, you wouldn't want to assign a drawer icon to the Trashcan any more than you should assign a program icon to a directory. The programs still run, but using "other" icons can cause some confusion later on.

For this reason, this section uses certain icon descriptions in certain contexts. For example, the book consistently calls the icon for a diskette a disk icon, etc.

The following icon types exist:

Name	Identifier	Object	Number
Diskette icon	WBDISK	standard diskette	1
Drawer icon	WBDRAWER	directory	2
Tool icon	WBTOOL	executable program	3
Project icon	WBPROJECT	program data file	4
Trashcan icon	WBGARBAGE	Trashcan	5
Kickstart icon	WBKICK	Kickstart diskette (Amiga 1000)	5

You can get additional information on the icon types from the Workbench. Check the following sources:

Disk icon information corresponds to drawer icons. The drawer icon stores the pictures of all icons and data which can be opened by double-clicking.

Projects (files) are of the same general design as the tools (programs) used to create them. Double-clicking a project icon opens the tools used to create that file, then the project itself.

The Trashcan is really just another form of drawer. The main difference is that you can't move it from one directory to another, nor can you move it to the Workbench.

---

## 7.2 Icon design

Now for the structure, so you can start thinking about designing your own icons. Icon data goes into a directory. Every file that has an icon has an extra file with the same name and a file extension of `.info`. This info file contains the information that goes into the Workbench.

---

### 7.2.1 DiskObject structure

Every icon file begins with a `DiskObject` structure, which contains all sorts of information (see the table below):

Identifier	Parameter	Bytes
<code>do_Magic</code>	magic number	2
<code>do_Version</code>	version number	2
<code>do_Gadget</code>	Click structure	4
<code>gg_LeftEdge</code>	left click range	2
<code>gg_TopEdge</code>	top click range	2
<code>gg_Width</code>	width of click range	2
<code>gg_Height</code>	height of click range	2
<code>gg_Flags</code>	invert flag	2
<code>gg_Activation</code>	\$0003	2
<code>gg_Type</code>	\$0001	2
<code>gg_GadgetRender</code>	pointer1 picture data	4
<code>gg_SelectRender</code>	pointer2 picture data	4
<code>gg_IntuiText</code>	"not used?"	4
<code>gg_MutualExclude</code>	"not useable!"	4
<code>gg_SpecialInfo</code>	"not useable!"	4
<code>gg_GadgetID</code>	"for own use!"	2
<code>gg_UserData</code>	"your Pointer!"	4
<code>do_Type</code>	icon type	1
<code>nothing</code>	fillbyte	1
<code>do_DefaultTool</code>	text structure	4
<code>do_ToolTypes</code>	text structure	4
<code>do_CurrentX</code>	current X-position	4
<code>do_CurrentY</code>	current Y-position	4
<code>do_DrawerData</code>	window structure	4
<code>do_ToolWindow</code>	program window	4
<code>do_StackSize</code>	reserved memory	4

For starters, the magic number is equal to `$E310`. This tells the system that this is where an icon is read. Next follows the version number,

which at the time of this writing is always \$0001. The above table indicates how many bytes each value occupies.

Four unused bytes follow the structure. These are normally reserved for a gadget click structure. Now things get more complicated: The symbol itself is actually divided into two separate areas—the graphic range and the click range. The click range helps determine the range in which you can click on the icon. The X- and Y-offsets of the click position follow, setting the upper left corner of the click range. Next comes the width and height of that range. It's important to remember that text is printed beneath the click range (i.e., under the icon). Be sure that the click range is high enough that the text can be counted as part of the graphic.

### *Gadgets*

Now comes the gadget structure. The next value changes the picture when you activate it. You have three options at your disposal:

1. The entire rectangular area in which the icon is displayed inverts. Just place a 4 in the `Flags` register. This is the simplest (but not the most attractive) method.
2. Only the drawn-in area inverts. This looks and works somewhat better than #1. This mode requires a 5 in the `Flags` register.
3. Instead of an inverse version of the icon, another icon appears altogether. Place a 6 in the `Flags` register.

Next the value constants \$0003 and \$0001 follow in the `DiskObject` structure. The first is the activation type, and the second marks a Book gadget. The pointers to icon graphic data follow. If you're switching between two graphics, the second pointer must be initialized.

The next 18 bytes are required by the system for normal gadgets. Its actual purpose appears to make no sense. It works best when you fill this area with zeros. These bytes are important to the next parameter: It distinguishes which icon type is available to the user. You insert the numbers which indicate the table. Since this should be given in one byte, and the processor can only address even addresses, these are the same as fillbytes.

### *Tool types*

In order to select the type, the pointer to the `Default Tool` structure then the pointer to the `ToolTypes` structure must be set (more on these pointers later).

The system stores the positioning in the `DiskObject` structure as the current X- and Y-coordinates. However, you also have the option of Workbench coordinates of \$80000000, \$80000000. These values are called `NO_ICON_POSITION`. As long as a user-created icon stays unchanged, it is found at the same position. A pointer to the window data follows if necessary, and a pointer to the `ToolWindow` structure.



To conclude, the stack depth tells the Workbench how much memory to allocate for this program or this data. The value of a data file has higher priority than a main program. This way you could reserve considerably more memory for the data records of a file.

## 7.2.2 Drawer structure

Now that you have the information about the average `DiskObject` structure, you can continue on with the individual types.

First comes the `Drawer` structure, which is almost equal to a diskette. The big difference is that the directory and the Trashcan use this structure. It contains all the data needed for opening a new directory window. The table reads as follows:

Identifier	Parameter	Bytes
<code>wi_LeftEdge</code>	left corner	2
<code>wi_TopEdge</code>	top edge	2
<code>wi_Width</code>	width	2
<code>wi_Height</code>	height	2
<code>wi_DetailPen</code>	drawing color 1	1
<code>wi_BlockPen</code>	drawing color 2	1
<code>wi_IDCMPFlags</code>	gadget flags	4
<code>wi_Flags</code>	window flags	4
<code>wi_FirstGadget</code>	gadget structure	4
<code>wi_CheckMark</code>	checkmark	4
<code>wi_Title</code>	title text	4
<code>wi_Screen</code>	screen pointer	4
<code>wi_BitMap</code>	window bitmap	4
<code>wi_MinWidth</code>	minimum width	2
<code>wi_MinHeight</code>	minimum height	2
<code>wi_MaxWidth</code>	maximum width	2
<code>wi_MaxHeight</code>	maximum height	2
<code>wi_Type</code>	\$0001	2
<code>actx-pos</code>	current X-position	2
<code>acty-pos</code>	current Y-position	2

These are handled as an independent window structure, which extends the coordinates for the current position. This may need some explanation:

The upper left corner coordinates and the window size appear. When the user moves and closes the window, the diskette doesn't leave the system, and the directory window isn't opened at the position given by the current coordinates.

The parameters then follow for color control. The values set the colors for the lines and blocks used in a window. Normally \$FF stands for -1, which takes the color from the screens in use. This makes color control much simpler.

### *Handling window changes*

The next byte contains a pointer and flag used by the system internals. First comes the IDCMP flag, which sets the reaction to any changes to a window. The window flag determines the setup of the directory window. Then five pointers to structures or memory ranges follow, whose changes require knowledge of the operating system.

This way all windows set up in any size within the minimum and maximum limits set by `MinHeight`, `MaxWidth` and `MaxHeight`.

## 7.2.3 Image structure

Every icon needs an `Image` structure. They contain the graphic data, and are set into the respective file twice when necessary.

Identifier	Parameter	Bytes
<code>im_LeftEdge</code>	left corner	2
<code>im_TopEdge</code>	top edge	2
<code>im_Width</code>	width	2
<code>im_Height</code>	height	2
<code>im_Depth</code>	depth	2
<code>im_ImageData</code>	bitplane pointer	4
<code>im_PlanePick</code>	graphic data	1
<code>im_PlaneOnOff</code>	use	1
<code>im_NextImage</code>	next graphic	4

After information about the sizes and positions of several bitmaps, the image setup contains the graphic itself. The number of bitmaps depend upon the screen's depth. The Workbench has a normal depth of two bitmaps on which the icon is also based.

The image parameters repeat after the icon position is given to the `DiskObject` structure. The position is just an offset of this parameter. No values are left out concerning the width, height and number of bitplanes, just as on the other bitplanes.

The next four bytes are a pointer to the current graphic data. This pointer can change the next couple of parameters somewhat. For example, `PlanePick` depends on the number of bitplanes for its graphic display. And `PlaneOff` controls an unused icon's activity.

The last parameter is a pointer to another `Image` structure. This lets you combine several objects into one unit.

The bytes of the individual bitplanes follow the Image structure. First comes bitplane 1, then bitplane 2, and so on (if more bitplanes are used). The system computes the number of bytes needed for the width by rounding off the number of pixels to the next highest multiple of 16. The height is calculated by the number of pixels in height. This number is rounded off to the next highest multiple of 8. The Amiga needs these bytes to create any bitplane.

### 7.2.4 DefaultTool text

Unlike the Image structure, used by every icon, you only need the DefaultTool text for diskettes and data files. Diskettes use the text to state the diskette hierarchy needed to call system programs. For example, every diskette contains the text `SYS: System/DiskCopy`, used to access the disk copy program (if you remove this text the disk cannot be copied in this manner). Data files use this text to indicate the program used to create these files. If you remove these texts, the main program becomes inaccessible. Here's the parameter setup:

Identifier	Parameter	Bytes
<code>char_num</code>	number of characters	4

This list contains only the truly concrete data (the number of characters). Everything else is flexible. Every text must end with a nullbyte, so that the end is identifiable.

### 7.2.5 ToolTypes text

The section on the Info function of the Workbench (Section) mentioned that the string gadget under ToolTypes lets you give additional information about the main program. For example, you could set up a text file for handling as an IFF file. The program requires other information that doesn't appear in this area. You can easily add this information, and use the file in other programs as an interchange format file.

Identifier	Parameter	Bytes
<code>string_num</code>	text number	4

Like the DefaultTool text, the size of the ToolTypes gadget is extremely difficult to change. Assuming that this string isn't blank, the beginning of the text has the number of the string. You must increment the number contained here by one, then multiply by four, to

compute the string number. You can also find this number when you read the file. If you want the data expressed, you must reverse the procedure.

Next follows a string which begins with the length, and ends with a nullbyte. The number of characters is computed by `string_num` mentioned above.

## 7.2.6 Icon analyzer

The following program is a move toward the practical side of icon structure. This BASIC program reads the parameters of the filename, and displays these parameters and their corresponding values. This program would be easier to use if you could print this list to a printer (you may wish to modify it to do so).

```

REM ICONANALYZER
REM V1.3 and V2.0 tested
DIM DiskObject$(26,3),DiskObject(26)¶
DIM DrawerData$(20,3),DrawerData(20)¶
DIM Image$(2,9,3),Image(2,9)¶
DIM DefaultTool$(2,3),DefaultTool(2)¶
¶
¶
DEF FNSize%(Im)=Image(Im,4)*2*INT((Image(Im,3)+15)/16)¶
¶
WIDTH 75¶
¶
INPUT "Filename:";File$¶
¶
OPEN File$+".info" FOR INPUT AS 1¶
  ¶
  summary$=INPUT$(LOF(1),1)¶
  ¶
CLOSE 1¶
¶
summary$=summary$+STRING$(40,0)¶
¶
GOSUB LoadHeader¶
¶
  IF DiskObject(18)=1 THEN¶
    GOSUB LoadDrawer¶
    GOSUB LoadImage¶
    GOSUB LoadDefaultTool¶
    GOSUB LoadToolTypes¶
  END IF¶
¶
  IF DiskObject(18)=2 OR DiskObject(18)=5 THEN¶
    GOSUB LoadDrawer¶
    GOSUB LoadImage¶

```

```

    GOSUB LoadToolTypes¶
END IF¶
¶
IF DiskObject (18)=3 THEN¶
    GOSUB LoadImage ¶
    GOSUB LoadToolTypes¶
END IF¶
¶
IF DiskObject (18)=4 THEN¶
    GOSUB LoadImage¶
    GOSUB LoadDefaultTool¶
    GOSUB LoadToolTypes¶
END IF¶
¶
END¶
¶
¶
LoadHeader:¶
    RESTORE DiskObject¶
    po=1 : PRINT¶
    PRINT "Disk Object Structure" : PRINT¶
    FOR i=1 TO 26 ¶
        GetBytes DiskObject$(i,1),DiskObject$(i,2),
DiskObject$(i,3),DiskObject(i)¶
    NEXT i ¶
RETURN¶
¶
LoadDrawer:¶
    RESTORE DrawerData¶
    PRINT¶
    PRINT "Drawer Data Structure" : PRINT¶
    FOR i=1 TO 20¶
        GetBytes DrawerData$(i,1),DrawerData$(i,2),
DrawerData$(i,3),DrawerData(i)¶
    NEXT i¶
RETURN¶
¶
LoadImage:¶
    Im=1¶
    GOSUB GetImage¶
    IF DiskObject(12)<>0 THEN Im=2 : GOSUB GetImage¶
RETURN¶
¶
GetImage:¶
    RESTORE Image¶
    PRINT¶
    PRINT "Image Structure" : PRINT¶
    FOR i=1 TO 9¶
        GetBytes Image$(Im,i,1),Image$(Im,i,2),
Image$(Im,i,3),Image(Im,i)¶
    NEXT i¶
    bytes=FNSize%(Im)¶
    PRINT¶
    PRINT "BitPlanes" : PRINT¶
    WIDTH 60¶
    FOR j=1 TO Image(Im,5)¶

```

```

PRINT
PRINT "Bitplane";j
FOR i=1 TO bytes
  a$=HEX$(ASC(MID$(summary$,po,1)))
  IF LEN(a$)<2 THEN a$="0"+a$
  PRINT a$;
  IF i/2=INT(i/2) THEN PRINT " ";
  po=po+1
NEXT i
PRINT
NEXT j
WIDTH 75
RETURN
LoadDefaultTool:
  RESTORE DefaultTool
  PRINT
  PRINT "Default Tool" : PRINT
  GetBytes DefaultTool$(1,1),DefaultTool$(1,2),
  DefaultTool$(1,3),DefaultTool(1)
  IF DefaultTool(1)>80 THEN
  DefaultTool(1)=DefaultTool(1)/16
  GetString DefaultTool(1)
  RETURN
LoadToolTypes:
  RESTORE ToolTypes
  PRINT
  PRINT "ToolTypes" : PRINT
  IF po>LEN(summary$) THEN RETURN
  GetBytes ToolTypes$(1,1),ToolTypes$(1,2),
  ToolTypes$(1,3),ToolTypes(1)
  FOR i=1 TO ToolTypes(1)/4-1
    RESTORE DefaultTool
    ToolTypes$(2,3)=""
    GetBytes ToolTypes$(2,1),ToolTypes$(2,2),
    ToolTypes$(2,3),ToolTypes(2)
    IF ToolTypes(2)>80 THEN
    ToolTypes(2)=ToolTypes(2)/16
    GetString ToolTypes(2)
  NEXT i
  RETURN
SUB GetString (length) STATIC
  SHARED po,summary$
  ts=po : a=1
  IF length=0 THEN EXIT SUB
  WHILE a<>0
    a=ASC(MID$(summary$,po,1))
    a$=HEX$(a)
    IF LEN(a$)<2 THEN a$="0"+a$
    PRINT a$;" ";

```

```

        po=po+1¶
    WEND¶
    PRINT¶
    PRINT MID$(summary$,ts,po-ts-1)¶
    ¶
END SUB¶
¶
¶
SUB Decimal (he$,dec) STATIC¶
¶
    dec=0¶
    FOR i=1 TO LEN(he$) ¶
        a=ASC(MID$(he$,LEN(he$)+1-i,1))-48¶
        IF a>9 THEN a=a-7¶
        dec=dec+16^(i-1)*a¶
    NEXT i¶
    ¶
END SUB¶
¶
SUB GetBytes (identifier$,parameter$,value$,dec) STATIC¶
¶
    SHARED po,summary$¶
    READ identifier$,parameter$,bytes¶
    PRINT identifier$;TAB(20);parameter$;TAB(47);¶
    a$=MID$(summary$,po,bytes)¶
    po=po+bytes¶
    IF bytes=1 THEN value=ASC(a$)¶
    IF bytes=2 THEN value=CVI(a$)¶
    IF bytes=4 THEN¶
        FOR j=1 TO 4¶
            a=ASC(MID$(a$,j,1))¶
            h$=HEX$(a)¶
            IF LEN(h$)<2 THEN h$=h$+"0"¶
            value$=value$+h$¶
        NEXT j¶
    ELSE¶
        value$=HEX$(value)¶
    END IF¶
    PRINT "$";value$;TAB(57);¶
    Decimal value$,dec¶
    PRINT dec ¶
    ¶
END SUB¶
¶
¶
¶
DiskObject:¶
¶
DATA do_Magic,Magic Number,2¶
DATA do_Version,Version Number,2¶
DATA do_Gadget,Click Structure,4¶
DATA gg_LeftEdge,Left Click Range,2¶
DATA gg_TopEdge,Top Click Range,2¶
DATA gg_Width,Click Range Width,2¶
DATA gg_Height,Click Range Height,2¶
DATA gg_Flags,Invert Flag,2¶

```

```

DATA gg_Activation,$0003,2¶
DATA gg_Type,$0001,2¶
DATA gg_GadgetRender,Pointer1 Picture Data,4¶
DATA gg_SelectRender,Pointer2 Picture Data,4¶
DATA gg_IntuiText,"not used??",4¶
DATA gg_MutualExclude,"not usable!",4¶
DATA gg_SpecialInfo,"not useable!",4¶
DATA gg_GadgetID,"for own use!",2¶
DATA gg_UserData,"your Pointer!",4¶
DATA do_Type,Icon type,1¶
DATA nothing,Fillbyte,1¶
DATA do_DefaultTool,Text Structure,4¶
DATA do_ToolTypes,Text Structure,4¶
DATA do_CurrentX,Current x-Position,4¶
DATA do_CurrentY,Current y-Position,4¶
DATA do_DrawerData,Window Structure,4¶
DATA do_ToolWindow,Program Window,4¶
DATA do_StackSize,Reserved Memory,4¶
¶
DrawerData:¶
¶
DATA wi_LeftEdge,Left Edge,2¶
DATA wi_TopEdge,Top Edge,2¶
DATA wi_Width,Width,2¶
DATA wi_Height,Height,2¶
DATA wi_DetailPen,Drawing Color 1,1¶
DATA wi_BlockPen,Drawing Color 2,1¶
DATA wi_IDCMPFlags,Gadget Flags,4¶
DATA wi_Flags,Window Flags,4¶
DATA wi_FirstGadget,Gadget Structure,4¶
DATA wi_CheckMark,CheckMark,4¶
DATA wi_Title,Title Text,4¶
DATA wi_Screen,Screen Pointer,4¶
DATA wi_BitMap,Window BitMap,4¶
DATA wi_MinWidth,Minimum Width,2¶
DATA wi_MinHeight,Minimum Height,2¶
DATA wi_MaxWidth,Maximum Width,2¶
DATA wi_MaxHeight,Maximum Height,2¶
DATA wi_Type,$0001,2¶
DATA actx-pos,Current x-Position,4¶
DATA acty-pos,Current y-Position,4¶
¶
Image:¶
¶
DATA im_LeftEdge,Left Edge,2¶
DATA im_TopEdge,Top Edge,2¶
DATA im_Width,Width,2¶
DATA im_Height,Height,2¶
DATA im_Depth,Depth,2¶
DATA im_ImageData,BitPlane Pointer,4¶
DATA im_PlanePick,Graphic Data,1¶
DATA im_PlaneOnOff,Use,1¶
DATA im_NextImage,Next Graphic,4¶
¶
DefaultTool:¶
¶

```



```

DATA char_num, Number of Characters, 4¶
¶
ToolTypes:¶
¶
DATA string_num, Text Number, 4¶

```

### ***Program description***

After creating arrays for all structures, the program prompts for the filename you want analyzed. Do not enter the `.info` file extension, since the program provides that extension automatically. Next, all data contained in the file goes into `summary$`, so that disk access won't be needed later. If the text contains no closing nullbyte (Intuition normally does this), nullbytes are added. The main program jumps to the `DiskObject` structure reading routine.

Once the routine closes, the program branches to examine the icon type. The available structures are viewed, then the program branches to the required routines for looking into each structure.

The most important subroutine of all, `LoadHeader`, analyzes the `DiskObject` structure. This loads the name and the byte lengths of individual parameters from the `DATA` statements. The `DATA` lines are searched for the `GetBytes` subroutine, used by almost every subroutine.

After `GetBytes` reads the text and data lengths, the text goes into the window. From this text, the program computes the corresponding number to be displayed from the bytes. Then a subroutine executes for converting the hexadecimal values to decimal notation so the user can read the text more easily.

The `LoadDrawer` subroutine works in the same way as `LoadHeader`. It reads the starting data, but computes the size of the graphic array from `Size%`; this lets you incorporate this size with your own display routines. Then the routine tests for a possible `Double-Image`. If there is a `Double-Image`, both `Image` structures must be read.

The `LoadDefaultTool` routine reads the text length from `Get-Bytes`. This number is multiplied by 16 for most test-icons, when this is needed. Next follows the call for the `GetString` routine, which reads the corresponding number of the string.

The same goes for `LoadToolTypes`, only the number of the text must be read.

---

## 7.3 Making your own icons

Now that you have some information about the structure of icons, you can now learn how to use and create your own icons. It's much easier to take an established icon and change it to your own needs. You can use the icon editor built into the Workbench diskette for this purpose.

---

### 7.3.1 Two graphics, one icon

This section tells how you can force the Amiga to display a new graphic for an icon that has been clicked, instead of simply inverting the original icon colors. This is a common method that can be applied to any icon type. Later on, you'll learn other extras such as changing drawer icons only.

The change must set the pointer to the second Image structure into which the new data is inserted. This problem is easier to solve than you might think. You must create two icons with a program like the Icon Editor. The only stipulation is that both icons must be the same size. After you enter the name, both icons are combined into one unit.

With this combined icon, you can create wonderful effects. For example, you can make the Trashcan icon "lid" open up when you click on the Trashcan icon (some versions of the Workbench already have this feature). You can also make a drawer icon "open" when you click on it (again, this already happens on some later Workbench diskettes).

---

### 7.3.2 Text in graphics

Another option for enhancing normal icons is placing text above the icon graphic.

As you saw from the DiskObject structure, the graphic range proper is different from the click range. This click range is given in the DiskObject structure at parameters 4-7. The icon's text appears below this click range. If you lower the height of the click range, then you can raise the text proportionately. This means that you can move the text up, and have it somewhere other than underneath the icon.

### 7.3.3 The icon editor

These changes require a program that allows you to access and change certain bytes. Save these altered bytes to diskette.

The program below is an extension of the analyzer program listed earlier. The entire program is listed below. Load your analyzer program, compare the listing with this listing and add the new lines. Save the modified program under the name IconEditor.

```

DIM DiskObject$(26,3),DiskObject(26)¶
DIM DrawerData$(20,3),DrawerData(20)¶
DIM Image$(2,9,3),Image(2,9)¶
DIM DefaultTool$(2,3),DefaultTool(2)¶
DIM Address(100,3)¶
¶
ON TIMER(.5) GOSUB KeyTest¶
TIMER ON¶
¶
DEF FNSize%(Im)=Image(Im,4)*2*INT((Image(Im,3)+15)/16)¶
¶
WIDTH 75 : Adr=1 : AdrNum=1¶
¶
INPUT "Pathname:";Path$¶
INPUT "Filename:";File$¶
¶
OPEN Path$+File$+".info" FOR INPUT AS 1¶
¶
    summary$=INPUT$(LOF(1),1)¶
    ¶
CLOSE 1¶
¶
summary$=summary$+STRING$(40,0)¶
¶
LstBytes:¶
number=0 : lst=0¶
GOSUB LoadHeader¶
¶
IF DiskObject(18)=1 THEN¶
    GOSUB LoadDrawer¶
    GOSUB LoadImage¶
    GOSUB LoadDefaultTool¶
    GOSUB LoadToolTypes¶
END IF¶
¶
IF DiskObject(18)=2 OR DiskObject(18)=5 THEN¶
    GOSUB LoadDrawer¶
    GOSUB LoadImage¶
    GOSUB LoadToolTypes¶
END IF¶
¶

```

```

IF DiskObject (18)=3 THEN¶
  GOSUB LoadImage ¶
  GOSUB LoadToolTypes¶
END IF¶
¶
IF DiskObject (18)=4 THEN¶
  GOSUB LoadImage¶
  GOSUB LoadDefaultTool¶
  GOSUB LoadToolTypes¶
END IF¶
¶
PRINT¶
PRINT "End of File!"¶
¶
WHILE last=0¶
  SLEEP¶
  IF lst=1 THEN GOTO LstBytes¶
WEND¶
¶
END¶

KeyTest:¶
¶
IF INKEY$<>" " THEN RETURN¶
WINDOW 2, "Input", (0,0)-(631,53), 6¶
¶
Start:¶
PRINT "Address:"Adr, Address (AdrNum, 3) ¶
INPUT "Command: ", Command$¶
ComKey$=LEFT$(Command$, 1) ¶
ComTxt$=MID$(Command$, 2) ¶
ComValue#=VAL(ComTxt$) ¶
IF ComKey$="#" THEN¶
  FOR TestI=1 TO number¶
    IF Address (TestI, 1)=ComValue# THEN Adr=ComValue# :
AdrNum=TestI¶
  NEXT TestI¶
  GOTO Start¶
END IF¶
IF ComKey$="e" THEN last=1¶
IF ComKey$="s" THEN¶
  IF LEN(ComTxt$)>0 THEN File$=ComTxt$¶
  OPEN ":mod. Icons/"+File$+".info" FOR OUTPUT AS 1¶
  PRINT#1, summary$¶
  CLOSE 1¶
  KILL ":mod. Icons/"+File$+".info.info"¶
  GOTO Start¶
END IF¶
IF ComKey$="a" THEN¶
  bytes$="" : value#=ComValue#¶
  FOR KeyI=Address (AdrNum, 2)-1 TO 1 STEP -1¶
    a=INT (value#/256^KeyI) ¶
    value#=value#-a*256^KeyI¶
    bytes$=bytes$+CHR$(a) ¶
  NEXT KeyI¶
  bytes$=bytes$+CHR$(value#) ¶

```

```

MID$(summary$,Adr,Address(AdrNum,2))=bytes$¶
Address(AdrNum,3)=ComValue#¶
GOTO Start¶
END IF¶
IF ComKey$="1" THEN lst=1¶
¶
WINDOW CLOSE 2¶
¶
RETURN¶
¶
¶
LoadHeader:¶
RESTORE DiskObject¶
po=1 : PRINT¶
PRINT "Disk Object Structure" : PRINT¶
FOR I=1 TO 26 ¶
    GetBytes DiskObject$(I,1),DiskObject$(I,2),
DiskObject$(I,3),DiskObject(I)¶
NEXT I ¶
RETURN¶
¶
LoadDrawer:¶
RESTORE DrawerData¶
PRINT¶
PRINT "DrawerData Structure" : PRINT¶
FOR I=1 TO 20¶
    GetBytes DrawerData$(I,1),DrawerData$(I,2),
DrawerData$(I,3),DrawerData(I)¶
NEXT I¶
RETURN¶
¶
LoadImage:¶
Im=1¶
GOSUB GetImage¶
IF DiskObject(12)<>0 THEN Im=2 : GOSUB GetImage¶
RETURN¶
¶
GetImage:¶
RESTORE Image¶
PRINT¶
PRINT "Image Structure" : PRINT¶
FOR I=1 TO 9¶
    GetBytes Image$(Im,I,1),Image$(Im,I,2),
Image$(Im,I,3),Image(Im,I)¶
NEXT I¶
bytes=FNSize%(Im)¶
PRINT¶
PRINT "BitPlanes" : PRINT¶
WIDTH 60¶
FOR j=1 TO Image(Im,5)¶
    PRINT¶
    PRINT "Bitplane";j¶
    FOR I=1 TO bytes¶
        a$=HEX$(ASC(MID$(summary$,po,1)))¶
        IF LEN(a$)<2 THEN a$="0"+a$¶
        PRINT a$;¶
    
```

```

        IF I/2=INT(I/2) THEN PRINT " ";¶
        po=po+1¶
    NEXT I¶
    PRINT¶
    NEXT j¶
    WIDTH 75 ¶
    RETURN¶
    ¶
    LoadDefaultTool:¶
    RESTORE DefaultTool¶
    PRINT¶
    PRINT "DefaultTool" : PRINT¶
    GetBytes DefaultTool$(1,1),DefaultTool$(1,2),
    DefaultTool$(1,3),DefaultTool(1)¶
    IF DefaultTool(1)>80 THEN
    DefaultTool(1)=DefaultTool(1)/16¶
    GetString DefaultTool(1)/16¶
    RETURN¶
    ¶
    LoadToolTypes:¶
    RESTORE ToolTypes¶
    PRINT¶
    PRINT "ToolTypes" : PRINT¶
    IF po>LEN(summary$) THEN RETURN¶
    GetBytes ToolTypes$(1,1),ToolTypes$(1,2),
    ToolTypes$(1,3),ToolTypes(1)¶
    FOR I=1 TO ToolTypes(1)/4-1¶
        RESTORE DefaultTool¶
        ToolTypes$(2,3)=""¶
        GetBytes ToolTypes$(2,1),ToolTypes$(2,2),
        ToolTypes$(2,3),ToolTypes(2)¶
        IF ToolTypes(2)>80 THEN
        ToolTypes(2)=ToolTypes(2)/16¶
        GetString ToolTypes(2)¶
    NEXT I ¶
    RETURN¶
    ¶
    ¶
    SUB GetString (length) STATIC¶
    ¶
    SHARED po,summary$¶
    ¶
    ts=po : a=1¶
    IF length=0 THEN EXIT SUB¶
    ¶
    WHILE a<>0¶
        a=ASC(MID$(summary$,po,1))¶
        a$=HEX$(a)¶
        IF LEN(a$)<2 THEN a$="0"+a$¶
        PRINT a$;" ";¶
        po=po+1¶
    WEND¶
    PRINT¶
    PRINT MID$(summary$,ts,po-ts-1)¶
    ¶
    END SUB¶

```

```

¶
¶
SUB Decimal (he$,dec) STATIC¶
¶
  dec=0¶
  FOR I=1 TO LEN(he$) ¶
    a=ASC(MID$(he$,LEN(he$)+1-I,1))-48¶
    IF a>9 THEN a=a-7¶
    dec=dec+16^(I-1)*a¶
  NEXT I¶
  ¶
END SUB¶
¶
SUB GetBytes (identifier$,parameter$,value$,dec) STATIC¶
¶
  SHARED po,summary$,Address(),number¶
  READ identifier$,parameter$,bytes¶
  PRINT identifier$;TAB(20);parameter$;TAB(47);¶
  a$=MID$(summary$,po,bytes)¶
  IF bytes=1 THEN value=ASC(a$)¶
  IF bytes=2 THEN value=CVI(a$)¶
  IF bytes=4 THEN¶
    value$=""¶
    FOR j=1 TO 4¶
      a=ASC(MID$(a$,j,1))¶
      h$=HEX$(a)¶
      IF LEN(h$)<2 THEN h$=h$+"0"¶
      value$=value$+h$¶
    NEXT j¶
  ELSE¶
    value$=HEX$(value)¶
  END IF¶
  PRINT "$";value$;TAB(57);¶
  Decimal value$,dec¶
  PRINT dec;TAB(71);po¶
  number=number+1¶
  Address(number,1)=po : Address(number,2)=bytes :
  Address(number,3)=dec¶
  po=po+bytes¶
  ¶
END SUB¶
¶
¶
DiskObject:¶
¶
DATA do_Magic,Magic Number,2¶
DATA do_Version,Version Number,2¶
DATA do_Gadget,Click Structure,4¶
DATA gg_LeftEdge,Left Click Range,2¶
DATA gg_TopEdge,Top Click Range,2¶
DATA gg_Width,Click Range Width,2¶
DATA gg_Height,Click Range Height,2¶
DATA gg_Flags,Invert Flag,2¶
DATA gg_Activation,$0003,2¶
DATA gg_Type,$0001,2¶

```

```

DATA gg_GadgetRender,Pointer1 Picture Data,4¶
DATA gg_SelectRender,Pointer2 Picture Data,4¶
DATA gg_IntuiText,"not used??",4¶
DATA gg_MutualExclude,"not useable!",4¶
DATA gg_SpecialInfo,"not useable!",4¶
DATA gg_GadgetID,"for own use!",2¶
DATA gg_UserData,"your Pointer!",4¶
DATA do_Type,Icon type,1¶
DATA nothing,Fillbyte,1¶
DATA do_DefaultTool,Text Structure,4¶
DATA do_ToolTypes,Text Structure,4¶
DATA do_CurrentX,Current x-Position,4¶
DATA do_CurrentY,Current y-Position,4¶
DATA do_DrawerData,Window Structure,4¶
DATA do_ToolWindow,Program Window,4¶
DATA do_StackSize,Reserved Memory,4¶
¶
DrawerData:¶
¶
DATA wi_LeftEdge,Left Edge,2¶
DATA wi_TopEdge,Top Edge,2¶
DATA wi_Width,Width,2¶
DATA wi_Height,Height,2¶
DATA wi_DetailPen,Drawing Color 1,1¶
DATA wi_BlockPen,Drawing Color 2,1¶
DATA wi_IDCMPFlags,Gadget Flags,4¶
DATA wi_Flags,Window Flags,4¶
DATA wi_FirstGadget,Gadget Structure,4¶
DATA wi_CheckMark,CheckMark,4¶
DATA wi_Title,Title Text,4¶
DATA wi_Screen,Screen Pointer,4¶
DATA wi_BitMap,Window BitMap,4¶
DATA wi_MinWidth,Minimum Width,2¶
DATA wi_MinHeight,Minimum Height,2¶
DATA wi_MaxWidth,Maximum Width,2¶
DATA wi_MaxHeight,Maximum Height,2¶
DATA wi_Type,$0001,2¶
DATA actx-pos,Current x-Position,4¶
DATA acty-pos,Current y-Position,4¶
¶
Image:¶
¶
DATA im_LeftEdge,Left Edge,2¶
DATA im_TopEdge,Top Edge,2¶
DATA im_Width,Width,2¶
DATA im_Height,Height,2¶
DATA im_Depth,Depth,2¶
DATA im_ImageData,BitPlane Pointer,4¶
DATA im_PlanePick,Graphic Data,1¶
DATA im_PlaneOnOff,Use,1¶
DATA im_NextImage,Next Graphic,4¶
¶
DefaultTool:¶
¶
DATA char_num,Number of Characters,4¶
¶

```



```
ToolTypes:¶
¶
DATA string_num,Text Number,4¶
```

### ***Program description***

Most of this program matches the icon analyzer program in structure and program flow. One change is the byte number following all changeable parameters. In addition, pressing **[Spacebar]** calls a window. This window lets you enter the following simple file management commands:

- # num** Enter the address for num at which you want the change made. From there you can select the position where you want your bytes added.
- a num** The current address is assigned the value placed in num. The routine converts the given number to byte format.
- s name** This saves the info file bytes in the directory `:mod.Icons`. You should make this directory before running this program (use `makedir mod.Icons` in the Shell to create this directory). If a name isn't given after s, the name used for the previous loading procedure is assigned to s.
- l** Once you've made changes, this lets you list the program structure.
- e** This command ends the program. The e command must be given, since the program's display structure is within a delay loop.

When you want to exit the editor, press the **[←]** key at any prompt without entering any other text.

The authors realize that this editor isn't the most comfortable one in the world to work with. However, a more user-friendly editor would take up much more memory, and the current version of the editor performs all the necessary functions.

## **7.3.4 Color changes**

Any window can open in its own color, including the Workbench window. The default Workbench colors are effective enough, but they aren't very interesting. To change these colors, you must change the data and colors before opening the window. Changing window structure is very similar to changing drawer structure.

You can see the `Drawing color 1/2` using the icon editor in Section 7.3.3. The `Drawing color` contains the value `$FF` or 255. A few details about screen color changes were mentioned earlier. Using the icon editor write a value between 0 and three in the corresponding byte to change the color.

The best thing to do is experiment with these options. Don't be surprised, though, when you try to open one of the stored info files the `Info` screen opens for a moment then disappears again. This happens because no subdirectory exists for the window, which is apparently very important to a drawer icon. Enter the `Shell` and create a directory for every info file using the `makedir` command.

From there, you can then see all the new window colors. Some color combinations don't work very well. Others cancel out text. Work toward what you can see best in terms of contrast and readability.

**8**

**Error trapping**



## 8. Error trapping

Controlled error handling is an absolute necessity for large programs. This can save the user a lot of trouble from incorrect input. Very few programs are equipped with foolproof error checking. All the user has to do is type in input that the computer can't accept, and the system may crash. Error trapping is another facet of user-friendliness.

However, you must first know how errors are handled before knowing the location in the program where the error occurs. You can't find the latter on your own, but there are a few rules you can follow to help your programs run error-free.

This chapter shows you how you can foolproof your programs from errors. You'll read about routines that check for files on diskette without stopping from an error message, programs that generate requesters and even a demonstration of easy menu creation.

### *Workbench 2.0*

The Workbench 2.0 FD files were not available at the time this book was published, so the following programs have only been tested on Workbench 1.2 and 1.3. When the new 2.0 library FD files are available, the 2.0 `bmap` file can be created. The following programs may require minor changes to operate using the 2.0 `bmap` files.

---

## 8.1 Errors—and why

You may encounter errors even in programs which should not have errors. These programs include those available commercially and those you wrote yourself. We can divide these errors into two generic groups. The first group consists of errors that the programmer may have overlooked. These are the lines that result from leaving out a parenthesis or formula (syntax). This error type happens often when you or the user tries modifying a program. The only way to avoid syntax (or any) error is to test a program. But how?

First, write down a list of program sections that must be used. Note the program lines that operate under certain conditions. A number of errors may only occur under certain conditions. When you test the program, you have to test every section by calling them.

There are more error sources to annoy the user and programmer alike. A frequently encountered error is the `Subscript Out of Range` error. This happens when you try to access an array element past the default 10 elements of an array. Make a list of the arrays used, and make sure that you define them all properly. To make control easier, use one particular section for dimensioning arrays at the beginning of the program.

Math errors are another source of problems. Almost any calculation can lead to an error. Any slip of the hand can lead to an `Overflow` error, or a `Division by Zero`. Make sure your computations test for incorrect input, particularly in division, exponentiation, etc.

---

### 8.1.1 Disk access errors

Imagine this: You write the perfect data and address base. The user types in the name of the file that uses this program, and all he gets is a `File Not Found` error. AmigaBASIC returns an error message instead of a requester (as with Workbench).

A file under that name may exist, but you may have accidentally created it from another program, and it may have a different format from the program currently in use. The best that can happen is that the data can confuse the program. Most of the time the result is a `Type Mismatch` or similar error.

An even more aggravating error occurs when the file is on the right diskette, but the file you want is in another directory. The result is a File Not Found error.

---

### 8.1.2 User input errors

Any database program requires the user to enter values. However, even values have their limitations. Numbers should be within a certain range and/or have a certain number of decimal places; texts can only be a certain length or can only contain certain characters. The normal INPUT statement does not consider these conditions. It accepts numeric input as well as text. The wrong kind of input results in a Redo from Start error message, screen scrolling and repeated input.

The option of selecting only certain characters is unsupported. If the user goes past the assigned text length, the program cuts off these extra characters. This means that important information may be lost.

---

### 8.1.3 Menu errors

This is where errors get harder to pinpoint. User menus consist of entire subroutines and functions. The user selects an item and the program reacts. However, menus are not infallible.

One or more menu items may be unusable in certain conditions. Selecting a menu item which you should not use could lead to no reaction at all or even a system failure.

One harmless example could be a Save item on the fictional database program mentioned above. Selecting this item when no data has been entered doesn't crash the computer, but the data diskette now has a blank record that could be very difficult to later remove.

---

## 8.2 Trapping errors

It's possible to trap or even bypass errors. The keyword in solving these problems mentioned above is *prevention*.

### *Checking for errors*

As already mentioned, you can prevent simple error messages like `Division by Zero` by checking for these errors. This method is much more user-friendly than the program just stopping with an error. Program breaks give the user a new problem—he has to become a programmer and find the bug himself. Either you can set the program up to prompt for the correct data, or at least have the program jump to the beginning. These are crude, but either route is better than a break.

There are other ways to handle errors in BASIC. `ON ERROR GOTO` sends the system to a given line when an error occurs. The programmer assigns the line or routine. From there, the program can mention the nature of the error, or return to the area just after the incorrect line.

### *Requester*

The system requester is a much friendlier solution to error handling. For example: If the wrong diskette is in the disk drive, a window appears in the upper left hand corner. This window displays the text, "Please insert volume in any drive". From there, you can select the `Cancel` gadget to exit, or the `Retry` gadget to go on. The requester is the last chance you get to correct an error, without getting an error message. However, the requester is the only way to get around certain problems, such as exchanging diskettes when you only have one disk drive.

You may not get a chance to test your program under every circumstance, so you may have to create your own errors using subroutines. These errors can test your error checking thoroughly.

---

### 8.2.1 Error checking programs

Now that you've read through the theory, you can go on to practical programming. When an error occurs, nothing angers a user more than a program break. This is because most users aren't professional programmers, and even if they do program, they may not understand most of the material within a program written by someone else. You as a programmer must make things as simple for the user as possible. Programs should offer the user a chance to correct errors with some flexibility. You've already seen an example of user-friendly



programming in the system requester mentioned above; it gave you an opportunity to insert the correct diskette.

You can write this type of flexible programming. You're probably thinking of one way—open a window, write the text and read for the mouse click. That's one possible solution although it's also complicated. Instead, you can let the operating system draw a requester for you. You'll see how this is done (and how you can insert your own information) below.

Before you can program a requester, you must clearly know what you want the requester to do. It can serve the same occasions as those served by the Workbench requesters.

For example, you can set up a requester for a file that the system can't find on diskette. BASIC usually returns an error message. You must first suppress the error message, then call the requester that matches a File not Found error message.

### *Bypassing errors*

Since the File not Found error message usually accompanies opening a file that is not on the diskette or in the correct directory, you can't just read status during OPEN. A sequential file gives you another alternative, however. You can open the file using the APPEND option. Either the file exists as defined by a pointer, which allows adding to the file, or the file doesn't exist, and a new file opens. The LOF function lets you see if the file existed previously. A file exists if the file is at least one character in length; otherwise, the length is equal to zero. If the length is equal to zero, the program deletes the newly opened file.

Here is a program that demonstrates the above procedures:

```
' Check for existing file
' on diskette
'
' © by Wgb, August '87
'
FileName$="AmigaBasic2"
Mainprogram:
Again:
PRINT "Searching for the file: ";
WRITE FileName$
CALL CheckFile (FileName$)
IF exist=-1 THEN
PRINT "Okay, the file exists!"
ELSE
PRINT "File not found...sorry."
END IF
```

```

┌
END┌
┌
' -----┌
┌
SUB CheckFile (File$) STATIC┌
┌
  SHARED exist┌
  ┌
  OPEN File$ FOR APPEND AS 255┌
    exist=(LOF(255)>1)┌
  CLOSE 255┌
  ┌
  IF exist=0 THEN KILL File$┌
  ┌
END SUB┌

```

You can use a more elegant (and more complex) method to determine whether a file exists on diskette (see the program below). This other way uses a subroutine that returns a corresponding value: 1 (file exists) or 0 (file not found).

The `Lock` function must be defined as a function within a program. Then the memory location of the name is given, ending with a nullbyte. Next, the routine supplies information about how the file should be accessed. Since the Amiga is a multitasking computer, you can choose one access by itself (Access Mode = Exclusive Write (-1)) or read access by multiple tasks (Access Mode = Shared Access (-2)). The first option provides write and read access for a single user. The second option allows more than one program and/or user to read one file at the same time.

The new routine uses Shared Access, a returned value of -2.

The value returned by the function is equal to zero if no file exists on diskette. The value must go into memory, since it can allow another try at file access.

The access secured through this routine should cancel the list of parameters, since this list takes memory and time. You can use the `UnLock` function for this cancellation. The routine returns the value received by `Lock`.

```

' Test for existing file on diskette┌
' using dos.library┌
'┌
' © by Wgb, August '87┌
'┌
┌
DECLARE FUNCTION Lock& LIBRARY┌
┌
LIBRARY "T&T2:bmaps/dos.library"┌
┌

```

```

FileName$="AmigaBasic2"
MainProgram:
  Again:
    PRINT "Searching for the file: ";
    WRITE FileName$
    CALL CheckFile (FileName$)
    IF exist=-1 THEN
      PRINT "File exists!"
      PRINT "File Header begins at Block";blk;"on this
Disk."
    ELSE
      PRINT "File not found!"
    END IF
  LIBRARY CLOSE
END
' -----
SUB CheckFile (File$) STATIC
  SHARED exist,blk
  File$=File$+CHR$(0)
  accessRead%=-2
  DosLock%=Lock&(SADD(File$),accessRead%)
  IF DosLock%=0 THEN
    exist=0
  ELSE
    exist=-1
    blk%=PEEKL(DosLock%*4+4)
  END IF
  CALL UnLock(DosLock%)
END SUB

```

Now that you have some understanding of how to check for a file on diskette or in a subdirectory, you should learn how you can create a requester in BASIC.

It's possible to write a requester completely in BASIC as described above. However, it's much easier to use the requester routine provided by the Amiga's operating system. This operating system module is called the `AutoRequest` function. This function takes your text and gadget requests, and does the rest. The program below contains a subroutine that does all this for you. This subroutine returns a value which tells the main program where to branch from that point.

```

' Test for a file on
' diskette

```

```

'
' © by Wgb, June '87
'
DECLARE FUNCTION AllocRemember& LIBRARY
DECLARE FUNCTION AutoRequest& LIBRARY
DECLARE FUNCTION Lock& LIBRARY
LIBRARY "T&T2:bmaps/intuition.library"
LIBRARY "T&T2:bmaps/dos.library"
FileName$="T&T2:AmigaBasic2"
Mainprogram:
Again:
PRINT "File: ";
WRITE FileName$
CheckFile FileName$
IF exist=-1 THEN
    PRINT "File exists!"
    PRINT "File Header begins at Block";blk&;"on this
Disk."
ELSE
    Request FileName$
    IF res&=1 THEN GOTO Again
    PRINT "File not found!"
END IF
LIBRARY CLOSE
END
' -----
SUB CheckFile (File$) STATIC
    SHARED exist,blk&
    TestFile$=File$+CHR$(0)
    accessRead&=-2
    DosLock&=Lock&(SADD(TestFile$),accessRead&)
    IF DosLock&=0 THEN
        exist=0
    ELSE
        exist=-1
        blk&=PEEKL(DosLock&*4+4)
    END IF
    CALL UnLock(DosLock&)
END SUB
' -----

```

```

SUB Request (FileName$) STATIC¶
¶
  SHARED add&,st$,res&,offs%¶
¶
  Quest$(0)="Please insert volume containing"¶
  Quest$(1)="File "+FileName$¶
  Quest$(2)="Can't find the file!"¶
  yes$="Retry"¶
  no$="Cancel"¶
  bt%=2¶
  wid%=8*38¶
  hi%=8*9¶
  offs%=0¶
¶
  opt%=2^0+2^16¶
  req%=AllocRemember&(0,400,opt%)¶
  IF req%=0 THEN ERROR 7¶
¶
  add%=req%¶
¶
  t1%=add%¶
  FOR loop2=0 TO bt%-1¶
    st%=Quest$(loop2)¶
    MakeHeader add&,st$,1,5,offs%+3¶
    offs%=offs%+8¶
  NEXT loop2¶
¶
  st%=Quest$(bt%)¶
  MakeHeader add&,st$,0,5,offs%+3¶
¶
  st%=yes$¶
  t2%=add%¶
  MakeHeader add&,st$,0,5,3¶
¶
  st%=no$¶
  t3%=add%¶
  MakeHeader add&,st$,0,5,3¶
¶
  res%=AutoRequest&(WINDOW(7),t1%,t2%,t3%,0,0,wid%,hi%)¶
  CALL FreeRemember(0,-1)¶
¶
END SUB¶
¶
SUB MakeHeader (ptr&,Text$,md%,le%,te%) STATIC¶
¶
  SHARED add&¶
  Text$=Text$+CHR$(0)¶
¶
  POKE ptr&,1¶
  POKE ptr&+1,0¶
  POKE ptr&+2,2¶
  POKEW ptr&+4,le%¶
  POKEW ptr&+6,te%¶
  POKEL ptr&+8,0¶
  POKEL ptr&+12,SADD(Text$)¶
  IF md%=0 THEN¶

```

```

    POKEL ptr&+16,0¶
  ELSE¶
    POKEL ptr&+16,ptr&+20¶
  END IF¶
¶
  add&=ptr&+20¶
END SUB¶

```

***Program  
description***

First the routine must "know" which text you want displayed. Three texts lie in the routine: the main text and additional texts from which it can select. The last two texts are displayed in one line and are surrounded by borders. These make up the gadgets which you click. After establishing the text, you must set the window size. If the requester window is too small, the text simply spills over or gets overwritten, making it hard to read.

The text must be placed in memory in a certain structure, with a memory range reserved for this structure. The operating system function `AllocRemember` sets this range aside. It allows selection of a memory range based on preset criteria.

PUBLIC	2 <sup>0</sup>
CHIP	2 <sup>1</sup>
FAST	2 <sup>2</sup>
CLEAR	2 <sup>16</sup>

Any type of memory can be used, just as long as it is cleared beforehand. If no memory is available, then an error message appears.


Assume for the moment that enough memory is available. Then the text goes into the reserved area. This text must still appear in a certain format. BASIC programmers can use `POKEs` for this formatting. This command is useful for the use and design of your own programs only—AmigaBASIC normally doesn't require any `POKEing`.

The first loop brings the information text into the reserved memory range. Then the two gadgets transfer to RAM. If everything runs correctly, then the `AutoRequest` function can begin its task. It first takes the addresses of the first text, the two gadget texts and the requester's size. These return a value and then a result of 1, if the first gadget is clicked by the user.

This value can then be followed by branches to the main program. Either the system repeats the loading procedure, because of an incorrect diskette or non-existent file, or the loading procedure stops and returns the user to the main program.

## 8.2.2 Trapping user input errors

Now you have a requester that checks for existing files. An even more important aspect of error trapping is keeping the user's input correct. User entry has its own problems and errors.

The simplest and best solution is to write an input routine that reads the input, retains the desired characters and ignores the rest without returning an error message. The routine must ascertain which characters are "legal" and which ones aren't. This can be accomplished by calling a string which contains valid characters, and a routine that lists the valid number of characters. The subroutine handles the rest of the characters. The subroutine ends when the user presses the  key.

Most input goes to a specific position of the window for display. Coordinates set this position, saving the trouble of using the LOCATE command. You can display any text through the INPUT command.

```
' Input Routine ¶
'¶
' © by Wgb May '87¶
'¶
¶
Mainprogram:¶
¶
DEFINT a-z¶
KAlpha$="abcdefghijklmnopqrstuvwxyz"¶
GAlpha$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"¶
NAlpha$="01234567890+--*/.,="¶
ZAlpha$=" ,.?!-;/;:''¶
Possl$=KAlpha$+GAlpha$+ZAlpha$¶
¶
GetInput "Last name:",LName$,Possl$,10,10,20,0¶
GetInput "First name:",CName$,Possl$,10,12,20,0¶
WRITE LName$,CName$¶
END¶
¶
¶
SUBRoutines:¶
¶
SUB GetInput (Text$,In$,Possl$,x,y,Letter,Pointer)
STATIC¶
¶
Xold=POS(0)¶
Yold=CSRLIN¶
Length=0¶
LOCATE y,x¶
PRINT Text$;¶
x=x+LEN(Text$)¶
¶
```

```

ReadOut:¶
  Cursor x+Length,y¶
  GetInkey i$¶
  IF i$=CHR$(13) THEN GOTO Done¶
  IF i$=CHR$(8) THEN GOTO RubOut¶
  IF Letter=Length THEN GOTO ReadOut¶
  ¶
  f=INSTR(Poss1$,i$)¶
  IF f=0 THEN¶
    BEEP¶
    GOTO ReadOut¶
  END IF¶
  ¶
  PRINT i$;¶
  In$=In$+i$ : Length=Length+1¶
  GOTO ReadOut¶
¶
RubOut:¶
¶
  IF Length=0 THEN GOTO ReadOut¶
  Length=Length-1¶
  PRINT " ";¶
  In$=LEFT$(In$,Length)¶
  GOTO ReadOut¶
¶
Done:¶
¶
  PRINT " ";¶
  LOCATE Yold,Xold¶
  IF Pointer AND 1 = 1 THEN¶
    l=LEN(In$)¶
    In$=In$+SPACE$(Letter-1)¶
  END IF¶
  ¶
END SUB¶
¶
SUB Cursor (x,y) STATIC¶
  COLOR 3¶
  LOCATE y,x¶
  PRINT "_";¶
  LOCATE y,x¶
  COLOR 1¶
  ¶
END SUB¶
¶
SUB GetInkey (Key$) STATIC¶
  ¶
  KeyRead:¶
    Key$=INKEY$¶
    IF Key$="" THEN GOTO KeyRead¶
  ¶
END SUB¶

```



***Program description***

Before calling this new input routine, you should define a string or set of strings containing groups of valid characters. For example, you can set up a string of lowercase characters, another one of uppercase characters, a third one made of numbers and a fourth string of other characters. These strings let you easily set which characters you want accepted. The new INPUT command accepts these strings as a constant.

GetInput itself gives the text contained in the variable as a string with all valid characters, its position, the number of characters entered and a pointer. This pointer determines whether the input text should be filled with spaces where invalid characters appear in the text. This pointer sets to 1 if this is the case.

Unfortunately, editing numbers is difficult. You can do this, however, with the following combination:

```
GetInput "Number: ",Number$,NumChar$,10,10,8
Number=VAL(Number$)
```

When NumChar\$ only contains numbers, you can make sure that no nulls stand in Number if you don't want to. At any rate, you won't get a Redo from Start error from numeric input.

***Cursor placement***

The subroutine stores the current cursor position at the beginning. Since this position stays the same when the program exits the routine, then it doesn't affect output. The text appears in the specified position, and the computer sets the starting position for input. The length of the text entered is still set to zero.


The read loop displays the current input position of the cursor, and the routine waits for a key press. Any character received goes through the control functions. If you press the **(Backspace)** key, the character most recently entered deletes whenever possible. Pressing the **(←)** key branches immediately to the end of the routine.

Next the routine checks to see if the next character is "legal." The routine examines the string constants you set for this character. If the character is valid, it is added to the input string; if not, the Amiga BEEPS and returns to the beginning of the read loop. The routine then waits for the next character.

***Adaptation***

You can naturally adapt this routine to your own needs. For example, this program doesn't provide for letting the user move the cursor around within the text. It allows simple character deletion, but user input would be a lot simpler if you could insert or delete characters in the middle of the input line.

Another feature missing from this routine is the acceptance of no input at all. This can be practical when one value is used repeatedly, and needs little if any changing. You can add this to the beginning of the subroutine by predetermining the length of the parameters that must appear on the screen.

Up to now, the only way you could end a prompt or input was by pressing the  key. You could change the pointer so that when, say, the second bit is set, input ends only when the entry contains a minimum of one character.

## 8.3 Correcting errors

This section deals with corrections. Up until now, this chapter has assumed that correction is the last possible option for incorrect input. Most of the time no one takes this route, since real-time error checking in BASIC simply takes too much time. The self-generated input routine showed that examining every character can take up to three seconds to see if the character is good, bad or indifferent. This can't be helped.

For example, say you only want one word out of a hundred possible words entered. The system checks every single character as it appears in the combination. When you end the input, it checks all available words against this input, and you can display an error message or branch to the input as needed.

Most of the time, responses occur in which you have no say whether or not all values are recognized. Here, checking is only possible as a last resort. If the program establishes that a value is invalid, then it can simply be corrected. The program doesn't go on immediately after this. The user must again switch on correction to see if the value just entered is valid or not.

You can see that this is a fairly complicated subject. The entire matter of error-free user input is difficult, and unfortunately you can't hold a patent on this kind of routine. Every program has its own features, and its own error sources. As a programmer, you must be sympathetic to the user, and consider every place in a program where an error can happen. This means that testing should occur wherever an error can occur—better that than a program break later on.

---

### 8.3.1 Ghosting menu items

One answer to bypassing errors is to force inaccessible menu items to appear in ghost print. Programming with the `MENU` command leaves all menu items open to selection. This makes designing menus fairly simple. But what if you want to deactivate menu items so that the entire menu becomes inactive? You can save yourself a lot of work using `MENU number,item,0` to deactivate individual items. This gets to be time-consuming when you call this command to create an entire menu in ghost print.

That's where this program comes in. It uses a `SUB` routine named `Able`, which lets you assign the desired status to multiple menu items.

You can deactivate an entire block if you wish, or assign check marks to an active block of menu items. The function is a practical replacement for the MENU command.

```
' PullDownTest
'
' © by Wgb in June '87
'
DEFINT a-z
MainProgram:
  GOSUB MenuDefinition
  PRINT"All menus active."
  Pause 5
  PRINT "Disk menu inactive."
  Able 1,0,0,0
  Pause 5
  PRINT "Drawing type set."
  Able 2,4,0,2
  Pause 5
  PRINT"Single-color drawing only."
  Able 3,1,5,0
  Able 3,1,0,1
  Pause 5
  PRINT"GET from Brush menu available only."
  Able 4,1,4,0
  Able 1,0,0,1
  Able 3,1,5,1
  Pause 5
  PRINT"Press a key to end the program."
  Able 1,0,0,0
  Able 2,0,0,0
  Able 3,0,0,0
  Able 4,0,0,0
  Able 5,1,2,0
  WHILE INKEY$=""
    SLEEP
  WEND
  MENU RESET
END
MenuDefinition:
  RESTORE MenuData
```

```

¶
READ Number¶
FOR i=1 TO Number¶
  READ Items,Length¶
  FOR j=0 TO Items¶
    READ Item$¶
    IF j>0 THEN¶
      Item$=LEFT$(Item$+SPACES$(Length),Length)¶
      IF i=2 OR i=3 THEN¶
        Item$=" "+Item$¶
      END IF¶
    END IF¶
    MENU i,j,1,Item$¶
  NEXT j¶
NEXT i¶
¶
RETURN¶
¶
SUB Able(MenuNr,Item,Number,Types) STATIC¶
¶
FOR i=Item TO Item+Number¶
  MENU MenuNr,i,Types¶
NEXT i¶
¶
END SUB¶
¶
SUB Pause (Seconds) STATIC¶
¶
Elapsed&=TIMER+Seconds¶
WHILE TIMER<Elapsed&¶
WEND¶
¶
PRINT¶
¶
END SUB¶
¶
MenuData:¶
¶
DATA 5¶
DATA 7,15,Disk¶
DATA New,Load,Load as¶
DATA Save,Save as¶
DATA Disk Command,Quit¶
¶
DATA 7,9,Draw¶
DATA Freehand,Line,Lines¶
DATA Circle,Rectangle,Polygon¶
DATA Fill¶
¶
DATA 6,11,Color¶
DATA One Color,Multicolor,Palette¶
DATA Shadow,Wipe,Transparent¶
¶
DATA 6,9,Brush¶
DATA Load,Load as,Save¶
DATA Save as, Clear,Get¶

```

```

¶
DATA 4,11,Extras¶
DATA Workbench,Coordinates¶
DATA Blend Out,End¶
¶
add&=ptr&+20¶

```

***Program  
description***

First all variables are defined as integers. You may wonder why this program declares just these few variables. The reason is that when you define these at the beginning, the speed increases greatly—all math operations run as integer arithmetic. Besides, no problems crop up during the subroutine calls. If whole-number constants appear there, then the `Type Mismatch` error occurs (the subprograms want real number variables). Then you must either add integer signs to the constants in the command line, or adapt the variable types in the `SUB` program.

After variable definition, the main program branches to the `SUB` program `MenuDefinition`, which reads the menu texts from the `DATA` statements at the end of the listing.

Now look at the `SUB` program itself. After the `DATA` statements generate the menu data, the corresponding number goes to the outermost loop. This loop reads all the data concerning the number of menu items per menu and the length for each text. The last value is very important, since after defining a menu you can open the corresponding array. It has a maximum `X`-length based upon the longest text. You can only activate the individual menu items that actually contain characters. Every line that contains less than the maximum number of characters fills in with blank spaces. You can also activate the spaces at the end of every menu item.

With this, you can make a graphic, move the menu items to the start of the current item, and place a `REM` character in front of the line, filling the `Item$` variable with `SPACE$`. When you select the menu item, make sure you realize that this was done.

Look at the inner loop of the `SUB` routine. This takes the mentioned number of menu items from the `DATA` statements, and defines them with the `MENU` function. Menus 2 and 3 can have check marks before their items, when two spaces precede the texts of these menus. The addition of spaces following the texts changes when the number of menu items is greater than null. The menu title must not be corrected in this case.

Now on the main program itself. It displays the text stating that all menus are active. From this the user can determine the branch to a subroutine which waits for a given number of seconds then returns to the main program.

The design of this routine is fairly simple. First the computer calculates the time number which must be assigned to the given

number of seconds. Then this waits in a delay loop until the current time is reached.

The main program displays another text that says that the `Disk` menu is inactive. After this, the most important subroutines execute. The parameters state that the first menu's title, as well as the other menu titles, should be set with zeros. This sets all the other menu items to zero.

The `SUB` routine is easy to call, but designed with ease of use in mind. For each parameter, a loop executes which assigns the specific item types to all menu items.





**9**

**Machine language**



## 9. Machine language

Although AmigaBASIC is a good programming language, it's definitely not the fastest. The best language in which to program is machine language. The reason for this is that machine language instructions run a thousand times faster than BASIC commands. Every feature of the computer can be accessed from machine language. This includes some which are not available in BASIC such as system routines that can be called at almost any time.

Machine language is hard to learn, but by programming on the Amiga with a good assembler, you can learn it relatively quickly. The *AssemPro* package from Abacus is a very good assembler for the beginner.

Let's discuss the C programming language, which executes at about 1/10th the speed of machine language. You can do many of the same things in C that you can in machine language. Unfortunately, many of the unusual routines you might want to call in C require fairly extensive programming. For all that trouble, you might as well program in machine language.

You may not know anything about machine language. That's fine because this chapter doesn't demand that much knowledge from you. However, for the most effective Amiga programming, we recommend that you start learning 68000 machine language. Refer to an elementary book on the subject (*Amiga Machine Language* from Abacus is a good text on 68000 Amiga machine code) and study. We repeat: You won't need that knowledge for this chapter since we'll explain the code as carefully as we can.

This chapter contains many useful machine language programs and routines. You'll find BASIC extensions and unusual demonstrations of what you can do with the mouse. We've even included a program designed for zapping viruses (a problem in the Amiga community).

The routines below were written using Abacus' *AssemPro* assembler. For those of you who don't have access to an assembler, we've included BASIC loaders for each program whenever possible.

## 9.1 Division by zero handler

Dividing a value by zero is one of the most common causes of Guru Meditations in the early Amiga operating systems. Let's start by looking at what happens during and after a division by zero. This will allow us to think about how we can solve this problem.

The processor itself creates an exception. The Status register from the 68000 passes to a buffer. The processor automatically switches to supervisor mode. The Trace bit is erased to disable single-step mode. Now the program counter (PC), Status register, Instruction register (and its condition when the error occurred), access address and the Superstate word (which explains the processor's condition) are placed on the Supervisor stack.

After an Exception routine the processor continues working with the program, which releases the error, the RTE (ReTurn from Exception) instruction executes. The Exception vector appears at RTE. You can find a variation on this in the Shell SetAlert command, which waits for error flag settings, then determines from the Superstate word whether RTE or a Guru follows.

We'd like to clarify the following: When you have a program that has such a small denominator that it cannot be represented by a word, the denominator is rounded off to zero. Remember, the 68000 commands DIVU and DIVS process words only. Here's where the problem occurs: the 68000 can't divide by zero.

RTE returns us to the program. This return goes to the address after the command that releases the error. This is good, right? Wrong. Think about this: What usually happens when you divide by a very small number, for example .00001? You could get a fairly large number as a result.

The register which held the result now contains a very small value. The result is that all subsequent calculations must also be wrong as soon the next Exception is released. The program only returns nonsense, which is of no help to the user. We can assume that the denominator was infinitely small instead of just zero. This allows division once again. We can write the following in place of "infinitely small":

$$\frac{1}{\text{Denominator} = \text{infinitely\_large}}$$

For division by this denominator, we get help from the old rule: When dividing by a fraction, multiply by the inverse:

```
Counter = x
Inverse denominator = infinitely_large

Result = Counter * Inverse denominator
        = x * infinitely_large
        = infinitely_large
```

We insert the largest possible value for `infinitely_large` which the commands `DIVS` and `DIVU` can process, and the computation is correct in spite of Exception. We write an Exception routine which stores the correct values in the corresponding registers before you return with `RTE`.

Therefore, you must know which command releases the Exception because the highest possible value for `DIVU` is `$FFFF`, and for `DIVS`, `$FFF`. Furthermore, we must determine in which data register this value must be transferred. The number of the data register is found directly in the opcode, as the following table shows:

Command	Command code bits															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIVU	1	0	0	0	x	x	x	0	1	1	y	y	y	y	y	y
DIVS	1	0	0	0	x	x	x	1	1	1	y	y	y	y	y	y

Bits 9-11 (x) return the number of the data register where the result is stored. Bits 0-5 give the addressing type, which you don't need to know about here. Here's the machine language version of the program:

```
;Division by Zero - Handler; by SM'88
Init_Trap:                ;Install handler
    Move.l 4,A6           ;ExecBase to A6
    Move.l #Div_End-Div,D0 ;Handler-Length to D0

    MoveQ #1,D1           ;MEMF_Public
    Jsr -198(a6)         ;call AllocMem
    Move.l D0,New_Trap    ;get address

    Beq.s Init_End       ;End, when error
    Move.l D0,A1         ;Address to A1
    Lea Div,A0           ;CodeStart-Address

    MoveQ #(Div_End-Div)-1,D0 ;CodeLength-1
Copy_Code:                ;Handler copy
    Move.b (A0)+,(A1)+   ;Byte copy
    DBra D0,Copy_Code    ;Next Byte
    Move.l New_Trap,20    ;New Trap-Vector

Init_End:                 ;Adieu
    MoveQ #0,D0          ;No = 0 !!!
    Divu D0,D1           ;Go Ahead, Make My Day
    rts                  ;End
New_Trap:                 ;New Trap-Vector
    dc.l 0               ;1 LongWord
```

```

Div:
  MoveM.l D0-A6, -(sp)      ;DivisionByZeroHandler
  Move.l   62(sp), A0       ;All Register (s.u.)
  Move.w  -2(A0), D0        ;get PC from Stack
  Move.l   #$ffff, D1      ;get last command
  Move.l   #$ffff, D1      ;large value for Divu

  Btst    #8, D0           ;Was it Divu-instruction?
  Beq.s   GoOn            ;then continue
  Move.l   #$7fff, D1     ;Else Divs-Value

GoOn:
  Lsr.w   #7, D0          ;Data register transfer
  AndI.l  #28, D0        ;Scroll command bitwise
  AndI.l  #28, D0        ;Ignore register

  Move.l   D1, 0(sp, d0.l) ;Register and Stack save
  MoveM.l (sp)+, D0-A6    ;Register load
  Rte     ;Return from Exception
Div_End:
  Label:  SizeOf

End

```

You may have been wondering how the new result value arrives in the data register. When storing the entire register on the stack, the highest address register is always stored first. Then the other registers are used in descending order (see the third line of the program). They're simply used with four multiplied register numbers as an offset for the stack access. The system performs a Guru meditation if something does not function properly.

We released a division by zero Exception after installing the new Trap vector (division by zero occurs in line 20).

Here's a short BASIC routine that stores the program as an AmigaDOS command (you should call this command using your Startup-sequence):

```

OPEN "sys:c/DIVZERO" FOR OUTPUT AS 1
FOR i=1 TO 176
  READ a$
  a%=VAL("&H"+a$)
  PRINT #1, CHR$(a%);
NEXT
CLOSE 1
KILL "sys:c/DIVZERO.info"
datas:
DATA 0,0,3,F3,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0
DATA 40,0,0,1B,0,0,3,E9,0,0,0,1B,2C,78,0,4,20,3C,0,0
DATA 0,30,72,1,4E,AE,FF,3A,23,C0,0,0,0,36,67,18,22,40,41,F9
DATA 0,0,0,3A,70,2F,12,D8,51,C8,FF,FC,21,F9,0,0,0,36,0,14
DATA 70,0,82,C0,4E,75,0,0,0,0,48,E7,FF,FE,20,6F,0,3E,30,28
DATA FF,FE,22,3C,0,0,FF,FF,8,0,0,8,67,6,22,3C,0,0,7F,FF
DATA EE,48,2,80,0,0,0,1C,2F,81,8,0,4C,DF,7F,FF,4E,73,0,0
DATA 0,0,3,EC,0,0,0,3,0,0,0,0,0,0,0,12,0,0,0,1C
DATA 0,0,0,2A,0,0,0,0,0,0,0,3,F2,0,0,0,3,F2

```

---

## 9.2 Attention: Virus alert!

Computer viruses are a major topic of discussion wherever you hear computer users talking shop. Some people say that there are no such things as computer viruses—that it's all media hype, and that viruses don't really exist. We'll leave the debate up to others. However, we personally believe in viruses.

Computer viruses spread with amazing speed. The most common viruses seen on the Amiga are the SCA virus and the Byte Bandit virus. At one time, you could simply say, "I'm safe as long as I use commercial software." This is no longer true today: This is a problem you should take very seriously.

### *Who's responsible?*

When viruses come under debate, the first question that crops up is, "Who's responsible for these programs?" Viruses came from the world of the software pirate who cracks the protection on a commercial program and perhaps adds a virus to it. The pirated copies spread viruses even further, and may even return to the manufacturers from whence the original programs came. Suddenly original games from the factory have viruses on them. This has happened to a few game manufacturers. A respected software manufacturer for another 68000-based computer was shocked that two magazine executives had planted a virus on a fairly expensive and powerful piece of illustration software.

How does a virus propagate? Generally the virus program hides in the boot block of a disk. When you boot using this disk, the operating system loads the boot sectors (the first two sectors of a disk). Usually these sectors contain the initialization routine for the DOS library. The `Install` command writes this routine to disks to make the disks bootable. The operating system jumps directly to the boot routine, which is exactly what the virus wants.

The virus copies itself to an area of memory, changes system vectors and goes through the DOS initialization. This places them in the system unnoticed. If you place another boot disk in a drive, one virus writes itself to the boot block. Another type of virus does this during a reset.

Unfortunately, many programs start with a loader in the boot block which the virus simply overwrites, destroying the disk. Moreover, infected computers suffer different interruptions caused by the virus. This is first noticeable when the virus decides that enough disks have been infected (it keeps track of the number of disks it's infected).

The sole enemy of Amiga viruses when we wrote this book is the `Install` command, which simply overwrites the infected boot block. Some known viruses can recognize the use of `Install`. When the `Install` command starts writing to the boot block, the virus sets a flag somewhere during the write procedure and reformats the disk. A system infected in such a manner is usually beyond help.

### 9.2.1 The ultimate virus killer

You should store the following program in the Startup-sequence of every boot disk you use. It examines and deletes the system vectors which the viruses can use. The entire boot disk must be reconfigured using the `Install` command. By disabling the virus program in memory, this cannot be written back to the disk after `Install`.

```

;VIRUS-KILLER V1.0; by SM'87

start:
  move.l 4,a6                ;EXECBASE at a6
  moveq #0,d1                ;Flags: no Virus here
  tst.l 46(a6)               ;Test, for distorted Cool-Capture
  beq.s noSCA                ;Wasn't SCA-Virus
  clr.l 46(a6)               ;Cool-Capture clear
  addq.b #1,d1                ;Bit 0 set
noSCA:
  cmpi.w #$FC,148(a6)       ;Vertical Blank Interrupt normal?
  beq.s noVBI                ;no?
  addq.b #2,d1                ;Bit 1 set
  bra.s ClearTag             ;KickTag-Pointer clear
noVBI:
  tst.l 550(a6)              ;KickTag-Pointer changed?
  beq.s GoOn                 ;no?
  addq.b #4,d1                ;Bit 2 set
ClearTag:
  clr.l 550(a6)              ;Pointer cleared
  clr.l 554(a6)              ;Pointer cleared
GoOn:
  move.b d1,Virusflag        ;Flag reserved
  lea dosname,a1             ;Address of Lib names
  moveq #0,d0                 ;Version is the same
  jsr -552(a6)                ;OpenLibrary
  move.l d0,dosbase          ;Reserve Library-Base
  beq errfix                  ;Branch on Error
  move.l d0,a6                ;Prepare for DOS call
  jsr -60(a6)                 ;Get Output-handle
  move.l d0,Outpuhandle       ;and store
  beq errfix                  ;Branch when error
  move.l #title,d2            ;Title at d2
  move.l #titleend-title,d3   ;Text-Length

```



```

    jsr writeout           ;Text output
    tst.b Virusflag       ;Virusflag test
    bne.s Virusfound     ;Branch on Virus
    move.l #clean,d2      ;Clear message
    move.l #cleanend-clean,d3 ;Length
    jsr writeout         ;Text output
    bra errfix           ;Program end
Virusfound:              ;Virus is active
    btst #0,Virusflag    ;Cool-Capture?
    beq.s notsca         ;No
    move.l #scaV,d2      ;Message
    move.l #scaVend-scaV,d3 ;Length
    jsr writeout         ;Text output
notsca:
    btst #1,Virusflag    ;VB-Interrupt?
    beq.s notvbi         ;No
    move.l #bbVvbi,d2    ;Message
    move.l #bbVvbiend-bbVvbi,d3 ;Length
    jsr writeout         ;output
    bra.s bbfound        ;next Message
notvbi:
    btst #2,Virusflag    ;Kicktag?
    beq.s errfix         ;No
    move.l #bbVtag,d2    ;Message
    move.l #bbVtagend-bbVtag,d3 ;Length
    jsr writeout         ;output
bbfound:
    move.l #bbv,d2       ;message
    move.l #bbvend-bbv,d3 ;Length
    jsr writeout         ;output
errfix:
    move.l 4,a6
    move.l dosbase,d0
    beq.s quit
    move.l d0,a1
    jsr -414(a6)
quit:
    moveq #0,d0
    rts
writeout:
    move.l outphandle,d1
    jmp -48(a6)
dosname:dc.b "dos.library",0
title:dc.b $c,$9b,"1;31;42m - Virus-Killer"
       dc.b " V1.0 - ",10,13
       dc.b "(c) 1988 by S. Maelger",10,13,10
       dc.b $9b,"0;31;40m"
titleend:align
clean:dc.b "No symptoms for Virus"
       dc.b "-Infection found !",10,13,10
cleanend:align
scaV:dc.b "Reset-Vector Cool-Capture has"
       dc.b " been used!",10,13
       dc.b "SCA-Virus suspected !",10,13
       dc.b "Virus in memory destroyed.",10,13,10
scaVend:align

```

```

bbVvbi:dc.b "Vertical Blank Interrupt has "
          dc.b "been used!",10,13
bbVvbiend:align
bbV:dc.b "Byte-Bandit-Virus suspected!",10,13
          dc.b "Virus in memory destroyed.",10,13,10
bbVend:align
bbVtag:dc.b "KickTagPointer is no longer"
          dc.b " in operating system!",10,13
bbVtagend:align
dosbase:dc.l 0
outputhandle:dc.l 0
Virusflag:dc.b 0
          end

```

Here is the BASIC loader version of the Virus check program listed above:

```

OPEN "sys:c/Virus_chk" FOR OUTPUT AS 1
FOR i=1 TO 800
  READ a$
  a%=VAL("&H"+a$)
  PRINT #1,CHR$(a%);
NEXT
CLOSE 1
KILL "sys:c/Virus_chk.info"
datas:
DATA 0,0,3,F3,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0
DATA 0,0,0,A4,0,0,3,E9,0,0,0,A4,2C,78,0,4,72,0,4A,AE
DATA 0,2E,67,6,42,AE,0,2E,52,1,C,6E,0,FC,0,94,67,4,54,1
DATA 60,8,4A,AE,2,26,67,A,58,1,42,AE,2,26,42,AE,2,2A,13,C1
DATA 0,0,2,8C,43,F9,0,0,1,12,70,0,4E,AE,FD,D8,23,C0,0,0
DATA 2,84,67,0,0,AA,2C,40,4E,AE,FF,C4,23,C0,0,0,2,88,67,0
DATA 0,9A,24,3C,0,0,1,1E,26,3C,0,0,0,46,4E,B9,0,0,1,8
DATA 4A,39,0,0,2,8C,66,16,24,3C,0,0,1,64,26,3C,0,0,0,2A
DATA 4E,B9,0,0,1,8,60,0,0,6A,8,39,0,0,0,0,2,8C,67,12
DATA 24,3C,0,0,1,8E,26,3C,0,0,0,5E,4E,B9,0,0,1,8,8,39
DATA 0,1,0,0,2,8C,67,14,24,3C,0,0,1,EC,26,3C,0,0,0,29
DATA 4E,B9,0,0,1,8,60,1C,8,39,0,2,0,0,2,8C,67,24,24,3C
DATA 0,0,2,52,26,3C,0,0,0,32,4E,B9,0,0,1,8,24,3C,0,0
DATA 2,16,26,3C,0,0,0,3B,4E,B9,0,0,1,8,2C,78,0,4,20,39
DATA 0,0,2,84,67,6,22,40,4E,AE,FE,62,70,0,4E,75,22,39,0,0
DATA 2,88,4E,EE,FF,D0,64,6F,73,2E,6C,69,62,72,61,72,79,0,C,9B
DATA 31,3B,33,31,3B,34,32,6D,20,2D,20,56,69,72,75,73,2D,4B,69,6C
DATA 6C,65,72,20,56,31,2E,30,20,2D,20,A,D,28,63,29,20,20,31,39
DATA 38,38,20,62,79,20,53,2E,20,4D,61,65,6C,67,65,72,A,D,A,9B
DATA 30,3B,33,31,3B,34,30,6D,4E,6F,20,73,79,6D,70,74,6F,6D,73,20
DATA 66,6F,72,20,56,69,72,75,73,2D,49,6E,66,65,63,74,69,6F,6E,20
DATA 66,6F,75,6E,64,20,21,A,D,A,52,65,73,65,74,2D,56,65,63,74
DATA 6F,72,20,43,6F,6F,6C,2D,43,61,70,74,75,72,65,20,68,61,73,20
DATA 62,65,65,6E,20,75,73,65,64,21,A,D,53,43,41,2D,56,69,72,75
DATA 73,20,73,75,73,70,65,63,74,65,64,20,21,A,D,56,69,72,75,73
DATA 20,69,6E,20,6D,65,6D,6F,72,79,20,64,65,73,74,72,6F,79,72,64
DATA 2E,A,D,A,56,65,72,74,69,63,61,6C,20,42,6C,61,6E,6B,20,49
DATA 6E,74,65,72,72,75,70,74,20,68,61,73,20,62,65,65,6E,20,75,73
DATA 65,64,21,A,D,0,42,79,74,65,2D,42,61,6E,64,69,74,2D,56,69
DATA 72,75,73,20,73,75,73,70,65,63,74,65,64,21,A,D,56,69,72,75

```

DATA 73,20,69,6E,20,6D,65,6D,6F,72,79,20,64,65,73,74,72,6F,79,65  
DATA 64,2E,A,D,A,0,4B,69,63,6B,54,61,67,50,6F,69,6E,74,65,72  
DATA 20,69,73,20,6E,6F,20,6C,6F,6E,67,65,72,20,69,6E,20,6F,70,65  
DATA 72,61,74,69,6E,67,20,73,79,73,74,65,6D,21,A,D,0,0,0,0  
DATA 0,0,0,0,0,0,0,0,0,0,3,EC,0,0,0,16,0,0,0,0  
DATA 0,0,0,30,0,0,0,36,0,0,0,42,0,0,0,52,0,0,0,5C  
DATA 0,0,0,68,0,0,0,6E,0,0,0,76,0,0,0,82,0,0,0,8E  
DATA 0,0,0,96,0,0,0,A2,0,0,0,AA,0,0,0,B2,0,0,0,BE  
DATA 0,0,0,C8,0,0,0,DO,0,0,0,DC,0,0,0,E2,0,0,0,EE  
DATA 0,0,0,F8,0,0,1,A,0,0,0,0,0,0,3,F2,0,0,3,F2

## 9.3 Machine language and BASIC

To call machine language routines in BASIC, a long variable must transfer the starting address of the routine. We demonstrate this on the operating system's Reset routine, which begins at memory location \$FC0000. Unfortunately, BASIC always causes difficulties when handling long variables. The BASIC interpreter almost exclusively computes with floating point variables, and later converts the number into long values.

The error frequently encountered is that the normal floating point variables are accurate to only a couple of decimal places. When calculating in long values, the converted result is low by a value between 1 and 5. To get around this you must either use machine language routines which are more accurate in long value arithmetic, or use strings:

```
SMreset=&=CVL(CHR$(0)+CHR$(&HFC)+MKI$(0))
```

For frequent use of system routines and machine language programs, you have the option of declaring all variables that have no label as long:

```
DEFLNG a-z
SMreset=CVL(CHR$(0)+CHR$(&HFC)+MKI$(0))
SMreset
```

**Note:**

Be careful! When you enter the above example and start it, the BASIC interpreter jumps to the Reset routine. This is only an example of jumping into a machine language routine, the routine happens to reset the computer. If you want to perform a reset in a BASIC program, there is a much better and faster method available:

```
POKEL 32, CVL(CHR$(0)+CHR$(&HFC)+MKI$(0))
```

But we digress—we're supposed to be talking about machine language. To take the next step in the direction of adding BASIC command enhancements, we write a short routine which switches the Power LED on and off. You may remember that this is what the Amiga does when you reset it (or when it resets on its own). The essential routine looks like this:

Code	Mnemonic
089000100BFE001	BCHG #1, \$BFE001 ;switch brightness
4E5	RTS ;that is all

Now comes the question of where we can put the code. That's not as easy as it sounds because the address where the code begins must always be even, otherwise a Guru #3 occurs (addressing error). The 68000 processor can only process commands at even addresses, so each command must be found at an even address. Here BASIC doesn't tell you what's going on, and places its variables bitwise in the variable buffer, when enough memory exists. To be absolutely certain that the address is even, the memory location should be allocated by the system. That can be done with the Exec routine AllocMem. The following routine uses AllocMem:

```
' LED-FLICKER.BAS
DEFLNG a-z
DECLARE FUNCTION AllocMem LIBRARY
LIBRARY"t&t2:bmaps/exec.library"
SMmagic=AllocMem(10,1) '10 bytes, memory area public RAM
FOR i%=0 TO 4
  READ Power$
  POKEW SMmagic+i%*2,VAL("&H"+Power$)
NEXT
DATA 879, 1, BF, E001, 4E75
PRINT "Watch the power LED flicker"
FOR i%=1 TO 20 'switch 20 times
  a!=TIMER+.5 'delay 0,5 seconds
  WHILE a!>TIMER:WEND 'wait
  SMmagic 'call Assembler routine
NEXT
FreeMem SMmagic,10 'free memory location
LIBRARY CLOSE 'close SYS
```

We have no problem with the simple routine, which uses none of its own registers. Any assembler on the market will accept the following example:

```
MOVE.L DO,Dataregister0
...
Dataregister0:dc.l 0
```

This routine defines a label within a program, onto which the data register D0 should be placed. When corresponding codes like those in the Power LED program are poked into memory, the Guru Meditation is guaranteed to appear in a hurry. The reason is the addressing method that was used in the above MOVE.L. It is addressed absolutely.

The absolute addressing cannot be used because we never know where the program will end up in memory once it loads.

The code must first approximate the respective beginning address, which we want to use. A good assembler has an option for letting the programmer create program counter relative code (PC relative). In every case you must write:

```
LEA dataregister0(PC),A0
MOVE.L D0,(A0)
...
```

If you only programmed using PC Relative code your program could not use all assembler instruction, and the programming would be very extensive. We want to show you a way you can use every machine language instruction, completely avoiding any memory allocations, bypassing any fancy load routines, and finally, allowing function calls to C programs.

### 9.3.1 Assembler and C programs from BASIC

We could just give you the facts about this kind of access. Instead, let's look at the entire (non-BASIC) program. Whether it's a favorite game, word processor or BASIC interpreter, programs are absolute addressed, yet the 68000 requires relative addressing so the program can be placed anywhere in memory. If you stop to think about this for a while, you may come to the conclusion that the addresses in the program can be present only as offsets. Shortly after loading the program the operating system must calculate the correct addresses so the program can adjust itself to the addresses it will be loaded at.

The problem: How to anticipate it. The operating system knows which commands must be coded. This is saved with a Link module, which contains this information, from the assembler. A saved assembler program usually consists of a code segment (your program), a data segment and a BSS segment. The BSS segment reserves free memory locations for the program immediately after loading. Also, a saved assembler program contains at least one link segment.

You can examine what the operating system does by using the load routine to start the program (your program should not start by itself). This is contained in the `dos.library` and can be called from BASIC with no problem. This routine is called `LoadSeg` and needs the name of the program ending with a null byte. Memory allocation and all other work is done from `LoadSeg`. The syntax looks like this:

```
MainSegment=LoadSeg ( SADD( "Filename" + CHR$(0) ) )
```

You need the `UnloadSeg` routine to release the reserved memory locations, which must be given the return value from `LoadSeg`:

```
UnloadSeg MainSegment
```

There are difficulties with the value that was received from `LoadSeg`. Because AmigaDOS is written in BCPL (a compiler language), the value is handled as a BCPL value. The BCPL language refers to this value as the Main or Code segment. For your own use, you must convert this to an address. DOS does so as if the memory consists of only long values placed one after another. Consequently, it handles the number of the long value from the start of memory as the return value.

To get a correct address, this value must be multiplied by four (four bytes per long value). This address is not exactly given in the program. The BCPL pointer to the next segment is stored in the first long value of the main segment. This segment begins with another BCPL pointer to another segment, and so on. The program appears following this pointer, so use the following routine to access a machine routine:

```
DEFUNG a-z¶
DECLARE FUNCTION loadseg LIBRARY¶
LIBRARY "T&T2:bmaps/dos.library"¶
'change the file name for your machine language program¶
bcpl.segstart=loadseg(SADD("df1:asm.prg"+prg"+CHR$(0)))¶
Segment=bcpl.segstart*4¶
Routine=Segment+4¶
Routine¶
unloadseg bcpl.segstart¶
LIBRARY CLOSE¶
```

Here's another item. To see the start addresses of the individual segments, insert the following code preceding the call to Routine:

```
PRINT "Main program begins at address";Routine
i%=1
WHILE Segment<>0
  PRINT i%;" Segment begins at address";Segment+4
  Segment=PEEKL(Segment)*4
WEND
```

Now we come to the parameter statement. BASIC deals with assembler routines in the same way that the C language deals with program sections. The parameters are stored in long value form on the stack, so in the following example parameter P3 goes onto the stack first, then P2 and finally P1. So, parameter P1 becomes the first parameter taken from the stack in the called program. After the parameters, the routine jump with JSR (Jump to SubRoutine) takes the code to the last return jump address on the stack:

```
Routine P1,P2,P3
```

The return jump address of the routine is on the stack:

```
Stackpointer + 12 = P3 (Long)
Stackpointer + 8 = P2 (Long)
Stackpointer + 4 = P1 (Long)
Stackpointer + 0 = return jump address (using JSR)
```

When you load a C routine with the above program, you can call it exactly as you would call an assembler program. When you can, you should always use machine language because machine code generated from C is actually slower than “real” machine language. Some BASIC compilers create code that executes faster than C because the operating system routines are called exactly the way you write BASIC commands.

Portions of the operating system are programmed in C and this is why the Amiga is so slow. C requires large amounts of memory for execution. Only a small percentage of the operating system is written in C, which comprises the largest section of the memory needed by the operating system. This explains why the operating system gets smaller and faster. The developers gradually replace the C routines with machine language routines as they have the opportunity to upgrade the system.

In a C program, the parameters are in the brackets in the same order as they are given in BASIC. How do you get from machine language to C? We should start by mentioning that you cannot change any of the registers without first saving their current values. These values should be stored at the beginning of the program:

```
START: MOVEMEM.L D0-A6,-(A7) ;save registers: with that
                                ;you find 15 registers and
                                ;the return jump address on
                                ;the stack.
                                ;the first parameter is
                                ;at SP+(16*4) = 64(SP)
MOVEM.L 64(SP),D0-D2          ;pick up the three
                                ;parameters
...
END: MOVEM.L (A7)+,D0-A6      ;register taken from stack
RTS
```

You may be wondering how the values returned to the BASIC program are accessed. With this in mind, we give the address of a variable as a parameter, and access the return values using the VARPTR or SADD function. The address is taken from the stack in the machine language program and written to the return value. When you return multiple values, an address is given to an array variable (be careful with strings where you get the address of the string descriptor, which consists of 5 bytes).



## 9.3.2 BASIC enhancement: ColorCycle

To illustrate parameter transfer in a machine language program, here's an enhancement to AmigaBASIC. It allows you to rotate the colors as is allowed in *DPaint*®:

```

;-----;
; BASIC-Extension  ColorCycle  SMmagic'88 ;
;-----;
; Syntax: AddressRoutine WINDOW(7),from,to ;
; from=Start  color; tos=End color        ;
; Color always "from" --> "to"           ;
; from < to: Rotation up                  ;
; from > to: Rotation down                ;
;-----;
Cycle:  MOVEM.L D0-A6,-(SP)  ;Reserve register on Stack
        MOVE.L 4,A6         ;get ExecBase from A6
        LEA GFXNAME,A1     ;Library-Name at A1
        MOVEQ #0,D0        ;Version is same
        JSR -552(A6)       ;OpenLibrary call
        TST.L D0           ;Test, is Base available
        BEQ.S Exit         ;When not, then End
        MOVE.L D0,A6       ;GfxBase at A6
        MOVE.L 64(SP),A0   ;WindowBase attended to
        MOVE.L 46(A0),A0   ;ScreenBase determined
        ADD.L #44,A0       ;ViewPort of Screens in A0
        MOVE.L 4(A0),A1    ;ColorTable to A1
        MOVE.L 4(A1),A1    ;ColorMap determined
        LEA CTab,A2        ;User Buffer to A2
        MOVEQ #15,D0       ;15 Longs (32 Words)
CopyCT: MOVE.L (A1)+,(A2)+ ;ColorMap copied
        DBRA D0,CopyCT     ;(when not changed else
        MOVEM.L 68(SP),D0-D1 ;get start- and End color
        ANDI.W #31,D0      ;should it be more than 31
        ANDI.W #31,D1      ;ditto
        LSL.B #1,D0        ;*2 (use as offset)
        LSL.B #1,D1        ;ditto
        LEA CTab,A1        ;Address of our buffer
        MOVE.W (A1,D1.W),D2 ;reserve last color
        CMP.B D0,D1        ;determine rotation dir.
        BEQ.S C1Lib        ;both colors same ???
        BGT.S Up           ;colors rotate up
Down:   MOVE.W 2(A1,D1.W),(A1,D1.W) ;colors then down
        ADDQ.B #2,D1       ;increment Offset
        CMP.B D0,D1        ;End reached?
        BNE.S Down        ;no? then next color
        BRA.S SetLC       ;close
Up:     MOVE.W -2(A1,D1.W),(A1,D1.W) ;color downward
        SUBQ.B #2,D1       ;decrement Offset
        CMP.B D0,D1        ;bottom reached?
        BNE.S Up          ;no? then next color

```

```

SetLC:  MOVE.W D2, (A1,D1.W)    ;color reserved
        MOVEQ #32,D0           ;32 colors set
        JSR -192(A6)           ;LoadRGB4 (a0=VP,a1=Ctab,d0)
ClLib:  MOVE.L A6,A1           ;GfxBase at A1
        MOVE.L 4,A6            ;get ExecBase
        JSR -414(A6)          ;Library closed
Exit:   MOVEM.L (SP)+,D0-A6    ;Register from Stack
        RTS                   ;end
GFXNAME: DC.B "graphics.library",0,0 ;Library-Name
CTab:   DS.W 32                ;32 Words buffer
        END

```

The code is configured so that you can either assemble it in PC relative or normal mode. The following BASIC program calls and demonstrates the ColorCycle routine:

```

'Load first routine:
DEFLNG a-z
DECLARE FUNCTION loadseg LIBRARY
LIBRARY"T&T2:bmaps/dos.library"
a=loadseg(SADD("T&T2:ColorCycle"+CHR$(0)))
prg=a*4+4
'A little graphic
FOR i%=0 TO 3
  LINE (0,i%*40)-STEP(80,40),i%,bf
NEXT
'Demo: Rotate colors forward and backward
FOR i%=0 TO 50
  t!=TIMER+.2
  WHILE t!>TIMER
  WEND
  prg WINDOW(7),1,3
NEXT
FOR i%=0 TO 50
  t!=TIMER+.2
  WHILE t!>TIMER
  WEND
  prg WINDOW(7),3,1
NEXT
'Release memory
unloadseg a
LIBRARY CLOSE

```

The following BASIC loader generates the ColorCycle file on disk:

```

OPEN "COLORCYCLE" FOR OUTPUT AS 1
FOR i=1 TO 300
  READ a$
  a$="&H"+a$
  PRINT#1,CHR$(VAL(a$));
NEXT
CLOSE 1
DATA 0,0,3,F3,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,40,0,0,3A,0,0
DATA 3,E9,0,0,0,3A,48,E7,FF,FE,2C,78,0,4,43,F9,0,0,0,94,70,0
DATA 4E,AE,FD,D8,4A,80,67,76,2C,40,20,6F,0,40,20,68,0,2E,D1
DATA FC,0,0,0,2C,22,68,0,4,22,69,0,4,45,F9,0,0,0,A6,70,F,24
DATA D9,51,C8,FF,FC,4C,EF,0,3,0,44,2,40,0,1F,2,41,0,1F,E3,8

```



```

INTNAME: DC.B "intuition.library",0 ;Library-Name
MOUSE:   DC.L 0,$3000300,$7A007A0,$1FF01FF0,$3FF03FF0
         DC.L $30F83FF8,$3DFC3FFC,$7BFC7FFC,$30FE3FFE
         DC.L $3F863FFE,$1FEF1FFF,$3FDE3FFE,$1F861FFE
         DC.L $FFC0FFC,$3F803F8,$E000E0,$3800380,$7E007E0
         DC.L $3400340,0,$600060,$700070,$200020,0
         END ;End the data
;One typical call in BASIC:
;ZZZ WINDOW(7),0 'sleeping
;ZZZ WINDOW(7),1 'normal

```

The following BASIC generator creates the machine code for this routine:

```

OPEN "ZZZ" FOR OUTPUT AS 1
FOR i=1 TO 260
  READ a$
  a$="&H"+a$
  PRINT#1,CHR$(VAL(a$));
NEXT
CLOSE 1
DATA 0,0,3,F3,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,40,0,0,31,0,0,3
DATA E9,0,0,0,31,48,E7,FF,FE,2C,78,0,4,43,F9,0,0,0,50,70,0,4E
DATA AE,FD,D8,4A,80,67,32,2C,40,20,6F,0,40,20,2F,0,44,8,0,0,0
DATA 67,6,4E,AE,FF,C4,60,12,43,F9,0,0,0,62,70,16,72,10,74,0
DATA 26,2,4E,AE,FE,F2,22,4E,2C,78,0,4,4E,AE,FE,62,4C,DF,7F,FF
DATA 4E,75,69,6E,74,75,69,74,69,6F,6E,2E,6C,69,62,72,61,72,79
DATA 0,0,0,0,0,3,0,3,0,7,A0,7,A0,1F,F0,1F,F0,3F,F0,3F,F0,30
DATA F8,3F,F8,3D,FC,3F,FC,7B,FC,7F,FC,30,FE,3F,FE,3F,86,3F,FE
DATA 1F,EF,1F,FF,3F,DE,3F,FE,1F,86,1F,FE,F,FC,F,FC,3,F8,3,F8
DATA 0,E0,0,E0,3,80,3,80,7,E0,7,E0,3,40,3,40,0,0,0,0,0,60,0
DATA 60,0,70,0,70,0,20,0,20,0,0,0,0,0,0,0,0,0,0,3,EC,0,0,0,2,0,0
DATA 0,0,0,0,0,A,0,0,0,30,0,0,0,0,0,0,0,3,F2,0,0,3,F2
DATA BECKER

```

This file is loaded with the LoadSegment routine. Here is a BASIC program that loads and demonstrates the routine:

```

'Load first routine:¶
DEFLNG a-z¶
DECLARE FUNCTION loadseg LIBRARY¶
LIBRARY"T&T2:bmaps/dos.library"¶
a=loadseg(SADD("T&T2:ZZZ"+CHR$(0)))¶
prg=a*4+4¶
'Demo: Mouse sleeping¶
¶
  prg WINDOW(7),0 :'Mouse sleeping¶
¶
FOR i%=0 TO 5000¶
NEXT¶
¶
prg WINDOW(7),1 :'mouse normal¶
¶
'Release memory¶
unloadseg a¶
LIBRARY CLOSE¶

```

# **10**

## **Input and output**



# 10. Input and output

Users normally think of input and output (or *I/O*) as the contact between the Amiga and its *peripherals*. Peripherals are devices such as printers, joysticks and disk drives. The Amiga treats the built-in disk drive as an external device, since disk drives are considered external by most computers.

The advanced user may wonder how to communicate with these devices on a more-or-less direct basis. The Amiga has a basic I/O system. Every device has a corresponding software module which converts the basic control codes into device-specific codes. These software modules have file extensions of `.device`. Some of these device files lie in KickStart memory, while some are on the Workbench diskette.

You must create an *I/O request block* to handle I/O. This is placed in a reserved area of memory. This section is defined as follows:

```
add& = starting memory
address: B=byte: W=word: L=longword
```

add&+	type	definition
0	L	pointer to previous node
4	L	pointer to next node
8	L	type
9	B	priority
10	L	pointer to name string
14	L	pointer to message port
18	W	message length in bytes
20	L	pointer to device block
24	L	pointer to unit block
28	W	I/O command
30	B	flags
31	B	I/O error number
32	L	actual array
36	L	length array
40	L	data array
44	L	offset array

Along with this structure a *message port* must be created. This is a segment of memory set aside for I/O communication.

The I/O request block can be thought of as a letter traveling through the mail. When a multitasking system such as the Amiga's appears to be handling several tasks at once, it's really handling one program at a time for a moment. When one of these programs must communicate with another "simultaneously running" program, this communication

travels as a message. The I/O request block is one messenger of this type. The BASIC interpreter of AmigaBASIC runs the I/O device as a program running parallel to the BASIC program. This hands the message block to the address of the other task. In reality, the data block stays in one place instead of moving around in memory. The foreign task passes final control over this memory. As long as an I/O request block shifts to another task, our own program doesn't access the memory. When the other task processes the message, control over this memory returns to our own program.

We won't bore you with the technical background involved, since that goes far beyond the scope of this book. If, however, you wish to pursue the details of this process, we recommend that you read any one of the books about Amiga system programming.

The following pages list a number of examples with which you can access disk drives and printers without much programming knowledge.

***Workbench  
2.0***

The Workbench 2.0 FD files were not available at the time this book was published, so the following programs have only been tested on Workbench 1.2 and 1.3. When the new 2.0 library FD files are available, the 2.0 `bmap` file can be created. The following programs may require minor changes to operate using the 2.0 `bmap` files.



# 10.1 Direct disk access

Trackdisk.device handles up to four 3-1/2" disk drives. With a little help, you can directly manipulate data stored on diskette.

Every Amiga floppy disk drive has two read/write heads, one head for each side of a diskette. The diskette is divided into 80 *cylinders* per side. Each cylinder consists of 11 sectors. Each sector contains 512 usable data bytes, as well as 16 sector processing bytes. The total file capacity is:

```

                2 heads
x   80 cylinders
x   11 sectors
x  512 bytes
-----
    900,120 bytes (880K)
    
```

There are 28,160 bytes unavailable to the user in addition to this 880K.

Now on to the programming: The following program has six high-level SUBS as well as four sublevel routines. All you'll need for now are the first six SUBS.

## Disk access

OpenDrive opens any disk drive. This SUB asks for the number of the disk drive (0=internal drive, 1-3=external drives). CreateBuffer reserves segments of memory. This routine asks for the variable containing the starting address of the memory to be allocated, as well as the desired buffer's size in bytes. DiscardBuffer releases the memory reserved by CreateBuffer. The only argument required is the starting address of the buffer. WorkDrive sends an I/O command to any open drive. CloseDrive closes a disk drive. MotorOff turns off the disk drive motor.

The following program lets you open any disk drive and view any one of the 1760 sectors. The program displays the data found in hexadecimal notation.

```

#####
'#                                     #
'# Program: Disk - Monitor             #
'# Author:  tob                         #
'# Date:    8/8/87                     #
'# Version: 1.0                         #
'#                                     #
#####
#
    
```

```

DECLARE FUNCTION OpenDevice% LIBRARY¶
DECLARE FUNCTION AllocMem% LIBRARY¶
DECLARE FUNCTION AllocSignal% LIBRARY¶
DECLARE FUNCTION FindTask% LIBRARY¶
DECLARE FUNCTION DoIO% LIBRARY¶
¶
LIBRARY "T&T2:bmaps/exec.library"¶
LIBRARY "T&T2:bmaps/graphics.library"¶
¶
var:      '* Variable¶
          DIM SHARED reg&(3,1)¶
¶
main:     '* Demonstration program¶
          PRINT TAB(20);"DISK MONITOR"¶
          PRINT¶
          LINE INPUT "Which drive (0 - 3)? ...."; dr$¶
          dr% = VAL(dr$)¶
¶
          OpenDrive dr%¶
          CreateBuffer d0&, 512&¶
¶
          LINE INPUT "Which sector (0 - 1759)?
          ....";sec$¶
          sec% = VAL(sec$)¶
          WorkDrive dr%, 2, sec%, d0&¶
          MotorOff dr%¶
¶
          WHILE sec$ <> "end"¶
            CLS¶
            PRINT "Sector ";sec%¶
            PRINT¶
            c% = 3¶
            FOR loop1% = 0 TO 512 - 1 STEP 25¶
              FOR loop2% = 0 TO 24¶
                check% = PEEK( d0& + loop1% + loop2%)¶
                h$      = HEX$(check%)¶
                IF LEN(h$) = 1 THEN¶
                  h$ = "0" + h$¶
                END IF¶
                he$ = he$ + h$¶
                IF check% < 31 THEN¶
                  d$ = d$ + "?"¶
                ELSE¶
                  d$ = d$ + CHR$(check%)¶
                END IF¶
                IF loop2% + loop1% = 512 - 1 THEN¶
                  loop2% = 24¶
                END IF¶
              NEXT loop2%¶
              LOCATE c%, 1¶
              c% = c% + 1¶
              out$ = he$ + " " + d$¶
              CALL Text(WINDOW(8), SADD(out$),
LEN(out$))¶
              he$ = ""¶
              d$ = ""

```

```

        NEXT loop1%
        LOCATE 1,20
        LINE INPUT "Which sector (0 - 1759, end)?
....";sec$
        sec% = VAL(sec$)
        WorkDrive dr%, 2, sec%, d0%
        MotorOff dr%
    WEND
¶
        DiscardBuffer d0%
        CloseDrive dr%
        CLS
        PRINT "All OK."
¶
        LIBRARY CLOSE
        END
¶
SUB OpenDrive (nr%) STATIC
IF reg&(nr%, 0) = 0 THEN
    CreatePort "disk.io", 0, port%
    IF port% = 0 THEN ERROR 255
    CreateStdIO port%, io%
    dev$ = "trackdisk.device" + CHR$(0)
    er% = OpenDevice% (SADD(dev$), nr%, io%, 0)
    IF er% <> 0 THEN
        RemoveStdIO io%
        RemovePort port%
        io% = 0
        port% = 0
        ERROR 255
    ELSE
        reg&(nr%, 0) = io%
        reg&(nr%, 1) = port%
    END IF
ELSE
    io% = reg&(nr%, 0)
    port% = reg&(nr%, 1)
END IF
END SUB
¶
SUB CloseDrive (nr%) STATIC
IF reg&(nr%, 0) <> 0 THEN
    io% = reg&(nr%, 0)
    port% = reg&(nr%, 1)
    CALL CloseDevice(io%)
    RemoveStdIO io%
    RemovePort port%
    reg&(nr%, 0) = 0
    reg&(nr%, 1) = 0
END IF
END SUB
¶

```

```

SUB MotorOff (nr%) STATIC¶
  io% = reg%(nr%, 0)¶
  IF io% <> 0 THEN¶
    POKEW io% + 28, 9¶
    POKEW io% + 36, 0¶
    e% = DoIO% (io%)¶
  ELSE¶
    BEEP¶
  END IF¶
END SUB¶
¶
SUB CreateBuffer (add%, size%) STATIC¶
  IF size% > 0 THEN¶
    size% = size% + 4¶
    opt% = 2^16¶
    add% = AllocMem% (size%, opt%)¶
    IF add% <> 0 THEN¶
      add% = add% + 4¶
      POKEW add% - 4, size%¶
    END IF¶
  ELSE¶
    BEEP¶
  END IF¶
END SUB¶
¶
SUB DiscardBuffer (add%) STATIC¶
  IF add% <> 0 THEN¶
    size% = PEEKL (add% - 4)¶
    add% = add% - 4¶
    CALL FreeMem (add%, size%)¶
  END IF¶
END SUB¶
¶
SUB WorkDrive (nr%, command%, sector%, buffer%) STATIC¶
  td.sector% = 512¶
  io% = reg%(nr%, 0)¶
  td.offset% = sector%*td.sector%¶
  IF io% <> 0 THEN¶
    POKEW io% + 28, command%¶
    POKEW io% + 36, td.sector%¶
    POKEW io% + 40, buffer%¶
    POKEW io% + 44, td.offset%¶
    er% = DoIO% (io%)¶
  ELSE¶
    BEEP¶
  END IF¶
END SUB¶
¶
'--- sub level routines for advanced use only ---¶
¶
SUB CreateStdIO (port%, result%) STATIC¶
  opt% = 2^16¶
  result% = AllocMem%(62, opt%)¶
  IF result% = 0 THEN ERROR 7¶
  POKE result% + 8, 5¶

```

```

        POKEL result& + 14, port&¶
        POKEW result& + 18, 42¶
    END SUB¶
¶
    SUB RemoveStdIO (io&) STATIC¶
        IF io& <> 0 THEN¶
            CALL FreeMem(io&, 62)¶
        ELSE¶
            ERROR 255¶
        END IF¶
    END SUB¶
¶
    SUB CreatePort (port$, pri%, result&) STATIC¶
        opt& = 2^16¶
        byte& = 38 + LEN(port$)¶
        port& = AllocMem&(byte&, opt&)¶
        IF port& = 0 THEN ERROR 7¶
        POKEW port&, byte&¶
        port& = port& + 2¶
        sigBit% = AllocSignal%(-1)¶
        IF sigBit% = -1 THEN¶
            CALL FreeMem(port&, byte&)¶
            ERROR 7¶
        END IF¶
        sigTask& = FindTask&(0)¶
¶
        POKE port& + 8 , 4¶
        POKE port& + 9 , pri%¶
        POKEL port& + 10, port& + 34¶
        POKE port& + 15, sigBit%¶
        POKEL port& + 16, sigTask&¶
        POKEL port& + 20, port& + 24¶
        POKEL port& + 28, port& + 20¶
        FOR loop% = 1 TO LEN(port$)¶
            char% = ASC(MID$(port$, loop%, 1))¶
            POKE port& + 33 + loop%, char%¶
        NEXT loop%¶
        CALL AddPort(port&)¶
        result& = port&¶
    END SUB¶
¶
    SUB RemovePort (port&) STATIC¶
        byte& = PEEKW(port& - 2)¶
        sigBit% = PEEK (port& + 15)¶
        CALL RemPort(port&)¶
        CALL FreeSignal(sigBit%)¶
        CALL FreeMem(port&-2, byte&)¶
    END SUB¶

```

**Variables**

reg& ()	contains important internal I/O addresses (e.g., I/O-request and I/Oport)
dr%	disk drive number (0-3)
d0&	512-byte buffer
sec%	sector number (0-1759)
loop1%	loop

loop2%	loop
check%	character read (decimal)
h\$	character read (hexadecimal)
he\$	line read (hexadecimal)
d\$	line read (decimal)
c%	current screen line

OpenDrive ()

nr%	number of open drive (0-3)
port&	message port address
io&	I/O block address
dev\$	trackdisk.device ended with null
er%	I/O error; 0=no error

Create-Buffer ()

size&	buffer size in bytes
opt&	options: 216 = CLEAR MEMORY
add&	address of found memory

WorkDrive ()

td.sector%	=512: bytes per sector
io&	I/O block address
td.offset&	byte offset from sector 0: multiple of 512
er%	I/O error code

CreatePort ()

port\$	name of new port
pri%	priority of new port (-128 to 127)
result&	address of found port (output)
opt&	memory option: 216 = CLEAR MEMORY
byte&	size of needed memory
sigBit%	signal bit
sigTask&	address of AmigaBASIC task handler
char%	ASCII code of character read

***Program description***

First the program establishes the number of the disk drive the user wants accessed. `OpenDrive` opens this drive. Next the program internally checks for whether the drive is already open, and whether an entry already lies in `reg&()`. If not, `CreatePort` turns to a message port named `disk.io`. The starting address lies in `port&`. If no port exists (`port&=0`), then an error occurs. Otherwise, `CreateStdIO` opens a port, passing the address over to the already existing port. The starting address of the I/O block goes to `io&`. The drive opens through the `Exec` function `OpenDevice%()`. When this routine returns a value greater than or less than 0, the drive cannot be opened. Possible reasons: Another task has control of the drive; an `Open` was not preceded by a `Close`; the drive doesn't exist; the drive is not connected. In such a case the port and I/O block are released, the variables return to null status and an error message appears on the screen. The address of the new port and the new I/O block goes into `reg&()`.

The program opens a buffer large enough to hold the data of one diskette sector (minimum size). This 512-byte buffer is created by `CreateBuffer`; the buffer's starting address appears in `d0&`. The user is asked for the sector he wants to view. The `SUB WorkDrive` reads this sector and places it in the buffer `d0&` (CMD READ, the read command, =2). This `SUB` fills the I/O request blocks the necessary values, and calls the `Exec` function `DoIO%()`, sent to the disk drive through the command block.

After `WorkDrive` finishes its work, the diskette motor must be switched off. `WorkDrive` turns the motor on, but not off. The reason: Multiple disk access can be tiring when you have to turn the disk drive on and off every time you need to go to the diskette. The `MotorOff SUB` turns the motor off. The `Motor` command (=9) in the I/O block writes the contents sent from `DoIO%()`.

Now comes the data in memory starting from `d0&`. Two loops read the values from the buffer and place these on the screen in decimal and hexadecimal notation. The program then asks for additional sectors. You either enter a number (0-1759) or the word "end" to quit. The first response calls up a new sector, the second response releases the buffer and closes the disk drive `CloseDrive` (the program tests for open disk drives through `reg&()`). If there is an open drive, the addresses of the I/O and portblock are read. `RemoveStdIO` and `RemovePort` release this structure, and the drive closes through `CloseDevice()`. Finally the program deletes the entries from `reg&()`.

## 10.1.1 The `trackdisk.device` commands

When you want to examine your own programs, you should use the `WorkDrive SUB` to access these programs. This `SUB` gives you the following commands:

### *Read data*

Command number: 2  
 Command call: `Workdrive number%, 2, sector%,  
 buffer&`

If your buffer is larger than 512 bytes, you can naturally load more than one sector at a time. The entry within the I/O array 36 must be changed: For example, `5*td.sector%` instead of `td.sector%` when your buffer can handle that much data.

### *Write data*

Command number: 3  
 Command call: `Workdrive number%, 3, sector%,  
 buffer&`

Writes the buffer contents to the given sector on the diskette.

### *Note:*

If you don't know what you're doing when writing to diskette, you could destroy the disk data. If you want to change the data on a sector, read the sector with command 2, edit the buffer and write the sector back to diskette.

You can write more than one sector at a time (see `Read data` above).

### *Motor*

Command number: 9  
 Command call: `Workdrive number%, 9, 0, 0`

Manipulates I/O array 36: 0=motor off, 1=motor on. `IO_Actual` returns the current status.

### *Format disk*

Command number: 11  
 Command call: `Workdrive number%, 11, track%,  
 trackbuf&`

This command writes a completely new track to diskette. One track consists of 11 sectors. `track%` must therefore be a multiple of 11. The track buffer must be large enough for 11 sectors. The command ignores all data previously stored on this track and can even overwrite hard errors.



### 10.1.2 Multiple disk drive access

The SUBS on the previous program are constructed in such a way that you can access up to four disk drives at a time. You must open every drive using the `OpenDrive` command and close each one individually later. In addition, every drive must have its own buffer available for copying data. You can naturally use a single buffer.

### 10.1.3 Sector design

A sector shows just a small part of a diskette's true contents. From this we can see the design of sectors (numbers are given in longwords [four-byte arrays]):

#### Root block (sector 880)

0	type (=2)
1	0
2	0
3	hashtable size (512-224)
4	0
5	checksum
6-77	hashtable: sector numbers in which main directory files or subdirectories lie
78	= FFFFFFFF (-1) when bitmap is valid
79-104	number of sector containing the bitmap (normally one sector). Every bit of the bitmap corresponds to a diskette sector and indicates whether the sector is free (bit set) or occupied (bit unset).
105	day of last date diskette was altered
106	minutes
107	ticks (1/50 second)
108-120	diskette name: BCPL string: first byte gives the number of characters in a string (maximum 30)
121	day of date this diskette was initialized
122	minutes
123	ticks
124	0
125	0
126	0
127	root-ID = 1

**User directory block**

0	type (=2)
1	header key (number of this sector)
2	0
3	0
4	0
5	checksum
6-77	hashtable: sector numbers in which main directory files or subdirectories lie
78	reserved
79	protection bits (EXEC, DEL, READ, WRITE)
80	0
81-104	commentary string (BCPL string)
105	day of date diskette was created
106	minutes
107	ticks (1/50 second)
108-123	directory name: BCPL string
124	next entry with equal has value
125	sector number of root directory
126	0
127	user directory (=2)

**File header block**

0	type (=2)
1	number of this sector
2	total number of data sectors for this file
3	number of used data block slots
4	sector number of first data block
5	checksum
6-77	sector numbers of data blocks
78	unused
79	protection bits (EXEC, DEL, READ, WRITE)
80	total file size in bytes
81-104	commentary string (BCPL string)
105	day of date diskette was created
106	minutes
107	ticks (1/50 second)
108-123	filename: BCPL string
124	next entry with equal hash value
125	sector number of root directory
126	0 or sector number of first extended block (file list block)
127	file type (=FFFFFFD)

**File list block**

0	type (=1)
1	number of this sector
2	total number of data blocks in list
3	number of used data block slots
4	first data block
5	checksum
6-77	sector numbers of data blocks
78-123	unused
124	0
125	sector number of root directory
126	next extended block
127	file type (=FFFFFFD)

**Data block**

0	type (=8)
1	number of this sector
2	sequence of data block
3	number of data in bytes
4	sector number of next data block
5	checksum
6-127	data

---

## 10.2 Memory handling

The memory system of the Amiga is extremely flexible. This is because the memory locations can be changed to fit the situation instead of having fixed memory. Unlike its predecessors, the Amiga has no specific memory set aside for machine language user applications. This kind of memory layout makes no sense to a multitasking computer where several programs must share memory.

Here are the most popular methods of memory handling.

---

### 10.2.1 Reserving memory through variables

Every time you assign a value to a variable you take a piece of working memory and reserve part of the stack for this value. The amount of memory reserved depends on the variable type. For example, a long integer variable like `f&` would reserve 4 bytes. Now you can use this memory for other purposes as well. The starting address comes from the BASIC `VARPTR` command:

```
VARPTR (f&)
```

You need more than four bytes to use variable arrays (`DIM f& (100)` reserves 400 bytes) or strings (`a$=SPACE$(100)` reserves 100 bytes). The starting address of the string comes from the call:

```
SADD (a$)
```

It should be mentioned here that the starting address of string memory is variable. Every new string definition can move old strings around in memory. Every memory access changes the starting address in memory. This means that the memory is not well suited for set data structures. The following method is a more practical route.

---

### 10.2.2 Allocating memory

The `AllocMem()` command gives you as much memory as you ask for, as long as that much memory is free. You can choose between three options:

Public memory	2 <sup>0</sup>
Chip memory	2 <sup>1</sup> (DMA and special purpose chips)
Fast memory	2 <sup>2</sup> (all other applications)
Clear memory	2 <sup>16</sup> (automatically clears memory)

The following SUBS reduce memory handling to a minimum.

```

#####
'#                                     #
'# Program: Memory Handler           #
'# Author:   tob                      #
'# Date:     8.12.87                  #
'# Version:  2.0                      #
'#                                     #
#####
┌
DECLARE FUNCTION AllocMem& LIBRARY┌
┌
LIBRARY "T&T2:bmaps/exec.library"┌
┌
demo:      '* reserve 4500 bytes┌
           PRINT "Memory left after reserving 4500
bytes: ";┌
           PRINT FRE(-1)┌
┌
           GetMemory mem&, 4500&┌
┌
           PRINT "Current memory status: ";┌
           PRINT FRE(-1)┌
┌
           FreeMemory mem&┌
┌
           PRINT "Ending memory status: ";┌
           PRINT FRE(-1)┌
┌
           LIBRARY CLOSE┌
           END┌
┌
┌
SUB GetMemory (add&, size&) STATIC┌
  IF size& > 0 THEN┌
    opt& = 2^16┌
    size& = size& + 4┌
    add& = AllocMem&(size&, opt&)┌
    IF add& <> 0 THEN┌
      POKEL add&, size&┌
      add& = add& + 4┌
    END IF┌
  END IF┌
END SUB┌
┌

```

```

SUB FreeMemory (add&) STATIC¶
  IF add& > 0 THEN¶
    add& = add& - 4¶
    size& = PEEKL (add&)¶
    CALL FreeMem(add&, size&)¶
  END IF¶
END SUB¶

```

***Program  
description***

The principle should be obvious from the example. It uses `GetMemory` to reserve a memory segment of any size for your use. Two variables return the address variable in which you'll find the starting address of the memory segment (or 0 if there isn't enough memory available) and the size of the desired segment. Reserving 1000 bytes is as simple as:

```
GetMemory myMem&, 1000&
```

You'll find the starting address of the segment in the variable `myMem&`:

```
PRINT myMem&
```

When you no longer need the memory, you can return it to the system with the call:

```
FreeMemory myMem&
```

You cannot go past the memory size allocated for this segment, since `GetMemory` actually has up to four bytes of memory reserved holding the bytes beyond the segment size.

---

## 10.3 The Printer Device

The printer device gives the BASIC programmer the opportunity to use the printer he has connected to his Amiga. If you have the proper printer driver for your printer and the printer device available, the interfacing usually runs flawlessly, and with a minimum amount of hassle for the user.

This chapter shows you how to set up your printer to perform tasks that are a bit unusual. You'll find a program available to let you control your printer outside of Preferences. In addition, this chapter contains a program which enables easy printed hardcopy from an open window.

---

### 10.3.1 Controlling printer parameters

Open a computer magazine and look through the advertisements. You'll see literally hundreds of printers on the market, all shouting at the user, "Buy me!" These printers all carry different price tags, different methods of producing printed matter (dot matrix, daisy wheel, inkjet, laser, thermal), and different qualities of printing.

Each printer type has its own special strengths and weaknesses. Here are some general descriptions of these pros and cons:

- Thermal printers are very inexpensive and very quiet, but require special paper and may not be graphic compatible.
- Daisy wheel printers produce excellent print for letters, theses, etc., but cannot print any graphics except the most rudimentary graphic output using available characters.
- Dot matrix printers can produce graphics (even in color), but often the NLQ (near letter quality) mode is inadequate for professional text printing.
- Laser printers have speed, high resolution and graphic capability, but the price is prohibitive for the average user.
- Inkjet printers are quiet, efficient and fairly clear printers, but their graphic reproduction varies greatly.

You can easily see that each printer type described above can address at least one of your personal printing needs. The Amiga can help. Once you select the printer you're using in the Change Printer screen of Preferences (on the Workbench disk), the Amiga automatically converts general printer commands and printer-specific command codes to your printer. These codes make your programs either completely compatible with your printer type, or as compatible as possible.

Preferences usually governs this print quality, but you can override the control using the following program. This program should give you some ideas of how the printer device communicates with the printer, and how you can adapt the printer device to your own needs.

Remember, do not enter the ¶ characters in the following program. We use this symbol to show where a BASIC line actually ends. We had to split some lines when formatting this book. You should enter these lines on one line in Amiga BASIC. The ¶ character shows where a line actually ends.

```

*****¶
'* Program: Read Printer Data¶
'* Date: May 28' 88¶
'* Author: tob¶
'* Version: 1.3¶
*****¶
CLS¶
PRINT "Searching for the .bmap files!¶
'EXEC-LIBRARY¶
DECLARE FUNCTION AllocMem& LIBRARY¶
DECLARE FUNCTION DoIO& LIBRARY¶
DECLARE FUNCTION OpenDevice% LIBRARY¶
DECLARE FUNCTION AllocSignal% LIBRARY¶
DECLARE FUNCTION FindTask& LIBRARY¶
LIBRARY "t&t2:bmaps/exec.library"¶
init:  '¶
      GetPrinterData¶
      ¶
      PRINT "Printer-Name      : "; prt.name$¶
      PRINT "Printer-Type     : "; prt.typ$¶
      PRINT "Color capability  : "; prt.color$¶
      PRINT "Characters per line : "; prt.columns%¶
      PRINT "Number of fonts    : "; prt.charsets%¶
      PRINT "Number of raster lines: ";prt.rows&¶
      PRINT "Max. num. Dots horiz : ";prt.xdots&¶
      PRINT "Max. num. Dots vert.  : ";prt.ydots&¶
      PRINT "Density: Dots/Inch h. : ";prt.xdotspi&¶
      PRINT "Density: Dots/Inch v. : ";prt.ydotspi&¶
      ¶
      END¶
      ¶
SUB GetPrinterData STATIC¶
  SHARED prt.DRPreq&¶
  SHARED prt.typ$, prt.colour$, prt.name$¶
  SHARED prt.columns%, prt.charsets%¶
  SHARED prt.rows&, prt.xdots&, prt.ydots&¶

```



```

SHARED prt.xdotspi&, prt.ydotspi&¶
¶
DIM prt.color$ (9)¶
DIM prt.printer$ (3)¶
¶
prt.color$ (1) = "Black-White"¶
prt.color$ (2) = "Yellow-Magenta-Cyan"¶
prt.color$ (3) = "Yellow-Magenta-Cyan or Black-White"¶
prt.color$ (4) = "Yellow-Magenta-Cyan-Black"¶
prt.color$ (5) = "Blue-Green-Red-White"¶
prt.color$ (6) = "Black-White Invers"¶
prt.color$ (7) = "Blue-Green-Red"¶
prt.color$ (8) = "Blue-Green-Red or Black-White"¶
prt.color$ (9) = "Blue-Green-Red-White"¶
¶
prt.printer$(0) = "b/w Text Printer"¶
prt.printer$(1) = "b/w Graphics"¶
prt.printer$(2) = "Color Text Printer"¶
prt.printer$(3) = "Color Graphics"¶
¶
¶
OpenPrinter¶
¶
prt.printerdata& = PEEKL (prt.DRPReq& + 20)¶
prt.extendeddata& = (PEEKL (prt.printerdata& + 92) + 12)¶
prt.name$ = ""¶
prt.name& = PEEKL (prt.extendeddata&)¶
prt.printer% = PEEK (prt.extendeddata& + 20)¶
prt.color% = PEEK (prt.extendeddata& + 21)¶
prt.columns% = PEEK (prt.extendeddata& + 22)¶
prt.charsets% = PEEK (prt.extendeddata& + 23)¶
prt.rows& = PEEKW (prt.extendeddata& + 24)¶
prt.xdots& = PEEKL (prt.extendeddata& + 26)¶
prt.ydots& = PEEKL (prt.extendeddata& + 30)¶
prt.xdotspi& = PEEKW (prt.extendeddata& + 34)¶
prt.ydotspi& = PEEKW (prt.extendeddata& + 36)¶
¶
prt.typ$ = prt.printer$ (prt.printer%)¶
prt.colour$ = prt.color$ (prt.color%)¶
¶
count = NULL ¶
char = PEEK (prt.name& + count)¶
¶
WHILE char <> NULL¶
    prt.name$ = prt.name$ + CHR$ (char)¶
    count = count + 1¶
    char = PEEK (prt.name& + count)¶
WEND¶
¶
ClosePrinter¶
¶
END SUB¶
¶
SUB OpenPrinter STATIC¶
    SHARED mem.chunk&¶
    SHARED prt.DRPReq&¶
    ¶

```

```

mem.clear& = 2^16      'clear memory before task¶
mem.DRPReq% = 62      '62 Bytes for DRPStructure¶
mem.port% = 37        '37 Bytes for Port-Struct.¶
mem.label% = 4        '4 Bytes for Organization¶
mem.size% = mem.DRPReq% + mem.port% + mem.label%¶
¶
mem.chunk& = AllocMem& (mem.size%, mem.clear&)¶
IF mem.chunk& = NULL THEN ¶
    ERROR 7           'OUT OF MEMORY ERROR¶
END IF¶
¶
prt.label& = mem.chunk&¶
prt.DRPReq% = mem.chunk& + mem.label%¶
prt.port% = mem.chunk& + mem.label% + mem.DRPReq%¶
prt.name$ = "printer.device" + CHR$(0)¶
¶
POKEL prt.label&, mem.size% 'allocate memory size¶
¶
status% = OpenDevice% (SADD(prt.name$), 0, prt.DRPReq%, 0)¶
IF status% <> NULL THEN¶
    PRINT "Printer is not available."¶
    CALL FreeMem (mem.chunk&, mem.size%)¶
    EXIT SUB¶
END IF¶
END SUB¶
¶
SUB ClosePrinter STATIC¶
    SHARED mem.chunk&¶
    ¶
    mem.size% = PEEKL (mem.chunk&)¶
    prt.DRPReq% = mem.chunk& + 4¶
    CALL CloseDevice (prt.DRPReq%)¶
    CALL FreeMem (mem.chunk&, mem.size%)¶
END SUB

```

**Variables**

prt.DRPReq&	I/O DumpRastPort structure (starting address here)
prt.typ\$	Printer category
prt.colour\$	Color capability
prt.name\$	Printer name
prt.columns%	Characters per line
prt.charsets%	Number of available fonts
prt.rows&	Number of pins available on print head
prt.xdots&	Max. number of pixels in the X-direction
prt.ydots&	Max. number of pixels in the Y-direction
prt.xdotspi&	Horizontal resolution (pixels per inch)
prt.ydotspi&	Vertical resolution (pixels per inch)

GetPrinterData():

prt.color\$()	Array—color types
prt.printer\$()	Array—printer types
prt.printerdata&	Starting address, PrinterData structure
prt.extendeddata&	Starting address, ExtendedData structure
prt.name&	Starting address, name string
prt.printer%	Printer type code number
prt.color%	Color type code number
count	Counter
char	Read character

OpenPrinter:

mem.chunk&	Starting address, reserved memory
mem.clear&	= 2 <sup>16</sup> ; set available memory to 0
mem.DRPReq%	= 62; reserve 62 bytes for structure
mem.port%	= 38; reserve 38 bytes for structure
mem.label%	= 4; reserve 4 bytes for organization
mem.size%	Memory requirement in bytes
prt.label&	Starting address, label memory
prt.DRPReq&	Starting address, DumpRastport structure
prt.port&	Starting address, Port structure
prt.name\$	Device name
status%	0 = everything's okay

**Program description**

When you look at it, you discover that the previous program consists of three subprograms:

```
GetPrinterData
OpenPrinter
ClosePrinter
```

The user will find the `GetPrinterData` subprogram most interesting. This subprogram internally calls the other two subprograms. The structure named `PrinterExtendedData` contains the information needed by the other subprograms. To arrive at this, it is necessary to first open the printer through `printer.device`. This is done using the `OpenPrinter` subprogram.

Next the `Exec` function `AllocMem()` allocates memory for two structures: a `Port` structure and a `DumpRastPort` structure. In addition, `AllocMem()` reserves four bytes. These bytes are eventually used as storage for the absolute memory size listed for `FreeMem()`.

When this method is used the `Exec` function `OpenDevice()` opens the printer. This call returns a `Status` report to the system. As long as the `Status` value doesn't equal zero, the printer cannot be opened.

Possible causes: Another task may be currently accessing the printer, or the printer wasn't properly closed before this access.

When the printer opens, the `DumpRastPort` structure contains a pointer to a structure named `PrinterData`. When the pointer is reset, it points to the `PrinterExtended` data structure, in which the necessary data is saved.

The data is read and stored in the correct variables. Then the printer is closed once again. This is accomplished using a call of the `ClosePrinter` routine. You must use this routine. When the printer is opened but not closed by the same program, it cannot be accessed until the computer is reset.

Here is an example of the program output:

```
Printer-Name      : EpsonQ
Printer-Type     : Color Graphics
Color capability  : Yellow-Magenta-Cyan-Black
Characters per line : 80
Number of fonts  : 10
Number of raster lines: 24
Max. num. Dots horiz : 720
Max. num. Dots vert. : 0
Density: Dots/Inch h. : 90
Density: Dots/Inch v. : 180
```

---

### 10.3.2 Graphic dumps using the printer device

The following program is an example of printer control programming. It shows you the essentials of printing the current contents of your BASIC window to the printer as a graphic hardcopy or screen dump.

This program supports all the special flags included in operating system 1.3. These flags let you reduce the size of a window's contents, enlarge the window, distort its structure, center it and more.

Remember, do not enter the ¶ characters in the following program. We use this symbol to show where a BASIC line actually ends. We had to split some lines when formatting this book. You should enter these lines on one line in Amiga BASIC. The ¶ character shows where a line actually ends.

```

*****
* Program: Graphic-Dump
* Date: May 28 1988
* Author: tob
* Version: 1.3
*****
PRINT "Searching for .bmap files!"
EXEC-LIBRARY
DECLARE FUNCTION AllocMem& LIBRARY
DECLARE FUNCTION DoIO& LIBRARY
DECLARE FUNCTION OpenDevice& LIBRARY
DECLARE FUNCTION AllocSignal& LIBRARY
DECLARE FUNCTION FindTask& LIBRARY
LIBRARY "T&T2:bmaps/exec.library"
init:  '
      CIRCLE (100,100),100
      PRINT STRING$(100,"_")
      special.nothing = 0 'no Special effects
      special.milcols = 1 'X-Dimension in 1/100  '0
Inch
      special.milrows = 2 'Y-Dimension in 1/100  '0
Inch
      special.fullcols = 4 'Maximum X-measurement  'g
      special.fullrows = 8 'Maximum Y-measurement  'g
      special.fraccols = 16 'fraction of max. X-
measurement
      special.fracrows = 32 'ditto, for Y-measurement
      special.center = 64 'Graphic centered on output
      special.aspect = 128 'correction X-Y-aspect
      special.density1 = 256 'Position 1 (lower)
      special.density2 = 512 'Position 2
      special.density3 = 768 'Position 3
      special.density4 = 1024 'Position 4
      special.density5 = 1280 'Position 5
      special.density6 = 1536 'Position 6
      special.density7 = 1792 'Position 7 (high)
      special.noformfeed= 2048 'no formfeed
      special.trustme = 4096 'no Reset output
      special.noprint = 8096 'calculation only, no print
      Hardcopy (special.center + special.density4), 100&,
100&
      'for Black/white printer, black and white screen
      PALETTE 0,1,1,1
      PALETTE 1,0,0,0
      Hardcopy (special.aspect + special.fullcols +
special.fullrows), 0&, 0&
      END
SUB Hardcopy (flags, x&, y&) STATIC
  SHARED prt.DRPReq&

```

```

OpenPrinter¶
¶
POKEL prt.DRPReq& + 52, x&¶
POKEL prt.DRPReq& + 56, y&¶
POKEW prt.DRPReq& + 60, flags¶
InitDRPReq¶
¶
PrtErr% = DoIO& (prt.DRPReq&)¶
¶
PrtErr$ (0) = "NO ERROR."¶
PrtErr$ (1) = "PRINTING STOPPED BY USER."¶
PrtErr$ (2) = "PRINTER CANNOT PRINT GRAPHICS."¶
PrtErr$ (3) = "./"¶
PrtErr$ (4) = "PRINT SIZE IMPOSSIBLE"¶
PrtErr$ (5) = "./"¶
PrtErr$ (6) = "NO MEMORY FOR INTERNAL VARIABLES."¶
PrtErr$ (7) = "NO MEMORY FOR PRINTER BUFFER."¶
¶
result$ = PrtErr$ (PrtErr%)¶
¶
PRINT result$¶
¶
ClosePrinter¶
END SUB¶
SUB OpenPrinter STATIC¶
  SHARED mem.chunk&¶
  SHARED prt.DRPReq&¶
  SHARED prt.port&¶
  ¶
  mem.clear& = 2^16 'Clear memory for task¶
  mem.DRPReq% = 62 '62 Bytes, DumpRastport Structure¶
  mem.port% = 38 '38 Bytes for Port-Structure¶
  mem.label% = 4 '4 Bytes for Organization¶
  mem.size% = mem.DRPReq% + mem.port% + mem.label%¶
  ¶
  mem.chunk& = AllocMem& (mem.size%, mem.clear&)¶
  IF mem.chunk& = NULL THEN ¶
    ERROR 7 'OUT OF MEMORY ERROR¶
  END IF¶
  ¶
  prt.label& = mem.chunk&¶
  prt.DRPReq& = mem.chunk& + mem.label%¶
  prt.port& = mem.chunk& + mem.label% + mem.DRPReq%¶
  prt.name$ = "printer.device" + CHR$(0)¶
  ¶
  POKEL prt.label&, mem.size% 'allocate memory size¶
  ¶
  status% = OpenDevice% (SADD(prt.name$), 0,
prt.DRPReq&, 0)¶
  IF status% <> NULL THEN¶
    PRINT "Printer is not free."¶
    CALL FreeMem (mem.chunk&, mem.size%)¶
  EXIT SUB¶
  END IF¶
END SUB¶
SUB InitDRPReq STATIC¶

```

```

SHARED prt.DRPReq&¶
SHARED prt.port&¶
SHARED p.sigBit%¶
¶
w.window&      = WINDOW(7)¶
w.rastport&    = PEEKL (w.window& + 50)¶
w.width%       = PEEKW (w.window& + 112)¶
w.height%      = PEEKW (w.window& + 114)¶
w.screen&      = PEEKL (w.window& + 46)¶
w.viewport&    = w.screen& + 44¶
w.colormap&    = PEEKL (w.viewport& + 4)¶
w.vp.modi%     = PEEKW (w.viewport& + 32)¶
¶
p.sigBit% = AllocSignal%(-1)¶
IF p.sigBit% = -1 THEN¶
PRINT "No Signalbit free!"¶
CALL FreeMem(p.io&,100)¶
EXIT SUB¶
END IF¶
p.sigTask& = FindTask&(0)¶
¶
POKE  prt.port&+8,4¶
POKEL prt.port&+10,prt.port&+34¶
POKE  prt.port&+15,p.sigBit%¶
POKEL prt.port&+16,p.sigTask&¶
POKEL prt.port&+20,prt.port&+24¶
POKEL prt.port&+28,prt.port&+20¶
POKE  prt.port&+34,ASC("P")¶
POKE  prt.port&+35,ASC("R")¶
POKE  prt.port&+36,ASC("T")¶
¶
CALL AddPort (prt.port&)¶
¶
POKE  prt.DRPReq& + 8, 5¶
POKEL prt.DRPReq& + 14, prt.port& ¶
POKEW prt.DRPReq& + 28, 11¶
POKEL prt.DRPReq& + 32, w.rastport&¶
POKEL prt.DRPReq& + 36, w.colormap&¶
POKEL prt.DRPReq& + 40, w.vp.modi%¶
POKEW prt.DRPReq& + 48, w.width%¶
POKEW prt.DRPReq& + 50, w.height%¶
¶
IF PEEKL (prt.DRPReq& + 52) = 0 THEN¶
POKEL prt.DRPReq& + 52, x&¶
END IF¶
¶
IF PEEKL (prt.DRPReq& + 56) = 0 THEN¶
POKEL prt.DRPReq& + 56, y&¶
END IF ¶
END SUB¶
¶
SUB ClosePrinter STATIC¶
SHARED mem.chunk&¶
SHARED prt.port&¶
SHARED p.sigBit%¶
¶

```

```

mem.size% = PEEKL (mem.chunk&) ¶
prt.DRPReq& = mem.chunk& + 4¶
CALL CloseDevice (prt.DRPReq&)¶
CALL RemPort (prt.port&)¶
CALL FreeSignal (p.sigBit&)¶
CALL FreeMem (mem.chunk&, mem.size&)¶
END SUB¶

```

### Variables

PrtErr%	Error number of I/O procedure
PrtErr\$()	Error message text
result\$	Current error message

### Program description

As you may have already noticed, this program contains the subprograms `OpenPrinter` and `ClosePrinter` that were described in the program in Section 3.6. The subs `Hardcopy` and `InitDRPReq` are new material. The `Hardcopy` subprogram should be highly valuable to the user. It ensures that the contents of the current BASIC window transfers to the printer as graphics, then it calls the other subprograms.

The printer must be open before it can print a graphic screen. The `OpenPrinter` subprogram opens the printer, similar to its task in the program in Section 3.6. The program `POKEs` the width and the height of the picture to be printed into the `DumpRastPort` request structure. The same thing happens with the special bits.

The program then calls `InitDRPReq`. This routine fills the rest of the structure with the standard values, and then turns to the BASIC window.

When the time is right, the `Exec` function `DoIO&` sends the `IORequest` structure to the printer. If the printing stops, or if the command cannot be executed for any reason, this function returns an error code to the `Status%` variable. The program converts this error code into readable text and displays this text on the screen. The `ClosePrinter` routine closes off access to the printer, and the program ends.

The `Hardcopy` function is unusually versatile. It makes use of all the capabilities that the printer device has to offer. The call of the sub-programs can look something like the sequence which follows below:

```

Hardcopy flags, width&, height&
  flags: special flags
  height: height of the print out
  width: width of the print out

```



**Flags**

- `special.nothing`  
The printout occurs without any special printing effects.
- `special.milcols`  
The routine supplies the printed width in 1/1000 inch increments instead of in points (1 inch equals approximately 2.5 cm).
- `Hardcopy special.milcols, 9000, 400`  
This call prints a graphic set at the size specified in the arguments. For example, the above sample command defaults to a width of nine inches (22.5 cm) and a height of 400 printed points.
- `special.milrows`  
Similar to `special.milcols`, but this command controls printable height.
- `special.fullcols`  
The printable width comes out as wide as the hardware can manage, regardless of the value given as an argument.
- `special.fullrows`  
Similar to `special.fullcols`, but this command controls printable height.
- `special.fraccols`  
The given width is interpreted as  $x/65535$ ths of the maximum width.
- `special.fracrows`  
Similar to `special.fraccols`. The given width is interpreted as  $x/65535$ ths of the maximum width.
- `special.center`  
The program prints the graphic centered on the page. The `special.center` flag ignores any previously specified parameters setting printable dimensions.
- `special.aspect`  
This flag maintains the ratio between height and width, regardless of the changes in height or width assigned by the user.
- `special.density1-7 (V1.3)`  
Print density:      1 = low (default)  
                              7 = high
- `special.noformfeed (V1.3)`  
Disables paper formfeed, useful when printing to laser printers. This allows the user to integrate text and graphics.

`special.trustme`

No reset is sent to the printer.

`special.noprint (v1.3)`

Processes all descriptions and computes all printing dimensions without executing a printout. This command allows the user to double-check printing parameters before doing an actual hardcopy.

# **11**

# **Hardware hacking**



---

# 11. Hardware hacking

Why do you spend so much time with the Amiga? It has the software and hardware that make it a quality computer. This chapter discusses hardware and some of the neat things you can do using Amiga hardware. You'll even learn some techniques you can use to upgrade your Amiga hardware.

Before we continue, we need to touch on a few points of information about your hardware:

1. This is not a course in electronic circuitry, and it was never intended to be. We assume you have some knowledge of electronics, components and circuitry. We also assume that you have some experience operating a soldering iron, and that you know how to use a screwdriver. If you don't possess this knowledge and experience, get it before you start tearing your Amiga apart. If you still aren't sure of what you're doing to the circuitry, **DON'T DO IT!** Get someone knowledgeable in electronics. One wrong solder joint could ruin your Amiga.
2. You void your warranty (if it's still in effect) if you open your case. Any user-implemented hardware changes to a device violates the warranty. This means that the dealer or manufacturer is under no obligation to repair the machine at their own expense. In short, if you break it, you'll probably end up paying to have it fixed, even if the warranty is in effect.
3. All changes described here were tested by us as explained at the beginning of this book. It isn't always possible for an author to test every version of a computer on the market, so we may have come up wrong on one or two of these things. If problems crop up, even though the project should theoretically work, change it back to the way it was!
4. **Use caution whenever working with electronic components.** Always have the power switched off when performing any electrical work (unplug the equipment just to be extra-safe). This is for your sake as well as the sake of the Amiga. Remove the components carefully, solder or connect carefully, reconnect components carefully.

5. **Most importantly, you don't have to do any of this.** We, as developers, had to at least try these things. You, as a user, don't need to try any of these hardware modifications. If you've read the last four warnings, and still feel willing to experiment on your own hardware, fine.

Now that we're done with the warnings, let's take a look at the inside of your computer. We'll look at the memory expansion first. With a few small changes, you can configure the memory to not interrupt any programs. The next section goes into detail about disk drives. There will be occasions when you want the disk drive turned off immediately, and this hardware enhancement will show you how. The next item is a real treat—you'll learn how to outfit your Amiga with a 68010 processor. We'll also talk about other processors.

## 11.1 Disabling memory expansion

Memory expansion offers many advantages. We often get angry at our Amigas because this additional memory is incompatible with many programs. In actuality the problem lies with the programs, not with the expansion. The programs can't tell which memory to use, so they don't work. For example, the sound chip and the graphics chip must access chip RAM, if a program uses them to access fast RAM the program crashes as a result. Many early Amiga programs had this problem.

The next two hardware tricks are based on this problem. There is an alternate software solution (see the *Amiga System Programmer's Guide* from Abacus for a program which disables fast RAM through software). It's easier to turn the Amiga on, flip the switch and run a non-fast-RAM Amiga.

---

### 11.1.1 The 2000A board

If you have an Amiga 2000, you should first establish whether you have the A board (this subsection applies only to this board). If you do not have the A board, skip this section. Look on the circuit board for a PAL chip with the label U3 (you'll also find U1 and U6 labels, but for now these aren't of any interest to us). This PAL chip handles free memory organization. All we have to do is tell this chip not to make the expanded memory available. The memory release control travels over the pins named -OVR (position 19) and -SELECT (position 17).

To disable expanded memory, you must ensure that these two pins are disconnected. If no current flows between these two pins, memory expansion remains disabled. There are two ways to do this:

The first and simplest consists of just breaking the connection between the PAL chip pins and the system. Turn off your Amiga. Carefully disconnect the conducting paths from these pins. Rig a double-pole switch between chip legs 19 and 17 and their connections (you may want to use solderless connectors for the pin legs and connections). Use enough wire to run the switch outside the case with a little slack. Drill a hole in the side of your Amiga case and install the switch in the case. When you turn the switch "on," the system recognizes the expanded RAM. However, when you turn the computer off for five to ten seconds, turn this switch "off" during that time and turn the computer

back on, the Amiga won't recognize the memory expansion. All the programs that wouldn't run under expanded memory now run without problem.

The second method is somewhat neater, but also more expensive and more difficult to implement. You'll need the following materials and tools:

**Materials:** 1 DPST (double-pole, single-throw) switch  
approx. 12-16 inches double-strand wire  
1 base (20 pole, to fit the PAL chip)  
solder

**Tools:** soldering iron  
sharp knife or screwdriver  
tweezers

Carefully remove the chip. Take the 20-pin base and cut or snap off the two corresponding pins (position 19 and position 17). Insert your modified base in the old mounting. Now reconnect the removed pins to the mounting using a two-pole switch and some wire. Drill a hole in the side of your Amiga case which allows the switch head to fit through. Make sure the switch is firmly connected. Make sure all solder connections are tight and "clean." Now insert the PAL chip in its new mounting.

Make sure all connections are right; correct any problems that you detect. Then do a test run of the switch before reassembling the Amiga case.

---

### 11.1.2 The Amiga 500: printed circuit board

The Amiga 500 board has a completely different design. Our goal here is to disable the 512K card available from Commodore-Amiga. This expansion card has a battery operated clock and fast RAM. This clock remains undisturbed by the following operations.

**Materials:** 1 SPST (single-pole, single throw) switch  
approx. 12-16" double-strand wire  
solder

**Tools:** soldering iron  
sharp knife or screwdriver

Turn off your Amiga and open the expansion "drawer." Remove the expansion card carefully (remember—use caution when removing, modifying and installing any parts). After you remove the card lay it out on the table in front of you, trace side (the side with all the etched



connections and solder joints) facing you. You should be able to see the solder joints and traces, and half of the edge card which plugs into the Amiga's expansion port.

Look at pin 32 of the edge card (make absolutely sure that this is pin 32). Follow its route up the printed circuit board. See how the trace (the etching) moves away from the solder point? That's our goal. You must somehow break the conducting path of this trace. Take a sharp screwdriver or sharp knife (an XActo® knife or sharp kitchen knife will work). Carve into the trace to create a space—make sure there's a definite break between the cut (you should be able to see the printed circuit board material through the cut, with no tracing material connecting). You may want to make the space of the cut fairly wide (e.g., 1/8"). Take a piece of two-lead wire and solder each lead at one end to the cut sections of the now-broken trace. Solder the other two ends to the SPST switch. That's all there is to it.

Re-install the expansion card carefully. Make sure all solder connections are tight and "clean." Connect everything up and boot the Workbench. When you have the switch in the "off" position, the Amiga 500 should ignore the memory expansion. When you turn the Amiga off again for five or ten seconds, flip the RAM switch to the "on" position and turn the power switch "on," this enables the memory expansion. If the computer doesn't do what it should when it should, you must have done something wrong. Check your solder joints (cold solder joints frequently occur if you aren't careful, thus not making a tight connection). If you have cold solder joints, re-solder the connections. There really aren't any other errors that could occur.

---

## 11.2 Disk drive switching

Additional external disk drives can cause as many problems as memory expansion. External drives are usually automatically configured by the operating system, which uses your working RAM. These autoconfig systems may cause problems because AmigaDOS can only manage data from the disk drive using RAM chips.

Each drive requires a 30K buffer for file management. Many programs need this memory, but once it's allocated for disk buffers, programs can't access that extra memory. The result: The program either crashes in mid-run or can't be started at all.

### *On/off switch for disk drives*

One solution is to install a switch to disable the drive as needed. You'll need the following equipment:

#### *Materials:*

1 SPST (single-pole, single-throw) switch  
approx. 4-8" single-strand wire  
solder

#### *Tools:*

soldering iron  
sharp knife  
screwdriver

You can easily install a switch if the external drive doesn't have one of its own. The switch interrupts the data direction of the computer by resetting the line which informs the Amiga that another drive is connected. Pin 21 (SEL1) of the drive plug handles the selection of the first external drive.

Turn off the Amiga and unplug it for safety's sake. Determine the correct lead for pin 21 and cut the wire. Solder the cut ends to one end of each strand of wire leading to the switch. Solder the ends of the switch wires to the SPST switch. You can either do this by connecting it at the plug itself, or within the disk drive. If you selected the latter, drill a hole in the disk drive case to match the switch. Mount the switch, reassemble everything carefully and test out the computer.

---

## 11.3 Installing a 68010

Would you like to make your Amiga faster without spending a lot of cash? Here's your chance. All you have to do is remove the old 68000 and replace it with 68010. This new Motorola chip is 99.99% compatible with the old chip. It has only one disadvantage but it's only a minor disadvantage.

You can install the new processor easily. No soldering (in most cases), no additional expensive components. The 68010 shows speed increases in certain processor commands of up to 80% in tests. If you look at this from a general standpoint, the program uses faster commands as well as those that already exist. All in all, the speed only increases by about 16% over the 68000, but a faster machine is a faster machine.

In addition, you have the option of making the new chip 100% compatible using an additional program (which you'll find at the end of this section).

Now let's see to the installation of your new processor. First you must buy a 68010 processor. That shouldn't present a problem: You can find ads for this chip in classified sections of computer magazines, and in any computer journal that deals almost exclusively with sales of components (e.g., *Computer Shopper*). Or perhaps you have an electronics shop in your neighborhood that has the chip available or can order it for you. Once you have the new processor, you can continue with the installation.

### *Getting started*

First you must open your Amiga case. This takes various amounts of time, depending on the type of Amiga you own. Just take your time dismantling the case, and pay attention to the order in which you take things apart. You'll need to know the order so that it'll be easier putting it back together. You should mark each piece, possibly with masking tape, as you go along.

The 68000 main processor should be easy to find. It's the largest chip on the main printed circuit board. It's probably labelled with "68000" or something similar. You must remove this chip.

However, before removing the main processor, we must mention something important. First, many of the 68000 chips were merely inserted in a chip socket. However, there may still be a few Amiga motherboards which have 68000s with soldered connections. If you're presently looking at a soldered processor, there are only two answers: Either you desolder the chip from the circuit board or have someone who has soldering experience to do it for you. If you don't have the soldering experience, you could mess it up badly.

The best solution is to use a solder plate (which makes all of the pins hot at the same time) so that the chip can be pulled out as one unit during the desoldering. Solder in an equivalent chip socket to ease chip replacement.

Let's assume for now that you have a socketed 68000. The first step is to remove this from the socket. You can do this using one or two tools: Special tweezers designed for the purpose of removing a chip level (all pins at once); or a screwdriver. The tweezers are expensive, and are only worth the purchase if you want to save the chip for later use (or if you are afraid of injuring the chip).

A flat screwdriver is a little riskier, but achieves the same result. Insert the blade of the screwdriver flat between the socket and the end of the chip. Rotate the blade gently about 10 degrees or so to pull the processor up from the socket. Repeat the same procedure on the other side. Keep moving from end to end, prying the chip up bit by bit. Before removing the processor completely, note the direction at which the notch of the old processor points. Remove the chip by hand (remember the direction the notch pointed—it's important). This method will work for removing almost any chip, particularly those chips with large numbers of legs.

Before you go on to the next step, we have a warning for you: Electronic components are extremely delicate. The slightest difference in voltage, say from a static charge, can "fry" a chip (render it useless). That's why you should always ground yourself before you handle the chassis or chips.

Next you should insert the new processor. Remove the 68010 from its packaging and place it on the socket from which you removed the 68000. Press down on the chip gently and evenly. Continue this gentle, even pressure until the bottom of the chip is flush with the top of the socket.

Now reassemble the Amiga. Make sure that all parts are accounted for (screws, washers, etc.)—you should have all the parts you removed. Take care that all fasteners are connected properly. Congratulations! You've just replaced the main processor. Now comes the power-on test. Plug in the Amiga and check all power connections. Turn it on. Everything should carry on as normal, except you should notice an increase in speed.

If something's wrong, this may be for one of two reasons:

1. There may be an improperly connected cable or chip pin. Check this first, before anything else.

2. Your clothing may have contained a static charge and touched the chip. As we mentioned above, chips aren't built to tolerate static electricity. If you touched a pin of the chip with your finger and your body contained a static charge, you may have destroyed your new processor. If nothing helps, you'll have to buy a new 68010 to test it. Try replacing the old 68000 to see if the entire Amiga is defective.

The 68010 has additional debugging instructions, which must be enabled by software on the Amiga. Programs that start with exception 4 will crash on the 68010, unless these changes are done. Fred Fish disk number 18 contains the program DeciGEL which does all of the setup work for you. The SetAlert command performs the same task from the Startup-sequence (you'll find SetAlert on the new Workbench 1.3 disk). *AssemPro* from Abacus also has a program to do this, along with the source code.

---

## 11.4 The roar of the fans

Do you have an Amiga 2000? In the beginning, we thought the noise the fan made was a show of quality. After a while the fan noise got pretty annoying.

We offer you two options. The first suggestion came from a TV repairman, who advised that we decrease the amount of power running the fan. We didn't feel that was such good advice, so we chose a more elegant solution.

The built-in Amiga 2000 fan is a Papst Multi-Fan 8312M. The M at the end of the number states the amount of noise it makes. After searching through merchant information, we found a similar model that performs the same task with half the noise level—the Papst Multi-Fan 8312L.

The hardest part of installing this model is finding it. Once you do, all the screws and connections are the same as the original equipment. One disadvantage of the entire process is the price of the new fan—about \$50. Once you've recovered from the shock, remember that the fan will have a long, quiet life.

For those who think that a new fan is too expensive, we recommend the method described by the TV repairman mentioned above. He suggested we cut the positive power connection to the fan and insert a 50 $\Omega$ , 5W potentiometer (the adjustable range should be between 0 $\Omega$  and 100 $\Omega$ ). By turning the potentiometer down the noise gets lower and softer. Make your judgements by the amount of heat accumulation your Amiga has, rather than fan speed (remember that the degree of heat increases with expansion cards).

To conclude this section, we leave you with a warning. The Amiga fan makes so much noise because of the potential amount of hardware it must ventilate. The developers of the Amiga assumed that every free expansion slot had a card plugged into it. If this is the case, then the fan should continue to run at full speed. Otherwise, feel free to experiment with running the fan at lower and quieter speeds.

---

## 11.5 New processor information

Half of the information about new processors usually ends up in technical journals. What can these new processors do for us, exactly? To answer this question, we should take a closer look at these processors.

---

### 11.5.1 The 68010: high power, low price

The 68010 is fully compatible with the 68000 command set. All 68000 commands are integrated into the 68010. These commands execute more quickly than in the 68000. In addition, the 68010 features four new commands consisting of a loop mode and three other registers. The amazing part of this chip is its easy replacement over the 68000: Low price, identical size and pinout to the 68000 (see Section 4.3 for installation information).

We should discuss the 68010's architecture. Every assembler supports 68010 programming. Three new registers exist in this chip that the 68000 didn't have: the `SourceFunctionCodeRegister` (SFC), the `DestinationFunctionCodeRegister` (DFC) and the `VectorBaseRegister` (VBR). The last register allows the examination of the beginning of the system vector table (between \$0 and \$3FF on the 68000). This value changes to \$0 after every reset. In addition, it can be very useful to change all vectors simply by switching over.

The Code register consists of only three bits and offers access to `Read(SFC)` and `Write(DFC)` just like User and Supervisor modes. Some news for the hardware hobbyist: When you connect pins FC0-FC2 to the address bus, four memory banks accommodate 16 megabytes. The operating system rewrite forces separations in user data/user program and supervisor data/supervisor environments.

Another difference from the 68000 lies in the 68010's loop mode. Prefetch technology makes this possible by reading a command while the processor retains the previous command. The 68000 reads the following loop from the address bus 75,000 times and accesses the address bus 75,000 times. The 68010 performs this loop only three times instead of 75,000 times:

```
MOVE.W #2499,D0
Loop: MOVE.L (A0)+, (A1)+
DBRA D0,Loop
```

The increase in speed should be evident to you. Unfortunately, only the following machine language instructions function in loop mode:

```
ABCD, ADD, ADDA, ADDX, AND, ASL, ASR, CLR, CMP, CMPA,
EOR, LSL, LSR, MOVE, NBCD, NEG, NEGX, NOT, OR, ROL, ROR,
ROXL, ROXR, SBCD, SUB, SUBA, SUBX, TST
```

The exceptions look somewhat different on the 68010 because more data is needed on the supervisor stack. The last data corresponds to that of the 68000 on the supervisor stack, so the major difference lies in the 68010's larger stack requirements. Bit 15 of the status word is interesting in this context, it tells if the processor executes the exception (0) or ignores it and executes the next command (1). This means that bus and address errors can be trapped using software. This will solve or prevent several Guru problems.

The new commands are called MOVEC, MOVE CCR, MOVES and RTD. The command MOVE SR, Destination only operates in supervisor mode, causing a Guru Meditation. You can program an equivalent exception routine which bypasses this problem. Few programmers are unaware of this problem. Here are the syntaxes of the four instructions (alternate syntaxes are given as needed):

```
MOVEC Register, Destination
```

```
MOVEC Source, Register
```

One of the three new registers or the USP can be substituted here for the Source and Destination arguments. Data size: Word.

```
MOVES Register, Destination
```

```
MOVES Source, Register
```

MOVES transfers data between four data banks according to the methods described above. It serves no purpose in major hardware manipulation on the Amiga.

```
MOVE CCR, Destination
```

MOVE CCR reads the status register.

```
RTD Value
```

RTD is the equivalent of RTS. This instruction adds the value (16 bits) to the stack pointer. This is practical when using the stack as a parameter statement.



## 11.5.2 The 68012: low cost, high memory

The mere size of the 68012 is the first thing the user notices about the chip. It has a square instead of a rectangular shape, so you can't just plug it into the 68000 socket. The user can take one of two routes to install this chip:

- Install a second socket.
- Buy an adapter board for the 68012.

This 100% 68010 compatible chip allows up to 2 gigabytes of working RAM. The first gigabyte lies in memory locations \$0 to \$3FFFFFFF and the second gigabyte lies in memory locations \$80000000-\$BFFFFFFF. The following diagram shows the pin arrangement as seen from below:

D12	D10	D8	D7	D5	D4	D2	D1	As	A1
D15	D14	D11	D9	D6	D3	D0	UDS	LDS	DTACK
A22	A23	D13	--	--	--	--	R/W	BG	BGACK
A21	GND	GND					--	VCC	BR
A20	VCC	A2D		MOTOTOLA		GND	GND	CLK	
A19	A18	A25		68012		--	RST	HALT	
A17	A15	--		(BOTTOM)		A27	VPA	VMA	
A16	A12	A13	--	A28	A29	--	IPL1	IPL2	E
A14	A11	A10	A8	A5	A2	A31	FC1	IPL0	BERR
-	A9	A7	A6	A4	A3	A1	FC0	FC2	A26

The dashed pins are unused; the A1 pin marks the upper right of the chip.

## 11.5.3 Monster processors: 68020, 68030, 6888x

### 68020

Information about the 68020 alone can fill volumes. Here are a few key points about this chip:

- 32-bit address bus: This bus enables direct addressing of 4,294,697,296 bytes (about 4 gigabytes). Pin A0 allows access to odd addresses (the 68000 could only do this by means of elaborate calculations).
- Dynamic bus structure: Allows switching between 8-, 16-, 24- and 32-bit data buses.

- True 64-bit arithmetic.
- 62 addressing types (50 of them different).
- Access to individual bits or bit fields.
- 28 additional instructions.
- Data types: bits, bytes, words, longs, packed BCD numbers, unpacked BCD numbers and bit fields.
- Acceptance of internal instructions: three-word prefetch.
- Processor internal instruction memory: 256-byte cache.
- Coprocessor interface and coprocessor instructions.
- Frequency measurement: standard = 16 MHz, others = 24 MHz.
- Three stack pointers: `MasterSP`, `InterruptSP`, `USerSP`.
- Two-cache register and extended SR.

**68030/68851** The 68030 processor surpasses all of this data three to four times over, and still remains compatible. It has 31 registers available for reading and writing. The 68851 coprocessor closely integrates with the 68030, and already runs in hardware-multitasking mode.

**68881** Real power comes into play when a floating point arithmetic coprocessor supplies math calculations directly to the 32/64-bit processors. The 68881 processor operates using eight floating point registers, and can process the following operand sizes:

Byte (8-bit)  
 Word (16-bit)  
 Long (32-bit)  
 Float (32-bit)  
 DoubleFloat (64-bit)  
 ExtendedFloat (96-bit)  
 BinaryCodedDecimal (96-bit)

The math commands encompass any calculations you can think of, including three different logarithms and everything that ever existed in all the Amiga math libraries put together. It is convenient to use general IEEE floating point format, so that a conversion occurs.

**68882** The 68882 is an extension of the 68881 processor.

**12**

**Hints and tips**



---

## 12. Hints and tips

This chapter contains many small hints and bits of information that could be thought of as “mini tricks and tips.” These include information specific to both Workbench 1.3 and 2.0. These tricks and tips will hopefully make your sessions with your Amiga more productive and more enjoyable.

---

### 12.1 Tips for the Shell

**#1:**  
*Automatic  
backups*

Have you ever edited your Startup-sequence using ED, quit ED, and then realized that your Amiga won't boot without the line you just deleted from the Startup-sequence? Don't panic. ED always creates an automatic backup of the last file edited, and places this file in the t : (temporary) directory. The t : directory keeps temporary files available on disk just in case you destroy the file you're currently editing.

To restore the old startup-sequence file, copy the file ed-backup located in the t : directory to the s : directory as the startup-sequence. The Shell command sequence is as follows:

```
copy t/ed-backup to s/startup-sequence
```

**Note:**

Do not copy the t : directory to the RAM disk. If you do, the t : directory will be deleted after a reset and you'll be unable to access it after a crash.

**#2:**  
*Interrupting  
Shell output*

You can pause text scrolling in the Shell window (i.e., whenever you execute the list, dir or type commands) by pressing any character key. The CON handler stops the text display until you remove this character by pressing the **Backspace** key. The output resumes once you press **Backspace**. You can easily pause and restart a disk directory display by pressing **Spacebar** and **Backspace**.

**#3:**  
*New Shell  
text modes*

You may have seen some disks which displayed text in different colors and type styles during booting. There's no magic here: You can change colors without any problem. Also, you can display text in italic, bold or underlined text. The normal Echo command permits these changes.

To use these text features, you need control characters to inform the computer that the next characters are escape sequences and must be executed rather than printed. These command characters are enclosed in quotation marks and always begin with the sequence “\*e”. The “\*e”

combination represents the **(Esc)** key and signifies an escape sequence. Characters and numbers follow this initialization. Some of these characters are separated from one another by semicolons, and comprise the control sequence itself. For example, the number "4" enables underlining and the number "42" represents a black background color. For available styles, refer to the values in the following tables. The control sequence concludes with "m" and the text you want displayed.

Try the following: Create a file named UNDERLINE using ED or another editor. Enter the following in this file:

```
echo"*e[4mUNDERLINE on"
echo"*e[0mNormal"
```

Save the file and exit the editor. Now enter Execute Underline and press **(←)** to see the results:

Underline  
Normal

The following list documents control sequence values:

Typestyle	Number	Remarks
normal	0	
bold	1	
italic	3	
underline	4	
inverse	7	

Foreground color	Number	Remarks
normal	30	set using Preferences
white	31	
black	32	
orange	33	

Background color	Number	Remarks
normal	40	set using Preferences
white	41	
black	42	
orange	43	

**#4:**  
**Ctrl** key  
 combinations

Key combinations can accomplish many things. Most of these key combinations are actuated in conjunction with the **Ctrl** key. When you press and hold the **Ctrl** key then press another key, the combination usually affects control of a screen or program. This combination usually bears the name *control key*.

The first four control keys allow you to stop the execution of many programs.

<b>Ctrl</b> <b>C</b>	stops an AmigaDOS command
<b>Ctrl</b> <b>D</b>	stops a running script file
<b>Ctrl</b> <b>E</b>	executes a break of higher priority
<b>Ctrl</b> <b>F</b>	executes a break of higher priority

Here are a few of the most used screen control combinations:

<b>Ctrl</b> <b>G</b>	screen flash (without signal tone)
<b>Ctrl</b> <b>H</b>	same as the <b>Del</b> key
<b>Ctrl</b> <b>J</b>	same as the <b>Tab</b> key
<b>Ctrl</b> <b>K</b>	same as <b>↑</b> (cursor up)
<b>Ctrl</b> <b>L</b>	erases the window (same as <b>Esc</b> <b>C</b> )
<b>Ctrl</b> <b>M</b>	same as the <b>←</b> key
<b>Ctrl</b> <b>N</b>	activates a new (an alternative) character set
<b>Ctrl</b> <b>O</b>	restores the original character set (same as <b>←</b> )
<b>Ctrl</b> <b>X</b>	deletes the contents of the current line

**#5:**  
**Copy more!**

The **Copy** command copies files from within the Shell. There are many variants to this command that are poorly documented (if any documentation exists at all). One application of **Copy** allows you to copy several files that have nothing to do with one another. These files cannot be grouped together using the wildcard characters (\* and ?).

The solution is very simple: You can copy files with the same names but different extensions using the bar character (|). The following example copies three files of the same name (**test.c**, **test.h** and **test.o**) to the RAM disk:

```
Copy :test.(C|H|O) to RAM:
```

We can use this in a different way by viewing the entire filename as an extension and creating a multiple copy from that. This avoids loading the **Copy** command for each copy. The following example copies the three Shell commands **Dir**, **List** and **Rename** from the **c:** directory of the current disk to the **c:** directory of the RAM disk:

```
Copy C:(DIR|LIST|RENAME) to RAM:C
```

**#6:**  
**Shell**  
 windows and  
 script files

You can supply your own dimensions and title when opening a new Shell window. In addition, you can assign the new Shell task to execute a script file. That looks like this:

```
NewShell Outputdevice: Script-file
```

In Version 1.2 the device for the `Outputdevice` argument should be `CON:` with its window size arguments. The script file executes after the `NewShell` opens. You can execute a script file in parallel with another process in another window.

Workbench 1.3 owners should either use `NewCon:` as the `Outputdevice` argument (this keeps all the `Shell`'s editing features resident in the new window), or `NewShell` without an output device.

Workbench 2.0 owners should either use `Con:` as the `Outputdevice` argument which keeps all the `Shell`'s editing features resident in the new window, or `NewShell` without an output device.

**#7:**  
*Disk icons*

Workbench 1.3 has a slightly different look from the previous version. All old devices are represented by their old icons. The RAM disk and the new RAMBO disk still use the old plain disk icon. The icon used by the Workbench disk can also be used by the RAM and RAMBO disks by adding two small `Copy` commands in the Startup-sequence. The following lines perform this task:

```
Copy Disk.info to RAM:
Copy Disk.info to RAD:
```

Place copy commands before the `LoadWB` command and you will have a uniform icon design. This method can be applied to any other disk as well.

**#8:**  
*One window,  
two Shells*

You cannot have two `Shells` operating at once in two windows. However, it's possible to have two `Shells` in one window. This is simply done with the following command sequence:

```
NewShell *
```

This command redirects the `NewShell` output to the present window, without opening a second window. You alternate between the first and second task. This saves you the somewhat complicated trouble of switching windows. If you do this, use `Run` to execute a program instead of `Execute` for a direct program call.

**#9:**  
*Borderless  
Shell*

There are programs that try to eliminate the border of the `Shell` window by using complicated methods. Many of the developers of these programs seem to have forgotten that the `Shell` is created using the `Console` device. The `Console` device contains commands which let you change the border's appearance using escape sequences. These sequences let you do just about everything with the size and appearance of the window:



Sequence	Explanation
<Esc> n u	sets the window width at n characters
<Esc> n x	sets the left border at n pixels
<Esc> n y	sets the upper border at n pixels
<Esc> n t	sets the number of lines at n
<Esc> c	sets everything back to normal

The Echo command is used to call each sequence. It is also possible to link multiple sequences using semicolons. The window must be resized after **(Esc) (C)** to display the border. Here is an example:

```
;This sequence configures for border OFF
echo "*e[80u*e[0x*e[0y*e[31t"
;Press <Ctrl><L><Return> to actuate borderless window

;This sequence returns system to normal mode
echo "*ec"
```

### #10: *Quick messages*

Usually the Startup-sequence displays messages about the Workbench version number and the current date. If you wish, you can add your own messages to this data.

The problem in doing so is that when the Echo command displays text, the command must be reloaded every time, which is quite time consuming. You can speed this up by writing the entire text to a file and using the Amiga's multi-tasking capability to display the new text on the screen during the Startup-sequence.

Enter the following line in your Startup-sequence file to call the text file that you want displayed:

```
Run Type Textfile
```

## 12.2 Tips for AmigaBASIC

### #11: *Closing with standard icons*

AmigaBASIC uses one particular icon for all the programs created from AmigaBASIC, as well as files created from AmigaBASIC programs. There are two ways to change this:

The first deals with the BASIC program icon. You can edit this icon using the icon editor after you finish developing the program. Assign this new icon to the finished program and copy it into the correct directory. The trick is to set the delete flag in the info file so that the info file cannot be deleted. This ensures that the icon doesn't get overwritten the next time you save the program.

There is a simpler method when it comes to data files. These data files should have an icon relating to the program (e.g., text files written in Notepad have a note icon). First you draw the desired icon with the icon editor. This icon is stored in the same directory as the main program. Then you read the icon at the beginning using the function `GetDiskObject`. Every time you save a file this icon gets saved under the same name. The result: Your icon replaces the AmigaBASIC icon. Here is an example program that gives a BASIC program the Shell icon, you will have to change the drawer and disk names (T&T2:Chapter1) for your own setup:

#### *Program start:*

```
REM IconInstall in Chapter1 drawer on disk named T&T2
LIBRARY "T&T2:bmaps/icon.library" :REM use ConvertFD
DECLARE FUNCTION GetDiskObject& LIBRARY
DECLARE FUNCTION PutDiskObject& LIBRARY
FileName$ = "SYS:Shell"+CHR$(0)
DiskAdr& = GetDiskObject&(SADD(FileName$))
File$ = "T&T2:Chapter1/IconInstall"
File$ = File$+CHR$(0)
status = PutDiskObject&(SADD(File$), DiskAdr&
```

#### *After closing file:*

### #12: *Modular work*

You should try to use modular program structure whenever possible. This makes the listing easier to follow and modify. Another advantage is that many of these modules can be merged into later programs that you write. This saves a lot of work. You have two possibilities which you can apply to merge modules into other listings:

First, load the BASIC interpreter and copy the module from the source program into the Clipboard. This Clipboard can then be pasted into a second BASIC program. Load the second program and paste this stored section back into the listing using the `Paste` function.

The second method requires that you save the program block to a file. Be careful that you use ASCII format—you cannot use the other formats for this method. You can then merge the new block into a program with:

```
MERGE Filename
```

This saves a lot of work and allows you to build a library of independent function modules.

**#13:**  
***Changing  
window and  
screen names***

In AmigaBASIC you can specify a window name when opening a new window, but you can't change the window name later on. The Intuition library offers a solution.

Open the library at the beginning of the program. This lets you specify both window and screen names by calling `SetWindowTitle()`. The following demonstration program renames your BASIC window to `test` and your Workbench screen to `Screen`. Please notice that the variables used to pass parameters to the `CALL` must be ended with null bytes, `(Chr$(0))`.

```
REM WindowTitle
LIBRARY "T&T2:bmaps/intuition.library":'set path name for
'wherever you have intuition library stored
CALL SetTitle("test","Screen")
END
```

```
SUB SetTitle(WinNam$, ScrNam$) STATIC
WinNam$ = WinNam$ + CHR$(0)
ScrNam$ = ScrNam$ + CHR$(0)
CALL SetWindowTitles(WINDOW(7), SADD(WinNam$),
SADD(ScrNam$))
END SUB
```

**#14:**  
***More memory***

Normally `Clear` allocates more memory in AmigaBASIC. But this command often doesn't work if it can't find the desired memory (a bug in AmigaBASIC). First it tries to provide new memory and then it tries to release old memory. When you want to change your memory only a little, you must have double the amount needed available, which is often not the case. There is only one method to achieve this goal. Simply set the area to the smallest size, then enter the desired number:

```
CLEAR ,1024
CLEAR ,500000
```

We don't recommend this method when you're in program mode instead of direct mode. Use the following in program mode:

```
CLEAR ,25000-FRE(0) 'only the necessary program memory
CLEAR ,FRE(-1)-50000 'entire free memory - security
```

**#15:**  
***Editing  
 BASIC  
 programs***

The BASIC editor is sometimes a nuisance. It scrolls horizontally with difficulty and slowly scrolls up or down. In addition, you cannot search for commands because no search function exists. AmigaBASIC offers some help. You can save programs as ASCII text with `Save "name",a` and then edit the program using a word processor such as *TextPro* or *BeckerText* from Abacus. The programs can also be transferred to other computers in this manner.

Word processors allow many more editing features than the AmigaBASIC editor. Since the Amiga is multitasking, the word processor and BASIC can be running at the same time. This lets you quickly move the edited program to AmigaBASIC for easy testing.

**#16:**  
***Faster  
 AmigaBASIC***

You'll probably agree that AmigaBASIC could be faster. When you start Amiga-BASIC from the Workbench, the task priorities are not optimally set for AmigaBASIC operation.

Two functions in `Exec.library` offer a cure for this problem. `SetTaskPri()` allows you to view the priorities of your tasks and BASIC. You can choose a value from 1 to 127. The larger the number, the faster AmigaBASIC runs. That means that other programs (tasks) receive less processor time. This method works especially well for calculations requiring a large amount of time and for graphic output.

First you access `Exec.library`. Then the task is found with `FindTask()` and changed with `SetTaskPri()`. When the BASIC program ends, it's important that the priority be reset so other programs can run. Here are the program segments:

***Program start***

```
LIBRARY "T&T2:bmaps/exec.library"
DECLARE FUNCTION FindTask& LIBRARY
BASICTask& = FindTask&()
CALL SetTaskPri(BASICTask&, 80)
```

***Program end***

```
CALL SetTaskPri(BASICTask&, 0)
```

**#17:**  
***No overflow  
 in line buffer***

Do you recognize this? The AmigaBASIC editor occasionally won't let you go back over a line using the **Backspace** key. It responds with the error message "line buffer overflow". You can easily get around this. Select a space with the mouse and then cut it with `<Amiga> [X]`. Now the editor will work correctly again.

**#18:**  
***Reset***

When you would like your BASIC program to reset the Amiga, use the following code (with caution):

```
bye = 16515072
CALL bye
```

---

## 12.3 Printer tips

### **#19:** *Changing printers without Preferences*

Very few people know that the printer driver is a stand alone program and can be started by itself. It's not handled as a program in the usual sense, so you can't access it through the Shell. The Workbench knows where the driver must be placed in memory. Copy the necessary driver from the `DEVS:Printers` directory into the main directory of the Workbench disk. Then you need a `Tool` type icon like the clock or the Shell icon. Give one of these icons the same name as the printer drivers. When you click on the icon from the Workbench the corresponding printer driver initializes. Here are the Shell commands for the Epson printer driver:

```
1> copy sys:devs/printers/epson sys:epson
1> copy Shell.info Epson.info
```

### **#20:** *Indirect printing pays off*

When you would like to use the functions of your printer in a BASIC program, most people access the printer directly by using the `PAR:` or `SER: device`. Normally, all printer output should occur over the printer device `PRT:`.

The Amiga implements generic sequences to control printer effects. The corresponding printer driver converts the sequences that are necessary for that particular printer. This ensures that the print routines for one printer can run on all others as well. The following table lists the command codes that can help you to achieve all different print styles. All sequences begin with `(Esc)`. It is best to define this with a string, as in the example below:

```
ESC$=CHR$(27)
```

The above `CHR$(27)` signifies the `(Esc)` key. This character string tells the printer to prepare for special codes to control the printer or other device.

The following is a table of type styles and escape sequences:

Typestyle	Sequence
Italic	on: ESC [ 3m off: ESC [ 23m
Bold	on: ESC [ 1m off: ESC [ 22m
Underlined	on: ESC [ 4m off: ESC [ 24m
Elite	on: ESC [ 2w off: ESC [ 1w
Compressed type	on: ESC [ 4w off: ESC [ 3w
Wide type	on: ESC [ 6w off: ESC [ 5w
NLQ	on: ESC [ 2 " z off: ESC [ 1 " z
Proportional type	on: ESC [ 2p off: ESC [ 1p
Superscript	on: ESC [ 2v off: ESC [ 1v
Subscript	on: ESC [ 4v off: ESC [ 3v

A command sequence can be sent together with the text to the printer with LPRINT:

```
LPRINT ESC$;"[4m";Text$;ESC$;"[24m"
```

---

## 12.4 Amiga Hints

**#21:** *Faster, faster, faster* It helps increase the speed of the Amiga, especially during long computation times, if you close all unnecessary screens.

**#22:** *Stop!* Pressing the left mouse button stops text output. When you release the mouse button, the text output can continue. This is especially useful when text is scrolling past rapidly in some Shell commands (e.g., the `Dir`, `List` and `Type` commands).

**#23:** *DiskDoctor First aid* Have you ever deleted a file that you really didn't want deleted? You can recover the file but you must act quickly. The Amiga doesn't really delete files. It simply resets the directory entry and the bitmap of the disk.

You don't need a disk monitor to rescue the file. Instead use the `DiskDoctor` program. `DiskDoctor` replaces deleted files providing that you haven't written to the disk since you deleted the files.

**#24:** *One Guru less* If you start a program from the Workbench, it runs without hesitation. However, starting the same program from the Shell may require as much as five seconds longer to execute. Why? Many Amiga users have asked this question, so here's the answer: On a multitasking computer every program has its own stack where a return address is stored. When a program starts from the Workbench, this stack is created using the value in the `Info` file.

It's different in the Shell. Here the Shell's stack value comes into play instead of the stack value of the `info` file. This is usually rather small. Remember when you start a program from the Shell, it takes on the current Shell stack size. This may not be enough memory and you may end up getting a Guru Meditation. If that's the case you should enlarge the stack before running the program. The stack size needed for a program can be seen by selecting `Information (Info in 1.3)` in the `Icons (Workbench in 1.3)` menu.

**#25:** *More of the same* The file `System-Configuration` can be found in the `devs:` directory of any given boot disk. This file contains all the parameters set by Preferences. Copy this file onto each boot disk you have. This gives you the same Preferences setting on all your boot disks. Here's a sample Shell command:

```
1> copy sys:devs/system-configuration df1:devs
```





**13**

**Devices and the  
FastFile System**



# 13. Devices and the FastFile System

A *device* is simply a piece of hardware with which the computer can exchange information. The disk drive is a typical device.

This data exchange between computer and device doesn't always have to go in both directions. A printer only accepts data, while a mouse only sends information to the computer.

The description of the `ASSIGN` command includes a list of devices that can be accessed from AmigaDOS. The standard devices of the Amiga are listed below:

```
PIPE AUX SPEAK CON RAW  
SER PAR PRT DFO
```

A colon (:) must always follow the device name, so that AmigaDOS can tell devices apart from directories or filenames.

## ***Handlers***

Handlers are found in the `L:` directory. Handlers are treated as if they are actual physical devices even though no hardware is required for their operation. The `SPEAK:`, `PIPE:` and `AUX:` devices are handlers. Handlers must be `MOUNTED` before they can be used. This is usually done in the `Startup-sequence` or the `StartupII` script file. They must also be described in the `MountList` located in the `DEVS:` directory.

---

## 13.1 The PIPE device

The PIPE device is a member of the DEVS: directory group. To show you what the handler can do, we'll start by viewing a better-known handler—the Clipboard device. This device performs temporary data exchange. If you need to exchange data between tasks while in the Shell, you'd use the Clipboard to transfer this data. Once the Mount command places the Clipboard in the Mount list, you can direct data to and from this device by output or input.

Disk access takes quite a bit of time. By adding memory expansion (the Amiga can access 4,294,967,296 bytes [over four gigabytes] of main memory through the 68020 processor), the majority of data messages, once exchanged on the disk through the Clipboard, can also be managed in RAM.

Here's where the PIPE device comes into play. You can think of the operation as if a pipeline were placed in RAM which could be filled from one side with your data. The data could then be poured out of the pipeline to the other application when needed. To use the PIPE device the system must first be informed that the PIPE handler should be activated. Enter:

```
MOUNT PIPE:
```

In the DEVS/Mountlist file you can enter the desired size of the pipeline with the editor ED. To fill the pipeline, you only need to make sure that the NewCon device receives the pipeline data instead of the Console device. This can be done with the Shell's output redirection command (>):

```
DIR >PIPE: SYS:
```

This directs the root directory of the boot disk into the pipeline. Nothing else happens after you enter this command, aside from a brief disk access. Enter the following command sequence to empty the PIPE and display its contents on the screen:

```
TYPE PIPE:
```

This command won't work if too much data enters the pipeline, as you may have seen from the above example. Should the pipeline be too full, the error message renders the pipeline data unusable.

---

## 13.2 The Speak device

The `Say` command already existed in Version 1.2 but the `Speak` device, added in Version 1.3, takes on an entirely different quality.

This device has some similarities to the `PIPE` device. It connects into the system like `PIPE` and redirects data. Unlike `PIPE`, the `Speak` device doesn't allow temporary storage. Whatever you enter will come out of the monitor speakers as speech.

Enter the following command sequence:

```
MOUNT SPEAK:  
DIR >SPEAK: SYS:
```

Perhaps you're tired of reading stories to your children every night, and you'd like a night off from that task. Have the Amiga do it. The following command starts a task and speaks the contents of the file named `BedTimeStories`:

```
RUN TYPE >SPEAK: "BedTimeStories"
```

Speech synthesis enthusiasts should try the following with the Extras disk in `df1`:

```
RUN TYPE >SPEAK: df1:AmigaBASIC OPT H
```

It makes more sense to make a prompt audible with `Ask`:

```
ASK >SPEAK: "Do you like the Amiga?"
```

---

## 13.3 The NewCon device

The NewCon device is probably the best new command accessible from the Shell. This device is in the Startup-sequence (Mount NewCon:). NewCon is similar to the Console device that opens Shell windows. Enter a command and press the **↵** key. Now press the **←** key (left cursor key) and observe what happens on the monitor. The NewCon device has been made an integral part to Version 2.0.

You can now edit the command line at any time, similar to what occurs in the List window of AmigaBASIC. When you mistype something in a longer command line, you can now correct this error without having to retype the entire text. Now enter the following commands:

```
DIR df0: DIRS
LIST ram: OPT A
TYPE s/startup-sequence
```

Pretend that you would like to look at the directory of drive DF0:. With the Shell you would have to re-enter the Dir command. Press the cursor up key to scroll up to the previously entered commands. Because only the commands are saved, the memory requirement to do this is small. The cursor keys can be used to edit the commands.

## 13.4 The FastFileSystem

You'll probably agree that AmigaDOS requires too much disk space. Version 1.3 used the external disk media extensively. A new disk format takes advantage of these improvements.

To make you familiar with the file/system relationship, we'll show you a big difference between the versions of AmigaDOS. One data block of an AmigaDOS disk containing program data consists of 512 bytes. Of these 512 bytes DOS only allocates 488 bytes for data and the remainder go to data management. When you load a program, the management data executes an elaborate memory transfer. This is where the FastFileSystem comes into play. It ensures that management data is no longer necessary in a data block and that all 512 bytes are ready for use as program data.

When you read sequential multiple data blocks of a program, a single read access can perform this task. Because the data management structure has been removed, this block can be read directly into the desired memory address. The increase in speed is enormous. All disk operations can be increased in speed by a factor of five. The disk space saved by releasing the management data of a disk is also quite large. We calculated that a 20 megabyte hard disk on which you placed the FastFileSystem could save 1.5 megabytes of memory.

The use of FFS boils down to this: You get more disk for the same money and you better disk access in any case.

Version 1.3 defaulted to an inactive FastFileSystem. You must first inform the system that you need the handler of the same name and from which medium this should come. We recommend that the desired device be entered into the Mount list using ED DEVS/Mountlist. This can later be added to the Startup-sequence using the Mount command. We've prepared a version of the modified Mount lists to let you quickly adapt to the FFS.

---

### 13.4.1 FFS and hard disks

The most cost-effective storage comes from hard disks, because they can hold large amounts of memory. Here is a possible Mount list entry that could be placed in the FastFileSystem of your hard disk. When you integrate it into your Mount list, do the following:

Copy all of the files from your hard disk to normal disks. Enter the following to mount it:

Mount FHD:

Format your FastHardDisk with the following command sequence:

```
FORMAT DRIVE FHD: NAME "FastHardDisk"
```

Copy your files onto the FHD: device. You'll be surprised how many more files you can put on the hard disk.

```
FHD:      Device = hddisk.device      /* access to HD */
          FileSystem = L:FastFileSystem /* all clear      */
          Unit = 1                    /* Device 0 waiting
                                     on AmigaDOS     */
          Flags = 0                   /* for OpenDevice */
          Surfaces = 4                /* Disk surfaces  */
          BlocksPerTrack = 15         /* Number of blocks
                                     per track      */
          Reserved = 2                /* Bootblocks     */
          Interleave = 0              /* Block setup    */
          LowCyl = 10                 /* From
                                     cylinder 10   */
          HighCyl = 800               /* to cyl. 800   */
          Buffers = 11                /* Read buffers   */
          BufMemType = 1              /* same
                                     (5=FastRAM)  */
          GlobVec = -1                /* No GlobVec     */
          Mount = 1                  /* Load handler
                                     immediately after
                                     entering MOUNT */
          DosType = 0x444F5301        /* Identifier code
                                     for FFS      */
#                                     /* End of entry  */
```

You must add the following line to the Startup-sequence to implement the FastHardDisk:

```
MOUNT FHD:
```

### 13.4.2 FFS and recoverable RAM disk

The RAM disk RAD: can also work with the FastFileSystem. However, RAD: becomes non-resistant to resets when used with FFS (data on this disk disappears following a reset). The memory conservation and speed factor alone are reasons enough to use RAD: for many applications. But the possibility always exists that the RAM disk could be wiped out. You'll need to change the name and Unit arguments of the RAD: Mount list. Use the next highest number to



avoid any overlap. The corresponding Mount list entry may look like the following:

```
FRAD: Device = ramdrive.device
      FileSystem = L:FastFileSystem
      Unit = 1 /* Number of the RAM-Disk */
      Flags = 0
      Surfaces = 2
      BlocksPerTrack = 11
      Reserved = 2
      Interleave = 0
      Mount = 1
      LowCyl = 0
      DosType = 0x444F5301
      HighCyl = 79 /* Available RAM (512K->5)*/
      Buffers = 22 /* ditto */
      BufMemType = 1 /* Same as where it lies */
#
```

You must insert the following in the startup sequence:

```
MOUNT FRAD:
FORMAT >NIL: <NIL: DRIVE FRAD: NAME "FSS in RAM"
```

The NIL device supports both redirection commands. The first suppresses any output, while the second suppresses any requesters asking you to insert a disk in RAM. If you work with programs which allow you to choose between DF0 to DF3, we recommend that the original name be given in the Mount list. The drive must be placed in the startup sequence in place of FRAD if the original device is desired.

## 13.5 The new math libraries

Who hasn't dreamt of a 68030 processor and a 68882 floating point math coprocessor? The prices of these components are a little out of most people's leagues.

New math libraries provide faster math calculations. The speed when processing IEEE floats, like those used with the `x#`- variables in BASIC, executes much faster. After an exact analysis we determined that we have found the fastest known floating point routine currently on the planet. So as not to lead you astray, here is an example of the way this routine can be used in BASIC:

```

DECLARE FUNCTION IEEEEDPSin# LIBRARY
LIBRARY "mathieeedoubtrans.library" 'BASIC does not
                                     'accept pathnames
'use CHDIR [Path for the BMAP-Files]!
PI#=4*ATN(1)      'PI is calculated (fullcircle=2*PI)
CIRC#=2*PI#
FOR I%=1 TO 359                                     'circle in degrees
  ANGLE#=CIRC#/I%                                   'angle from 2*PI
  HighLong&=PEEKL(VARPTR(ANGLE#))                 'first Long of DFloat
  LowLong& =PEEKL(VARPTR(ANGLE#)+4)                'second Long of Float
  SINUS#=IEEEEDPSin#(HighLong&,LowLong&)          'Call function
  PSET(I%,90-INT(SINUS#*50)),1                     'Draw pixel
NEXT
FOR I%=1 TO 359                                     'Just for Demo
  ANGLE#=CIRC#/I%                                   'Look how fast
  SINUS#=SIN(ANGLE#)                               'BASIC is...
  PSET(I%,90-INT(SINUS#*50)),2                     'other color
NEXT
LIBRARY CLOSE

```

The first `FOR/NEXT` loop is slower than the second loop. The `VARPTR` function must be called 720 times, the slow `PEEKL` must be called 720 times, the addition of the value four 360 times, and the routines call 360 times with assignment from two long values, which doesn't go quickly. More lines and variable assignments distort the first loop. All of these limitations are amazing. And the end result: The first loop is just as fast as the second loop.

The `MathIEEEdoubtrans` library includes all the possible transcendental math functions executable on double-precision floating point numbers. The `MathIEEEdoubbas` library, which contains the simple calculation functions, is fast. Transcendental math functions even come in handy for BASIC users, because they allow you to find the arcsine without using calculation programs at the same speed as the sine function.

Here is an overview of the functions:

**MathIeeedoubBas-Library**

x,y, Double			Double-precision floating point number (BASIC: 2 longs instead of x and y)
Long			Positive/negative long integer number
Long (D0)	IEEEDPFix -30	(x) (D0/D1)	Double float/long integer conversion
Double (D0/D1)	IEEEDPFIt -36	(Long) (D0)	Long integer/double float conversion
Long (D0)	IEEEDPCmp -42	(x,y) (D0/D1,D2/D3)	Compare x and y (cc set for bcc) Applies to the following: x > y -->1 x=y -->0 x<y -->-1
Long (D0)	IEEEDPTst -48	(x) (D0/D1)	Compare x and 0 (cc set; result handled as in IEEEDPCmp when y=0)
Double (D0/D1)	IEEEDPAbs -54	(x) (D0/D1)	Returns absolute value of x
Double (D0/D1)	IEEEDPNeg -60	(x) (D0/D1)	Function: Double=-x
Double (D0/D1)	IEEEDPAdd -66	(x,y) (D0/D1,D2/D3)	Function: Double=x+y
Double (D0/D1)	IEEEDPSub -72	(x,y) (D0/D1,D2/D3)	Function: Double=x-y
Double (D0/D1)	IEEEDPMul -78	(x,y) (D0/D1,D2/D3)	Function: Double=x*y
Double (D0/D1)	IEEEDPDiv -84	(x,y) (D0/D1,D2/D3)	Function: Double=x/y
Double (D0/D1)	IEEEDPFloor -90	(x) (D0/D1)	Returns greatest integer less than or equal to x
Double (D0/D1)	IEEEDPCeil -96	(x) (D0/D1)	Returns smallest integer greater than or equal to x

**MathIeeeDoubTrans-Library**

Double (D0/D1)	IEEEDPAtan -30	(x) (D0/D1)	Returns arc- tangent of x
Double (D0/D1)	IEEEDPSin -36	(x) (D0/D1)	Returns sine of x
Double (D0/D1)	IEEEDPCos -42	(x) (D0/D1)	Returns cosine of x
Double (D0/D1)	IEEEDPTan -48	(x) (D0/D1)	Returns tangent of x
Double (D0/D1)	IEEEDPSincos -54	(x,VARPTR) (D0/D1,A0)	Double calc: Compute sine of x, put cosine of x in VARPTR
Double (D0/D1)	IEEEDPSinh -60	(x) (D0/D1)	Returns hyperbolic sine of x
Double (D0/D1)	IEEEDPCosh -66	(x) (D0/D1)	Returns hyperbolic cosine of x
Double (D0/D1)	IEEEDPTanh -72	(x) (D0/D1)	Returns hyperbolic tangent of x
Double (D0/D1)	IEEEDPExp -78	(x) (D0/D1)	Exponent of e Function: Double=e^x
Double (D0/D1)	IEEEDPLog -84	(x) (D0/D1)	Returns natural logarithm of x
Double (D0/D1)	IEEEDPPow -90	(x,y) (D0/D1,D2/D3)	Function: Double=x^y
Double (D0/D1)	IEEEDPSqrt -96	(x) (D0/D1)	Returns square root of x
Float (D0)	IEEEDPTieee -102	(x) (D0/D1)	Calc x in IEEE floating point single precision
Double (D0/D1)	IEEEDPFieee -108	(Float) (D0)	Compute single precision float in double precision
Double (D0/D1)	IEEEDPAsin -114	(x) (D0/D1)	Returns arcsine of x
Double (D0/D1)	IEEEDPAcos -120	(x) (D0/D1)	Returns arccosine of x
Double (D0/D1)	IEEEDPLog10 -126	(x) (D0/D1)	Returns base 10 logarithm of x

The BASIC programmer should remember that you can't have a double float in the system routine. You must give this 64-bit variable in the form of two long values which get their values through PEEKL (VARPTR()). The sine demo in this section is an example of this technique. Another feature of BASIC is the short IEEE library name of the command of the same name.

You can give most libraries longer pathnames. With this library the actual pathname may already be so long that no more path data can be given. When you do this, a File not found error ensues. If you want

to access the math libraries, you must place the system start disk with the corresponding BMAP files in the current directory or in the LIBS: directory. Or you can add a CHDIR statement before the Library command.



# **Appendices**





# A. AmigaBASIC tokens

Token (hex.)	value (dec.)	AmigaBASIC command
80	128	ABS
81	129	ASC
82	130	ATN
83	131	CALL
84	132	CDBL
85	133	CHR\$
86	134	CINT
87	135	CLOSE
88	136	COMMON
89	137	COS
8A	138	CVD
8B	139	CVI
8C	140	CVS
8D	141	DATA
8E (3A)	142 (58)	ELSE
8F	143	EOF
90	144	EXP
91	145	FIELD
92	146	FIX
93	147	FN
94	148	FOR
95	149	GET
96	150	GOSUB
97	151	GOTO
98	152	IF
99	153	INKEY\$
9A	154	INPUT
9B	155	INT
9C	156	LEFT\$
9D	157	LEN
9E	158	LET
9F	159	LINE
A1	161	LOC
A2	162	LOF
A3	163	LOG
A4	164	LSET
A5	165	MID\$
A6	166	MKD\$
A7	167	MKI\$
A8	168	MKS\$
A9	169	NEXT
AA	170	ON
AB	171	OPEN

Token (hex.)	value (dec.)	AmigaBASIC command
AC	172	PRINT
AD	173	PUT
AE	174	READ
AF	175	REM
AFE8 (3A)	175 232 (58)	'
B0	176	RETURN
B1	177	RIGHT\$
B2	178	RND
B3	179	RSET
B4	180	SGN
B5	181	SIN
B6	182	SPACE\$
B7	183	SQR
B8	184	STR\$
B9	185	STRING\$
BA	186	TAN
BC	188	VAL
BD	189	WEND
BE EC	190 236	WHILE
BF	191	WRITE
C0	192	ELSEIF
C1	193	CLNG
C2	194	CVL
C3	195	MKL\$
C4	196	AREA
E3	227	STATIC
E4	228	USING
E5	229	TO
E6	230	THEN
E7	231	NOT
E9	233	>
EA	234	=
EB	235	<
EC	236	+
ED	237	-
EE	238	*
EF	239	/
F0	240	^
F1	241	AND
F2	242	OR
F3	243	XOR
F4	244	EQV
F5	245	IMP
F6	246	MOD
F7	247	\
F8 81	248 129	CHAIN
F8 82	248 130	CLEAR
F8 83	248 131	CLS
F8 84	248 132	CONT
F8 85	248 133	CSNG

Token (hex.)	value (dec.)	AmigaBASIC command
F8 86	248 134	DATE\$
F8 87	248 135	DEFINT
F8 88	248 136	DEFSNG
F8 89	248 137	DEFDBL
F8 8A	248 138	DEFSTR
F8 8B	248 139	DEF FN
F8 8C	248 140	DELETE
F8 8D	248 141	DIM
F8 8E	248 142	EDIT
F8 8F	248 143	END
F8 90	248 144	ERASE
F8 91	248 145	ERL
F8 92	248 146	ERROR
F8 93	248 147	ERR
F8 94	248 148	FILES
F8 95	248 149	FRE
F8 96	248 150	HEX\$
F8 97	248 151	INSTR
F8 98	248 152	KILL
F8 99	248 153	LIST
F8 9A	248 154	LLIST
F8 9B	248 155	LOAD
F8 9C	248 156	LPOS
F8 9D	248 157	LPRINT
F8 9E	248 158	MERGE
F8 9F	248 159	NAME
F8 A0	248 160	NEW
F8 A1	248 161	OCT\$
F8 A2	248 162	OPTION
F8 A3	248 163	PEEK
F8 A4	248 164	POKE
F8 A5	248 165	POS
F8 A6	248 166	RANDOMIZE
F8 A8	248 168	RESTORE
F8 A9	248 169	RESUME
F8 AA	248 170	RUN
F8 AB	248 171	SAVE
F8 AD	248 173	STOP
F8 AE	248 174	SWAP
F8 AF	248 175	SYSTEM
F8 B0	248 176	TIME\$
F8 B1	248 177	TRON
F8 B2	248 178	TROFF
F8 B3	248 179	VARPTR
F8 B4	248 180	WIDTH
F8 B5	248 181	BEEP
F8 B6	248 182	CIRCLE
F8 B8	248 184	MOUSE

Token (hex.)	value (dec.)	AmigaBASIC command
F8 B9	248 185	POINT
F8 BA	248 186	PRESET
F8 BB	248 187	PSET
F8 BC	248 188	RESET
F8 BD	248 189	TIMER
F8 BE	248 190	SUB
F8 BF	248 191	EXIT
F8 C0	248 192	SOUND
F8 C2	248 194	MENU
F8 C3	248 195	WINDOW
F8 C5	248 197	LOCATE
F8 C6	248 198	CSRLIN
F8 C7	248 199	LBOUND
F8 C8	248 200	UBOUND
F8 C9	248 201	SHARED
F8 CA	248 202	UCASE\$
F8 CB	248 203	SCROLL
F8 CC	248 204	LIBRARY
(F8)(D1)	(248)(209)	placed after target of SUB program call without CALL
F8 D2	248 210	PAINT
F8 D3	248 211	SCREEN
F8 D4	248 212	DECLARE
F8 D5	248 213	FUNCTION
F8 D6	248 214	DEFLNG
F8 D7	248 215	SADD
F8 D8	248 216	AREAFILL
F8 D9	248 217	COLOR
F8 DA	248 218	PATTERN
F8 DB	248 219	PALETTE
F8 DC	248 220	SLEEP
F8 DD	248 221	CHDIR
F8 DE	248 222	STRIG
F8 DF	248 223	STICK
F9 F4	249 244	OFF
F9 F5	249 245	BREAK
F9 F6	249 246	WAIT
F9 F7	249 247	USR
F9 F8	249 248	TAB
F9 F9	249 249	STEP
F9 FA	249 250	SPC
F9 FB	249 251	OUTPUT
F9 FC	249 252	BASE
F9 FD	249 253	AS
F9 FE	249 254	APPEND
F9 FF	249 255	ALL
FA 80	250 128	WAVE
FA 81	250 129	POKEW
FA 82	250 130	POKEL
FA 83	250 131	PEEKW

Token (hex.)	value (dec.)	AmigaBASIC command
FA 84	250 132	PEEKL
FA 85	250 133	SAY
FA 86	250 134	TRANSLATE\$
FA 87	250 135	OBJECT.SHAPE
FA 88	250 136	OBJECT.PRIORITY
FA 89	250 137	OBJECT.X
FA 8A	250 138	OBJECT.Y
FA 8B	250 139	OBJECT.VX
FA 8C	250 140	OBJECT.VY
FA 8D	250 141	OBJECT.AX
FA 8E	250 142	OBJECT.AY
FA 8F	250 143	OBJECT.ShellP
FA 90	250 144	OBJECT.PLANES
FA 91	250 145	OBJECT.HIT
FA 92	250 146	OBJECT.ON
FA 93	250 147	OBJECT.OFF
FA 94	250 148	OBJECT.START
FA 95	250 149	OBJECT.STOP
FA 96	250 150	OBJECT.CLOSE
FA 97	250 151	COLLISION
FB FF	251 255	PTAB

## B . Other tokens

Token	Definition
\$01	Variable number follows in hexadecimal notation (High/Low = 2 Byte), e.g.: (\$01) \$00 \$00 = Variable 0
\$02	Label number follows in hex (H/L = 2 Byte), e.g.: (\$02) \$01 \$00 = label 256
\$03	Jump to label with the following number (H/M/L = 3 B.), e.g.: (\$03) \$00 \$00 \$0A = to label 10
\$0B	An octal number follows (hexadecimal in High/Low format = 2 bytes). e.g.: (\$0B) \$00 \$06 = &O 6
\$0C	A 2-byte hexadecimal number follows in H/L format, e.g.: (\$0C) \$F8 \$EC = \$ F8EC
\$0E	Jump to the line with the following line number (H/M/L), e.g.: (\$0E) \$00 \$27 \$10 = after line 10000
\$0F	A positive integer with a value from 10 to 255 follows, e.g.: (\$0F) \$FF = 255
\$11- \$1A	A positive integer with a value from 0 to 9 follows, e.g.: \$11 = 0, \$12 = 1 ... \$19 = 8, \$1A = 9
\$1C	A 2-byte integer with leading character follows, e.g.: (\$1C) \$80 \$A0 = -160
\$1D	A 4-byte floating-point number follows, e.g.: (\$1D) \$3C \$23 \$D7 \$0A = 0.01
\$1E	A 4-byte integer follows, e.g.: (\$1E) \$00 \$00 \$80 \$00 = 32768&
\$1F	An 8-byte floating-point number follows, e.g.: (\$1F) \$3E45 \$798E \$E230 \$8C3A = 0.00000001

## C. Shell Escape sequences

Using the **Ctrl** and **Esc** keys, sequences can be entered directly in the CLI/Shell or by using the `Echo` command inside a batch file that can effect the output. When the `Echo` command is used, the **Esc** key can be set using the character combination `*e`.

### Escape sequences

<code>&lt;Esc&gt;c</code>	The contents of the CLI/Shell window are erased and all other modes are turned off.
<code>&lt;Esc&gt;[0m</code>	All other modes are turned off.
<code>&lt;Esc&gt;[1m</code>	Bold text is turned on.
<code>&lt;Esc&gt;[2m</code>	Color number 2 becomes the text color (black).
<code>&lt;Esc&gt;[3m</code>	Italic text is turned on.
<code>&lt;Esc&gt;[30m</code>	Color number 0 becomes the text color (blue).
<code>&lt;Esc&gt;[31m</code>	Color number 1 becomes the text color (white).
<code>&lt;Esc&gt;[32m</code>	Color number 2 becomes the text color (black).
<code>&lt;Esc&gt;[33m</code>	Color number 3 becomes the text color (orange).
<code>&lt;Esc&gt;[4m</code>	The text is underlined.
<code>&lt;Esc&gt;[40m</code>	Color number 0 becomes the background color (blue).
<code>&lt;Esc&gt;[41m</code>	Color number 1 becomes the background color (white).
<code>&lt;Esc&gt;[42m</code>	Color number 2 becomes the background color (black).
<code>&lt;Esc&gt;[43m</code>	Color number 3 becomes the background color (orange).
<code>&lt;Esc&gt;[7m</code>	The text becomes inverted.
<code>&lt;Esc&gt;[8m</code>	The text becomes invisible (blue).
<code>&lt;Esc&gt;[nu</code>	The CLI/Shell window becomes n characters wide.
<code>&lt;Esc&gt;[nt</code>	Number of lines in the CLI/Shell window is set to n.
<code>&lt;Esc&gt;[nx</code>	The left border is set at n pixels.
<code>&lt;Esc&gt;[ny</code>	The distance from the top is set at n pixels.

### Control sequences

When entering control sequences you must press the **Ctrl** key and the corresponding letter key.

<b>Ctrl</b> <b>H</b>	Deletes last character entered.
<b>Ctrl</b> <b>I</b>	Moves cursor one tab position to the right.
<b>Ctrl</b> <b>J</b>	Linefeed.
<b>Ctrl</b> <b>K</b>	Moves cursor up one line.
<b>Ctrl</b> <b>L</b>	Clears CLI/Shell window.
<b>Ctrl</b> <b>M</b>	Same as <b>Ctrl</b> <b>J</b> .
<b>Ctrl</b> <b>N</b>	Enables Alt character set.
<b>Ctrl</b> <b>O</b>	Enables normal character set.
<b>Ctrl</b> <b>X</b>	Deletes current line.
<b>Ctrl</b> <b>\</b>	Marks the end of a file.

## D. Printer Escape Sequences

The following printer escape sequences are translated using the printer drivers included in the Preferences editors.

Printer Escape sequence	Meaning
<Esc>c	Initialize (reset) printer
<Esc>#1	Disable all other modes
<Esc>D	Line feed
<Esc>E	Line feed + carriage return
<Esc>M	One line up
<Esc>[0m	Normal characters
<Esc>[1m	Bold on
<Esc>[22m	Bold off
<Esc>[3m	Italics on
<Esc>[23m	Italics off
<Esc>[4m	Underlining on
<Esc>[24m	Underlining off
<Esc>[xm	Colors (x=30 - 39 [foreground] or 40 - 49 [background])
<Esc>[0w	Normal text size
<Esc>[2w	Elite on
<Esc>[1w	Elite off
<Esc>[4w	Condensed type on
<Esc>[3w	Condensed type off
<Esc>[6w	Enlarged type on
<Esc>[5w	Enlarged type off
<Esc>[2"z	NLQ on
<Esc>[1"z	NLQ off
<Esc>[4"z	Double strike on
<Esc>[3"z	Double strike off
<Esc>[6"z	Shadow type on
<Esc>[5"z	Shadow type off
<Esc>[2v	Superscript on
<Esc>[1v	Superscript off
<Esc>[4v	Subscript on
<Esc>[3v	Subscript off
<Esc>[0v	Back to normal type



Printer Escape sequence	Meaning
<Esc>[2p	Proportional type on
<Esc>[1p	Proportional type off
<Esc>[0p	Delete proportional spacing
<Esc>[xE	Proportional spacing = x
<Esc>[5F	Left justify
<Esc>[7F	Right justify
<Esc>[6F	Set block
<Esc>[0F	Set block off
<Esc>[3F	Justify letter width
<Esc>[1F	Center justify
<Esc>[0z	Line dimension 1/8 inch
<Esc>[1z	Line dimension 1/6 inch
<Esc>[xt	Page length set at x lines
<Esc>[xq	Perforation jumps to x lines
<Esc>[0q	Perforation jumping off
<Esc>(B	American character set
<Esc>(R	French character set
<Esc>(K	German character set
<Esc>(A	English character set
<Esc>(E	Danish character set (Nr.1)
<Esc>(H	Swedish character set
<Esc>(Y	Italian character set
<Esc>(Z	Spanish character set
<Esc>(J	Japanese character set
<Esc>(6	Norwegian character set
<Esc>(C	Danish character set (Nr.2)
<Esc>#9	Set left margin
<Esc>#0	Set right margin
<Esc>#8	Set header
<Esc>#2	Set footer
<Esc>#3	Delete margins
<Esc>[xyr	Header x lines from top; footer y lines from bottom
<Esc>[xys	Set left margin (x) and right margin (y)
<Esc>H	Set horizontal tab
<Esc>J	Set vertical tab
<Esc>[0g	Delete horizontal tab
<Esc>[3g	Delete all horizontal tabs
<Esc>[1g	Delete vertical tab
<Esc>[4g	Delete all vertical tabs
<Esc>#4	Delete all tabs
<Esc>#5	Set standard tabs

---

## E. Program Notes

The companion diskette is not intended to be a 'stand-alone' product. Before using the programs of our companion diskette, be sure to read the portion of the book pertaining to the programs you are interested in.

Because of the many different languages and environments used for the various examples in our book, we cannot instruct you on how each particular program or file should be loaded. However, we have had a number of inquiries regarding the execution of the BASIC programs, and would like to describe the common steps involved.

If the program is written in AmigaBASIC, it should be either loaded from AmigaBASIC directly, or it can alternatively be double-clicked as long as you have placed a copy of AmigaBASIC in the main directory of the companion diskette, or have copied the program from the companion diskette to a diskette, which contains a copy of AmigaBASIC, in the main directory. When loading programs directly from AmigaBASIC, be sure to include the full drive specification and path (directories) as well as the filename. You can also use the CHDIR command first to change to the proper directory.

AmigaBASIC requires .bmap files to call system library functions from within an AmigaBASIC program, these files contain the necessary information for all commands organized in the library. The BasicDemos directory contains exec, dos, and graphics. Some, however, require that you create additional .bmap files prior to the first execution (many programs, for example, require the intuition and diskfont libraries). These can easily be created using the ConvertFD program in the BasicDemos drawer of the Extras diskette. The names of FD files are: libraryname\_lib.fd. The names of bmaps must be: libraryname.bmap. As an example, to create a layers.bmap: when prompted for the name of the FD file enter: fd1.2/layers\_lib.fd, when prompted for the bmap filename enter: layers.bmap.

The placement of bmaps is also critical, when you use the LIBRARY statement AmigaBASIC will look for the file in two places:

1. The libs: directory (usually the libs directory of the workbench disk you booted with).
2. The directory AmigaBASIC is cd'd to. When you enter AmigaBASIC, this is the directory containing the icon you double-click.

To run the programs in the basic demos drawer, either double-click the demo's icon, or:

- Double-click AmigaBASIC.
- Type `CHDIR "df1:BasicDemos"` <Return>, in the output window.
- Open the desired demo.
- Start the demo.

**Note:** ConvertFD adds an 'x' to the beginning of any Amiga system routine that conflicts with an AmigaBASIC keyword. The known conflicting routine names can be found at the end of ConvertFD. Using these .bmaps, the conflicting routines may be declared and called by adding an 'x' to the beginning of the routine name (i.e. xread). All other system routines are declared/called by their usual names.



# Index

2000A board .....351  
 3-D glasses .....105  
 68000 commands .....359  
 68010 processor .....350

## A

ADDBUFFERS .....24  
 ALIAS .....24  
 AllocMem() .....129  
 Amiga 500 board .....352  
 AmigaDOS .....7  
 Amiga hardware .....49  
 AmigaBASIC .....51, 207, 371  
 AmigaDOS .....8  
 Argument .....23  
 ASCII file .....158, 195  
 ASCII format .....371  
 ASCII text .....372  
 Ask .....24  
 ASSIGN .....24, 192  
 assign command .....10  
 Autoknob .....154  
 Automatic backups .....366  
 AVAIL .....25

## B

BASIC editor .....373  
 BASIC file checking program .....196  
 BCPL language .....311  
 Binary files .....195  
 BINDDRIVERS .....25  
 Blank line killer program .....223  
 Block .....192  
 bmap files .....53  
 boot block .....303  
 Border .....137  
 Borderless Shell .....368  
 BREAK .....25  
 Byte Bandit virus .....303

## C

C programming language.....299  
 CALL.....210  
 CD.....26  
 CHAIN command .....195  
 CHANGETASKPRI.....26  
 Checking for errors.....280  
 CHR\$(27) .....373  
 Chunks .....76  
 CLI  
   access .....8  
   appending files.....21  
   assign command .....10  
   copy command .....9, 12, 13, 14  
   diskcopy command .....15  
   execute command .....15, 20  
   join command .....21  
   list command .....11  
   loadwb command .....17  
   mkdir command .....10  
   multiple windows .....42  
   multitasking .....17, 18  
   newshell command .....18  
   printing C lists .....17  
   printing commands .....14  
   quick parameter .....14  
   quitting .....9  
   run command .....18  
   say command .....17  
   search command.....21  
   sort command.....21  
   stack command .....22  
   stopping programs .....11  
   text output .....19  
   type command .....17  
 Clipboard device .....380  
 CloseAll .....129  
 ClosePrinter routine .....340  
 cloud graphic .....315  
 Code register .....359  
 ColorCycle .....314  
 Command Line Interface—see CLI .....7  
 COMPLEMENT .....54

COMPLEMENT drawing mode ..... 54  
 COMPLEMENT mode ..... 56  
 Computer viruses ..... 303  
 CON handler ..... 365  
 Console Device ..... 117, 368, 382  
 control key ..... 367  
 Control sequences ..... 399  
 ConvertFD ..... 53, 126  
 COPY ..... 26, 367  
 copy command ..... 9  
 Copying diskettes ..... 15  
 copyright messages ..... 193  
 Cross-reference program ..... 216  
 cursor ..... 289  
 Cursor control ..... 60  
 Cylinders ..... 321

## D

DATA generator program ..... 212  
 DATA statements ..... 147  
 DATE ..... 27  
 DeciGEL ..... 357  
 DefChip() ..... 129  
 DELETE ..... 27  
 DestinationFunctionCodeRegister  
 (DFC) ..... 359  
 DEVS ..... 373  
 DIR ..... 27, 375  
 Direct disk access ..... 321  
 Discard ..... 247  
 Disk access errors ..... 278  
 Disk icons ..... 368  
 DISKCHANGE ..... 28  
 DISKCOPY ..... 28  
 diskcopy command ..... 15  
 DISKDOCTOR ..... 28, 375  
 Diskette sector design ..... 329  
 Diskettes  
   copying ..... 12, 13, 14  
 diskfont.library ..... 52  
 dos.library ..... 52  
 DrawBorder() ..... 147  
 Drawing modes ..... 54  
 drawing program ..... 163  
 DualBitMap program ..... 167  
 DumpRastPort structure ..... 340

## E

ECHO ..... 28, 365  
 ED ..... 28  
 EDIT ..... 28  
 ELSE ..... 28, 210  
 Empty Trash ..... 247  
 ENDCLI ..... 29  
 ENDIF ..... 29  
 ENDSHELL ..... 29  
 ENDSKIP ..... 29  
 error handling ..... 277, 280  
 errors ..... 278  
 Escape sequences ..... 399  
 EVAL ..... 29  
 Exception routine ..... 300  
 exec.library ..... 52, 372  
 EXECUTE ..... 29  
 execute command ..... 11, 15, 20  
 Extended selection ..... 247  
 Extras diskette ..... 53

## F

fade-in ..... 89  
 fade-out ..... 89  
 Fade-over ..... 91, 93  
 FAILAT ..... 29  
 FAULT ..... 30  
 FF ..... 30  
 FFS ..... 383  
 file monitor ..... 178  
 FILENOTE ..... 30  
 final argument ..... 23  
 floating point variables ..... 308  
 Floodfill ..... 78  
 Fonts ..... 114  
 FORMAT ..... 30  
 FreeMem() ..... 339

## G

Gadget structure ..... 137, 153  
 GadgetDef ..... 130, 135  
 gadgets ..... 126, 129, 135, 146  
 GET/GETENV ..... 30  
 GetMsg() ..... 132  
 GetPrinterData subprogram ..... 339

Graphic commands..... 54  
 Graphic dumps .....340  
 graphics.library ..... 52  
 Guru Meditation .....300, 375

## H

Halfbrite mode ..... 85  
 Hardcopy .....344  
 hexadecimal .....178  
 HighCyl ..... 47  
 Hold-and-Modify mode (HAM)..... 85

## I

I/O (input/output) .....319  
 I/O message port .....319  
 I/O request block .....319  
 Icons ..... 13, 125, 245  
 ICONX ..... 31  
 IF ..... 31  
 IFF transfer ..... 61  
 IFF-object conversion..... 71  
 INFO ..... 31, 153  
 InitDRPReq .....344  
 Input and output .....319  
 INSTALL ..... 31  
 Install command .....303  
 Instruction register .....300  
 Interchange File Format (IFF)..... 61  
 Interleaved Bitmap (ILBM) ..... 61  
 IntuiText .....136  
 Intuition .....7, 56, 125, 158  
 Intuition knob graphic .....153  
 Intuition library .....372  
 Intuition window .....126  
 IntuitionMsg .....131  
 intuition.library..... 52  
 INVERSEVID ..... 54  
 INVERSEVID drawing mode ..... 54  
 IOrequest .....344

## J

JAM 1..... 54  
 JAM 1 drawing mode ..... 54  
 JAM2..... 54, 136  
 JOIN ..... 31

join command.....21  
 JSR (Jump to SubRoutine) .....311

## K

kernel.....52  
 key combinations .....367  
 keyword.....23  
 Kickstart diskette..... 8  
 knob graphic ..... 153

## L

LAB .....32  
 Label handling.....205  
 LIBRARY command.....51  
 line buffer overflow .....374  
 Link module .....310  
 LIST .....32, 375  
 list command .....11  
 listing.....233  
 LoadSeg ..... 310  
 LoadWB .....33, 368  
 loadwb command .....17  
 LOCATE.....60  
 LOCATE command.....60  
 LOCK .....33, 282  
 LowCyl.....47  
 LPRINT .....375

## M

Machine language.....299  
 MAKEDIR .....33  
 makedir command .....10  
 MAKELINK .....33  
 memory .....371  
 Memory allocation .....332  
 memory allocation routine .....130  
 memory expansion .....351  
 memory handling .....177, 332  
 Memory reservation .....332  
 Menu errors.....279  
 menus.....291  
 MERGE command .....195  
 Microsoft Corporation.....51  
 Mlist .....129  
 Modular work .....370

Motorola chip .....	355
MOUNT .....	33
Mount command .....	43
mouse .....	7
MOVE .....	60
MOVE CCR .....	360
MOVE command.....	60
MOVE SR, Destination .....	360
MOVES .....	360
Multiple arguments.....	23
multitasking .....	17, 18

## N

NEWCLI .....	34
newcli command .....	18
NewCon .....	368
NewCon device .....	382
NEWSHELL .....	34, 367, 368
NIL device .....	385

## O

OpenAll .....	129
OpenDevice() .....	339
OpenPrinter subprogram .....	339
OpenWindow() .....	129

## P

PAL.....	193
PALETTE command.....	87, 172
Papst Multi-Fan.....	358
PAR .....	373
Patching .....	193
PATH .....	34
Peripherals .....	319
PIPE device .....	380
PolyDraw() .....	147
Power LED .....	309
Preferences .....	126, 146, 248, 373
printer device .....	335
printer parameters .....	335
Printer spooler .....	47
PrinterData .....	340
PrinterExtendedData .....	339
Program header checking program .....	198
PROMPT .....	34

PropInfo .....	153
proportional gadget .....	153
PROTECT .....	35
Protected files .....	195
PRT .....	373

## Q

Quick messages .....	368
QUIT .....	35

## R

RAM disk .....	384
RAMBO .....	368
Read(SFC) .....	359
RELABEL .....	35
REmarks.....	209
REMRAD .....	35
RENAME .....	36
Renaming commands.....	45
Requester.....	13, 287
Reserving memory .....	332
RESIDENT .....	36
RTD .....	360
RTE (ReTurn from Exception) .....	300
RTS .....	360
rubberband .....	56
Rubberband demo.....	56
rubberbanding .....	159
circles .....	163
shapes .....	161
RUN .....	36
run command .....	14, 18

## S

say command .....	17, 381
SCA virus .....	303
screen names .....	372
script file .....	367
Scrolling tables .....	146
SEARCH .....	36
search command.....	21
Self-modifying programs .....	239
SER .....	373
SET/SETENV .....	37
SetAlert command .....	357



SETCLOCK ..... 37  
 SETDATE ..... 37  
 SetDrMd() command demo ..... 55  
 SETPATCH ..... 37  
 SetTextFont program ..... 114  
 Shell ..... 8, 246, 365, 368  
     access ..... 8  
     appending files ..... 21  
     assign command ..... 10  
     copy command ..... 9, 12, 13, 14  
     diskcopy command ..... 15  
     Ed editor ..... 15  
     execute command ..... 15, 20  
     join command ..... 21  
     list command ..... 11  
     loadwb command ..... 17  
     mkdir command ..... 10  
     multiple windows ..... 42  
     multitasking ..... 17, 18  
     newshell command ..... 18  
     output ..... 365  
     printing C lists ..... 17  
     printing commands ..... 14  
     quick parameter ..... 14  
     quitting ..... 9  
     run command ..... 18  
     say command ..... 17  
     search command ..... 21  
     SetAlert command ..... 300  
     sort command ..... 21  
     stack command ..... 22  
     stopping programs ..... 11  
     text modes ..... 365  
     text output ..... 19  
     type command ..... 17  
 shifting grids ..... 101  
 Sizing gadget ..... 159  
 SKIP ..... 38  
 Sliders ..... 126, 153  
 Snapshot ..... 20  
 SORT ..... 38  
 sort command ..... 21  
 SourceFunctionCodeRegister (SFC) ..... 359  
 Speak device ..... 381  
 SpecialInfo ..... 153  
 SPST switch ..... 354  
 STACK ..... 38, 375  
 stack command ..... 22  
 standard icons ..... 370  
 Startup-sequence ..... 15, 41, 365  
     editing ..... 15

STATUS ..... 38  
 Status display ..... 192  
 Status lines ..... 166  
 Status register ..... 300  
 String gadget ..... 246  
 string gadgets ..... 153  
 SUB ..... 59  
 SUB programs ..... 59, 210  
 Superstate word ..... 300  
 Supervisor stack ..... 300  
 Switch ..... 23  
 system vectors ..... 304  
 System-Configuration ..... 375

## T

TabOut ..... 147  
 temporary files ..... 365  
 Text styles demo ..... 58  
 three-dimensional graphics ..... 96  
 tokens ..... 209, 211  
 Tool ..... 373  
 Trackdisk.device ..... 321  
 trap errors ..... 280  
 Trap vector ..... 302  
 Trashcan ..... 247  
 TYPE ..... 39, 375  
 type command ..... 17  
 Typestyles ..... 58

## U

UNALIAS ..... 39  
 UnDef() ..... 129  
 UNDERLINE ..... 366  
 Undo buffer ..... 193  
 Undo gadget ..... 193  
 UnloadSeg routine ..... 310  
 UNSET ..... 39  
 UNSETENV ..... 39  
 User input errors ..... 279  
 user interface ..... 245  
 User-friendliness ..... 125  
 Utilities ..... 177

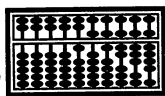
## V

Variable lister program .....	234
variables .....	211, 338
VARPTR command .....	332
Vector graphics .....	96
VectorBaseRegister (VBR) .....	359
VERSION .....	39

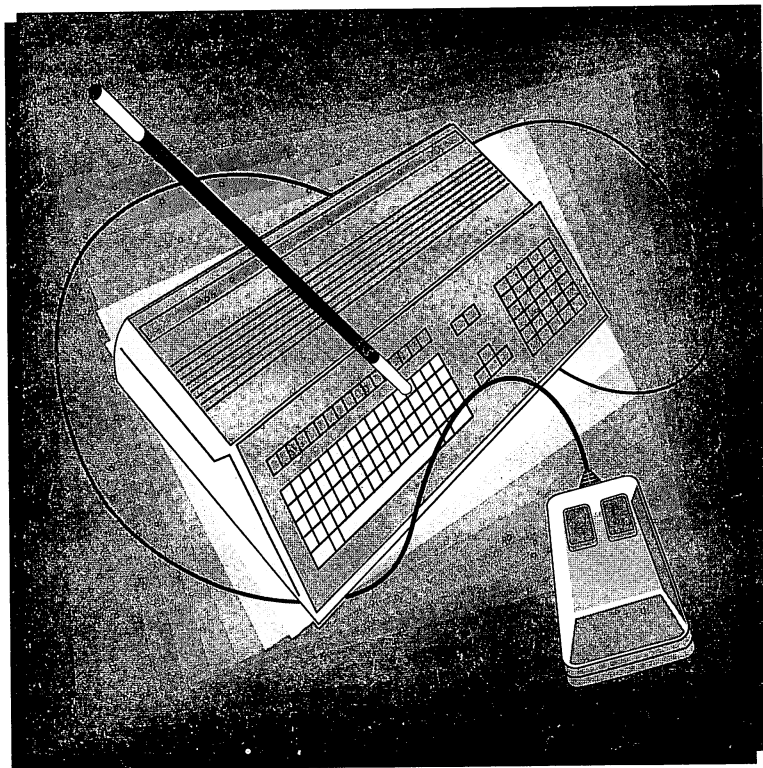
## W

WAIT .....	39
WHICH.....	40
WHILE.....	210
WHY .....	40
wildcard characters .....	11
WinDef() .....	129
window.....	7
Window coordinates .....	88
window name .....	371
Windows in BASIC .....	79
border color change .....	83
border structure.....	81
borderless.....	79
coordinate setting.....	88
gadget disable/enable .....	80
Halfbrite .....	85
HAM .....	85
monochrome Workbench.....	84
Workbench.....	8, 245, 368
Snapshot option.....	20
Workbench Versions .....	245
Write(DFC) .....	359

**Abacus**



# Amiga Catalog

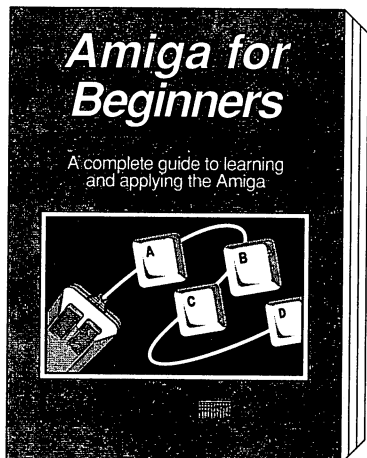


**Order Toll Free 1-800-451-4319**

## Amiga for Beginners

Vol.#1

A perfect introductory book if you're a new or prospective Amiga owner. **Amiga for Beginners** introduces you to Intuition (the Amiga's graphic interface), the mouse, windows, the versatile CLI. This first volume in our Amiga series explains every practical aspect of the Amiga in plain English. Clear, step-by-step instructions for common Amiga tasks. **Amiga for Beginners** is all the info you need to get up and running.



Topics include:

- Unpacking and connecting the Amiga components
- Starting up your Amiga
- Customizing the Workbench
- Exploring the Extras disk
- Taking your first step in AmigaBASIC programming language
- AmigaDOS functions
- Using the CLI to perform "housekeeping" chores
- First Aid, Keyword, Technical appendixes
- Glossary

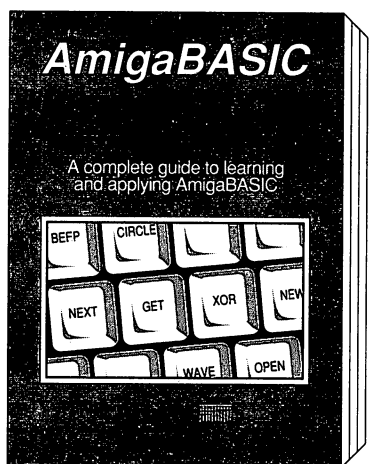
ISBN 1-55755-021-2. Suggested retail price: \$16.95

Companion Diskette not available for this book.

## Amiga BASIC: Inside and Out

Vol.#2

**Amiga BASIC: Inside and Out** is the definitive step-by-step guide to programming the Amiga in BASIC. This huge volume should be within every Amiga user's reach. Every Amiga BASIC command is fully described and detailed. In addition, **Amiga BASIC: Inside and Out** is loaded with real working programs.



Topics include:

- Video titling for high quality object animation
- Bar and pie charts
- Windows
- Pull down menus
- Mouse commands
- Statistics
- Sequential and relative files
- Speech and sound synthesis

ISBN 0-916439-87-9. Suggested retail price: \$24.95

**Companion Diskette available:** *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus. \$14.95*

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

## Amiga 3D Graphic Programming in BASIC

Vol.#3

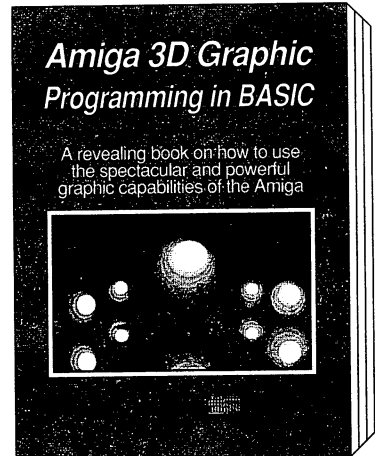
**Amiga 3D Graphic Programming in BASIC**- shows you how to use the powerful graphics capabilities of the Amiga. Details the techniques and algorithm for writing three dimensional graphics programs: ray tracing in all resolutions, light sources and shading, saving graphics in IFF format and more.

Topics include:

- Basics of ray tracing
- Using an object editor to enter three-dimensional objects
- Material editor for creating parameters of color, shading and mirroring of objects
- Automatic computation in different resolutions
- Using any Amiga resolution (low-res, high-res, interlace, HAM)
- Different light sources and any active pixel
- Save graphics in IFF format for later recall into any IFF compatible drawing program
- Mathematical basics for the non-mathematician

**ISBN 1-55755-044-1. Suggested retail price: \$19.95**

**Companion Diskette available:** *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus. \$14.95*



## Amiga Machine Language

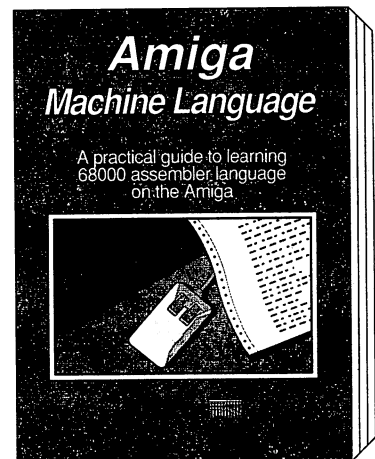
Vol.#4

**Amiga Machine Language** introduces you to 68000 machine language programming presented in clear, easy to understand terms. If you're a beginner, the introduction eases you into programming right away. If you're an advanced programmer, you'll discover the hidden powers of your Amiga. Learn how to access the hardware registers, use the Amiga libraries, create gadgets, work with Intuition and more.

- 68000 microprocessor architecture
- 68000 address modes and instruction set
- Accessing RAM, operating system and multitasking capabilities
- Details the powerful Amiga libraries for access to AmigaDOS
- Simple number base conversions
- Text input and output - Checking for special keys
- Opening CON: RAW: SER: and PRT: devices
- Menu programming explained
- Speech utility for remarkable human voice synthesis
- Complete Intuition demonstration program including Proportional, Boolean and String gadgets

**ISBN 1-55755-025-5. Suggested retail price: \$19.95**

**Companion Diskette available:** *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus. \$14.95*

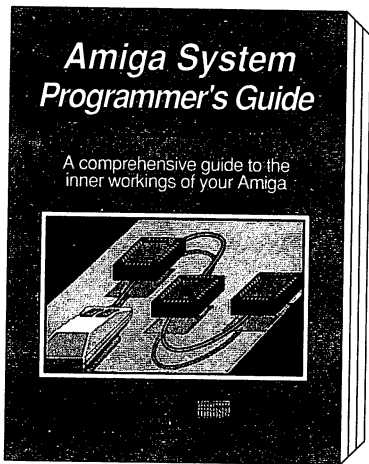


**See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada**

## Amiga System Programmer's Guide

Vol.#6

**Amiga System Programmer's Guide** is a comprehensive guide to what goes on inside the Amiga in a single volume. Explains in detail the Amiga chips (68000, CIA, Agnus, Denise, Paula) and how to access them. All the Amiga's powerful interfaces and features are explained and documented in a clear precise manner.



Topics include:

- EXEC Structure
- Multitasking functions
- I/O management through devices and I/O request
- Interrupts and resource management
- RESET and its operation
- DOS libraries
- Disk management
- Detailed information about the CLI and its commands
- Much more—over 600 pages worth

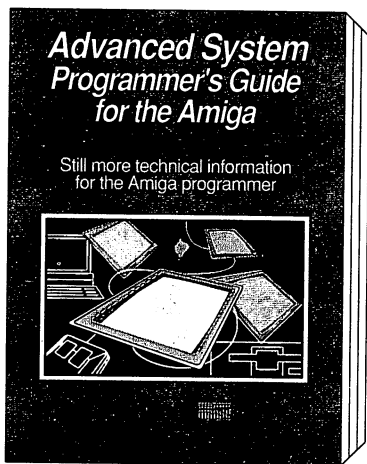
**ISBN 1-55755-034-4. Suggested retail price: \$34.95**

**Companion Diskette available:** *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus. \$14.95*

## Advanced System Programmer's Guide

Vol.#7

**Advanced System Programmer's Guide for the Amiga** - The second volume to our 'system programming' book. References all libraries, with basis and primitive structures. Devices: parallel, serial, printer, keyboard, gameport, input, console, clipboard, audio, translator, and timer trackdisk.



Some of the topics include:

- Interfaces- audio, video RGB, Centronics, serial, disk access, expansion port, and keyboard
- Programming hardware- memory organization, interrupts, the Copper, blitter and disk controller
- EXEC structures- Node, List, Libraries and Tasks
- Multitasking- Task switching, intertask communication, exceptions, traps and memory management
- I/O- device handling and requests
- DOS Libraries- functions, parameters and error messages
- CLI- detailed internal design descriptions

**ISBN 1-55755-047-6. Suggested retail price: \$34.95**

**Companion Diskette available:** *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus. \$14.95*

**See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada**

## AmigaDOS: Inside & Out

**Revised for 2.0**

**Vol.#8**

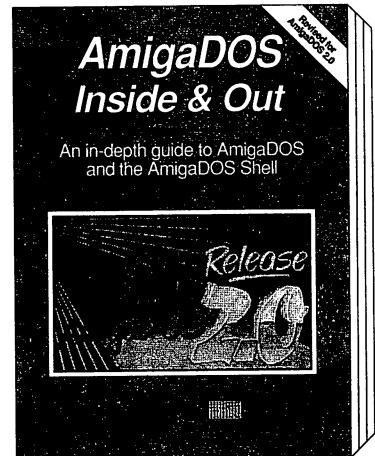
**AmigaDOS: Inside & Out** covers the insides of AmigaDOS from the internal design up to practical applications. **AmigaDOS Inside & Out** will show you how to manage Amiga's multitasking capabilities more effectively. There is also a detailed reference section which helps you find information in a flash, both alphabetically and in command groups. Topics include: Getting the most from the AmigaDOS Shell (wildcards and command abbreviations) • Script (batch) files - what they are and how to write them.

More topics include:

- AmigaDOS - Tasks and handling
- Detailed explanations of CLI commands and their functions
- In-depth guide to ED and EDIT
- Amiga devices and how the AmigaDOS Shell uses them
- Customizing your own startup-sequence
- AmigaDOS and multitasking
- Writing your own AmigaDOS Shell commands in C
- Reference for 1.2, 1.3 and 2.0 commands

**ISBN 1-55755-041-7. Suggested retail price: \$19.95**

**Companion Diskette available:** *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus. \$14.95*



## Amiga Disk Drives: Inside & Out

**Vol.#9**

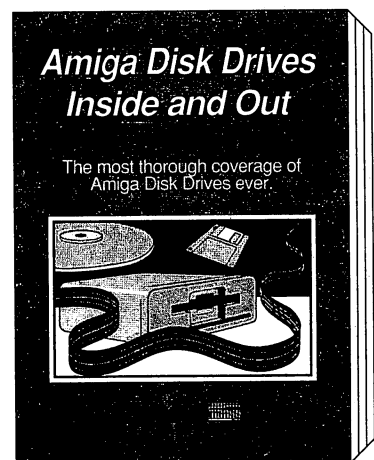
**Amiga Disk Drives: Inside & Out** shows everything you need to know about Amiga disk drives. You'll find information about data security, disk drive speedup routines, disk copy protection, boot blocks, loading and saving programs, sequential and relative file organization and much more.

Topics include:

- Floppy disk operations from the Workbench and CLI
- DOS functions and operations
- Disk block types, boot blocks, checksums, file headers, hashmarks and protection methods
- Viruses and how to protect your boot block
- Trackdisk device: Commands and structures
- Trackdisk-task: Function and design
- MFM, GCR, track design, blockheader, datablocks, coding and decoding data, hardware registers, SYNC and interrupts

**ISBN 1-55755-042-5. Suggested retail price: \$29.95**

**Companion Diskette available:** *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus. \$14.95*



**See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada**

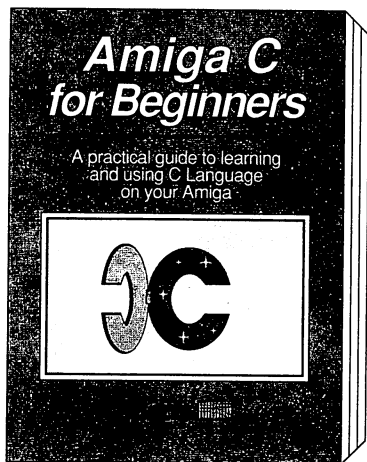
## Amiga C for Beginners

Vol.#10

**Amiga C for Beginners** is an introduction to learning the popular C language. Explains the language elements using examples specifically geared to the Amiga. Describes C library routines, how the compiler works and more.

Topics include:

- Beginner's overview of C
- Particulars of C
- Writing your first program
- The scope of the language (loops, conditions, functions, structures)
- Special features of the C language
- Input/Output using C
- Tricks and Tips for finding errors
- Introduction to direct programming of the operating system (windows, screens, direct text output, DOS functions) Using the LATTICE and AZTEC C compilers



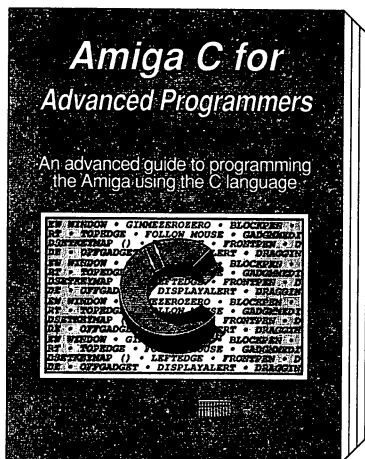
ISBN 1-55755-045-X. Suggested retail price: \$19.95

**Companion Diskette available:** *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus. \$14.95*

## Amiga C for Advanced Programmers

Vol.#11

**Amiga C for Advanced Programmers** contains a wealth of information from the C programming pros: how compilers, assemblers and linkers work, designing and programming user friendly interfaces utilizing the Amiga's built-in user interface Intuition, managing large C programming projects, using jump tables and dynamic arrays, combining assembly language and C codes, using MAKE correctly. Includes the complete source code for a text editor.



Topics include:

- Using INCLUDE, DEFINE and CAST
- Debugging and optimizing assembler sources
- All about programming Intuition including windows, screens, pulldown menus, requesters, gadgets and more
- Programming the console device
- A professional editor's view of problems with developing larger programs
- Debugging C programs with different utilities

ISBN 1-55755-046-8. Suggested retail price: \$34.95

**Companion Diskette available:** *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus. \$14.95*

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada



## Amiga Graphics: Inside & Out

Vol.#13

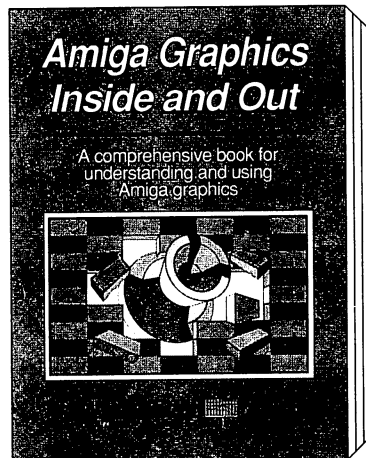
**Amiga Graphics: Inside & Out** will show you the super graphic features and functions of the Amiga in detail. Learn the graphic features that can be accessed from AmigaBASIC or C. The advanced user will learn how to call the graphic routines from the Amiga's built-in graphic libraries. Learn graphic programming in C with examples of points, lines, rectangles, polygons, colors and more. Complete description of the Amiga graphic system- View, ViewPort, RastPort, bitmap mapping, screens, and windows.

Topics include:

- Accessing fonts and type styles in AmigaBASIC
- Loading and saving IFF graphics
- CAD on a 1024 x 1024 super bitmap, using graphic library routines
- Access libraries and chips from BASIC- 4096 colors at once, color patterns, screen and window dumps to printer
- Amiga animation explained including sprites, bobs and AnimObs, Copper and blitter programming

**ISBN 1-55755-052-2. Suggested retail price: \$34.95**

**Companion Diskette available:** *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus. \$14.95*



## Amiga Desktop Video Guide

Vol.#14

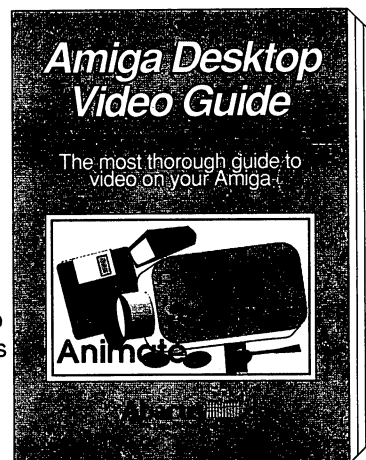
**Amiga desktop Video Guide** is the most complete and useful guide to desktop video on the Amiga. **Amiga Desktop Video Guide** covers all the basics- defining video terms, selecting genlocks, digitizers, scanners, VCRs, camera and connecting them to the Amiga.

Just a few of the topics described in this excellent book:

- The basics of video
- Genlocks
- Digitizers and scanners
- Frame Grabbers/ Frame Buffers
- How to connect VCRs, VTRs, and cameras to the Amiga
- Animation
- Video Titling
- Music and videos
- Home videos
- Advanced techniques
- Using the Amiga to add or incorporate Special Effects to a video
- Paint, Ray Tracing, and 3D rendering in commercial applications

**ISBN 1-55755-057-3. Suggested retail price: \$19.95**

**Companion Diskette not available for this book.**

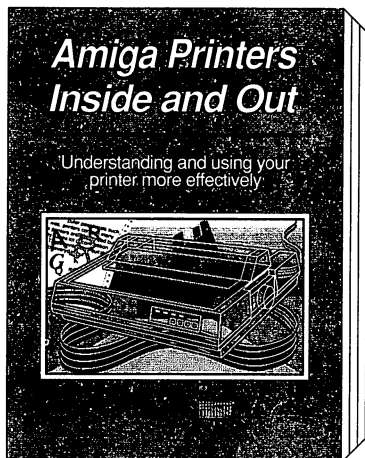


**See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada**

## Amiga Printers: Inside & Out

Vol.#15

Your printer is probably the most used peripheral on your Amiga system and probably the most confusing. Today's printers come equipped with many built-in features that are rarely used because of this confusion. This book shows you quickly and easily how to harness your printer's built-in functions and special features.



Topics include:

- How printers work, and why they do what they do
- Basic printer configuration using the DIP switches
- AmigaDOS commands for simple printer control
- Printing tricks and tips from the experts
- Recognizing and fixing errors
- WORKBENCH Printer drivers explained in detail
- Amiga fonts as printer fonts and much more!



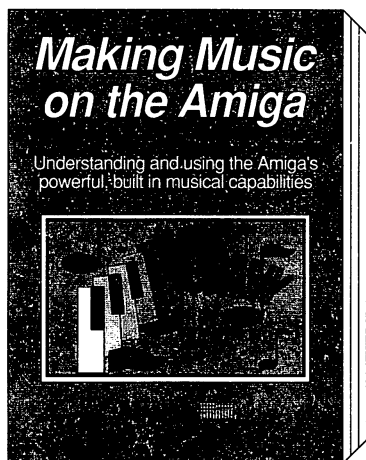
ISBN 1-55755-087-5. Suggested retail price: \$34.95

**Companion Diskette Included** at no additional cost: *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings.*

## Making Music on the Amiga

Vol.#16

The Amiga has an orchestra deep within it, just waiting for you to give the downbeat. **Making Music on the Amiga** takes you through all the aspects of music development on this great computer. Whether you need the fundamentals of music notation, the elements of sound synthesis or special circuitry to interface your Amiga to external musical instruments, you'll find it in this book.



Topics include:

- Basics of sound generation
- Music programming in AmigaBASIC
- Hardware programming in GFA BASIC
- IFF formats (8SVX and SMUS)
- MIDI fundamentals: Concept, function, parameters, schematics and applications
- Digitization: Capture and edit sound, schematics, applications
- Applications: Using Perfect Sound, Aegis Sonix, Deluxe Music Construction Set, Deluxe Sound Digitizer, Audio Master and Dynamic Drums



ISBN 1-55755-094-8. Suggested retail price: \$34.95

**Companion Diskette Included** at no additional cost: *Contains public domain sound sources in AmigaBASIC, C, GFA BASIC and assembly language.*

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

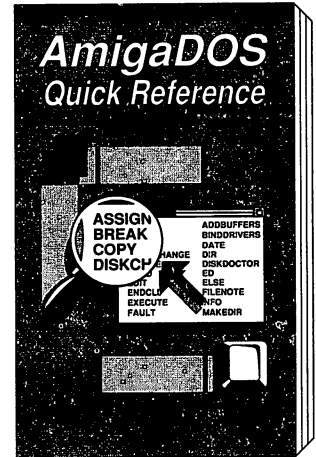
# AmigaDOS Quick Reference

**AmigaDOS Quick Reference** is an easy-to-use reference tool for beginners and advanced programmers alike. You can quickly find commands for your Amiga by using the three handy indexes designed with the user in mind. All commands are in alphabetical order for easy reference. The most useful information you need fast can be found including:

- All AmigaDOS commands described with examples including Workbench 1.3
- Command syntax and arguments described with examples
- CLI shortcuts
- CTRL sequences
- ESCape sequences
- Amiga ASCII table
- Guru Meditation Codes
- Error messages with their corresponding numbers

Three indexes for instant information at your fingertips! The **AmigaDOS Quick Reference** is an indispensable tool you'll want to keep close to your Amiga.

**ISBN 1-55755-049-2. Suggested retail price: \$9.95**  
**Companion Diskette not available for this book.**



## Abacus Amiga Book Summary

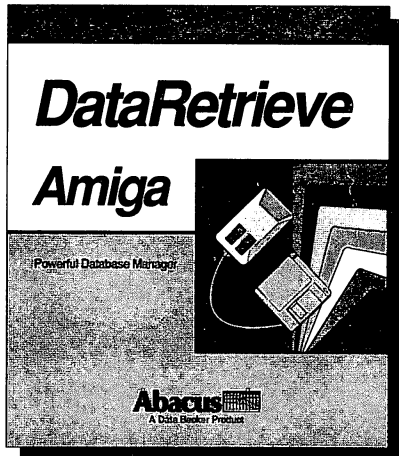
Vol.1	Amiga for Beginners	1-55755-021-2	\$16.95
Vol.2	AmigaBASIC: Inside and Out	0-916439-87-9	\$24.95
Vol.3	Amiga 3D Graphic Programming in BASIC	1-55755-044-1	\$19.95
Vol.4	Amiga Machine Language	1-55755-025-5	\$19.95
Vol.6	Amiga System Programmers Guide	1-55755-034-4	\$34.95
Vol.7	Advanced System Programmers Guide	1-55755-047-6	\$34.95
Vol.8	AmigaDOS: Inside and Out	1-55755-041-7	\$19.95
Vol.9	Amiga Disk Drives: Inside and Out	1-55755-042-5	\$29.95
Vol.10	'C' for Beginners	1-55755-045-X	\$19.95
Vol.11	'C' for Advanced Programmers	1-55755-046-8	\$24.95
Vol.13	Amiga Graphics: Inside & Out	1-55755-052-2	\$34.95
Vol.14	Amiga Desktop Video Guide	1-55755-057-3	\$19.95
Vol.15	Amiga Printers: Inside & Out w/ disk	1-55755-087-5	\$34.95
Vol.16	Making Music on the Amiga w/disk	1-55755-094-8	\$34.95
Vol.17	Best Amiga Tricks & Tips w/ disk	1-55755-107-3	\$29.95
	AmigaDOS Quick Reference	1-55755-049-2	\$9.95

**See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada**

# DataRetrieve

## A Powerful Database Manager for the Amiga

Imagine a powerful database for your Amiga: one that's fast, has a huge data capacity, yet is easy to work with. Now think **DataRetrieve**. It works the same way as your Amiga- graphic and intuitive, with no obscure commands. Quickly set up your data files using convenient on-screen templates called masks. Select commands from the pull-down menus or time-saving shortcut keys. Customize the masks with different text fonts, styles, colors, sizes and graphics. If you have any questions, Help screens are available at the touch of a button. **DataRetrieve** is the perfect database for your Amiga.



### Features include:

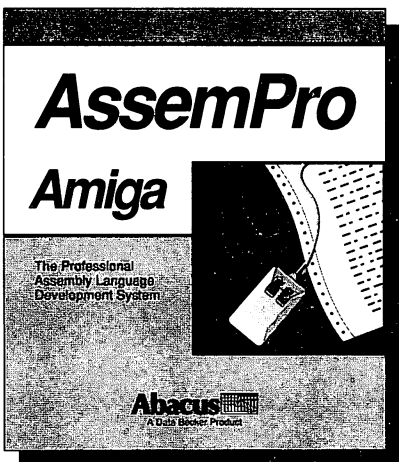
- Enter data into convenient screenmasks
- Work with 8 databases concurrently
- Define different field types: text, date, time, numeric and selection
- Customize 20 function keys to store macro commands and text
- Specify up to 80 index fields for superfast access to your data
- Perform simple or complex data searches
- Create subsets of a larger database for even faster operation
- Exchange data with other packages: form letters, mailing lists
- Produce custom printer forms: index cards, labels, Rolodex® cards, etc. Adapts to most dot-matrix and letter-quality printers
- Protect your data with passwords
- Get Help from online screens
- Not copy protected

**Suggested retail price: \$79.95**

# AssemPro

## Assembly Language Development System for the Amiga

**AssemPro** also has the professional features that advanced programmers look for. Lots of "extras" eliminate the most tedious, repetitious and time-consuming machine language programming tasks. Like syntax error search/replace functions to speed program alterations and debugging. And you can compile to memory for lightning speed. The comprehensive tutorial and manual have the detailed information you need for fast, effective programming.



### Features include:

- Integrated editor, debugger, disassembler and reassembler
- Large operating system library
- Runs under CLI and Workbench
- Produces either PC-relocatable or absolute code
- Create custom macros for nearly any parameter
- Error search and replace functions
- Menu-controlled conditional and repeated assembly
- Full 32-bit arithmetic
- Advanced debugger with 68020 single-step emulation
- Fast assembly to either memory or disk
- Written entirely in machine language
- Runs on any Amiga with 512K or more

**ISBN 1-55755-030-1. Suggested retail price: \$99.95**

*Machine language programming requires a solid understanding of the Amiga's hardware and operating system. We do not recommend this package to beginning Amiga programmers.*

**See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada**

# TextPro

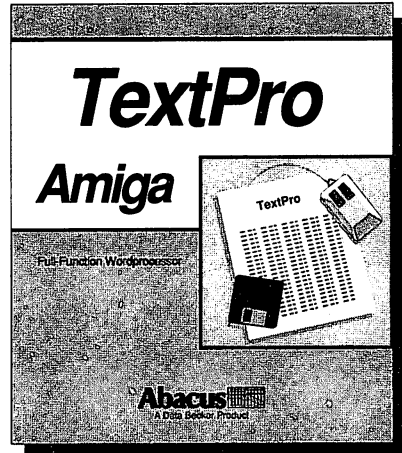
## The Ideal Word Processing Package for the Amiga

**TextPro** is an full-function word processing package that shares the true spirit of the Amiga: easy to use, fast and powerful, with a surprising number of "extra" features. You can write your first **TextPro** documents without even reading the manual. Select options from the pulldown menus with your mouse, or use the time-saving shortcut keys to edit, format and print your documents. **TextPro** sets a new standard for word processors in its price range. Easy to use, packed with advanced features- it's the ideal package for all of your wordprocessing needs.

### Features include:

- Fast editing and formatting on screen
- Display bold, italic, underline, superscript and subscript characters
- Select options from dropdown menus or handy shortcut keys
- Automatic wordwrap and page numbering
- Sophisticated tab and indent options, with centering and margin justification
- Move, Copy, Delete, Search and Replace options
- Automatic hyphenation
- Customize up to 30 function keys to store often-used text, macro commands
- Merge IFF format graphics into your documents
- **BTSnap**- program for saving IFF graphics from any program
- Load and save files through RS-232 port
- Flexible, ultrafast printer output- drivers for most popular dot-matrix and letter quality printers included

ISBN 1-55755-027-1. Suggested retail price: \$79.95



# BeckerText

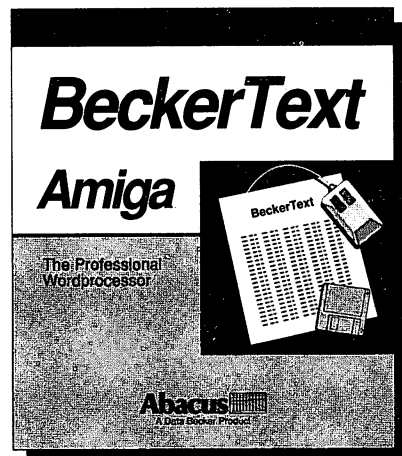
## Professional Word Processing Package for the Amiga

**BeckerText** is more than just a word processor. Merge sophisticated IFF-graphics anywhere in your document. Hyphenate, create indexes and generate a table of contents for your documents, automatically. And what you see on the **BeckerText** screen is what you get when you print the document- real WYSIWYG formatting on your Amiga. Print up to 5 columns per page. Includes built-in spell checker, automatic table of contents, index generation, "fill-in-the-form" templates, math calculations, programmable function keys and more.

### Features include:

- Select options from pulldown menus or handy shortcut keys
- Bold, italic, underline, superscript and subscript characters
- Automatic wordwrap and page numbering
- Sophisticated tab and indent options, with centering and margin justification
- Move, Copy, Delete, Search and Replace
- Write up to 999 characters per line with horizontal scrolling
- Check spelling as you write or interactively proof document; add to dictionary
- Customize 30 function keys to store often-used text and macro commands
- **BTSnap**- program for converting text blocks to IFF graphics
- C-source mode for quick and easy C language program editing
- Adapts to virtually any dot-matrix, letter-quality or laser printer
- Comprehensive tutorial and manual
- Not copy protected

Suggested retail price: \$150.00

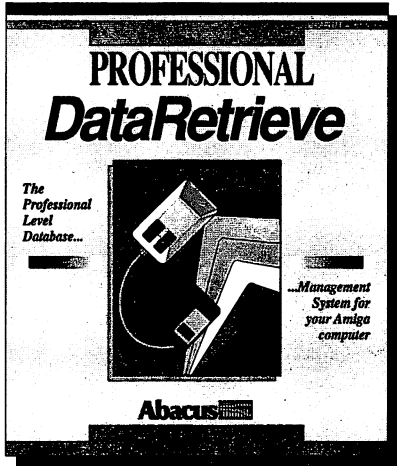


See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

# Professional DataRetrieve

Professional level Database Management System for your Amiga

**Professional DataRetrieve** has complete relational data management capabilities. Define relationships between different files (one to one, one to many, many to many). Change relations without file reorganization. Includes an extensive programming language which includes more than 200 BASIC-like commands and functions and integrated program editor. Design custom user interfaces with pulldown menus, icon selection, window activation and more. Perform calculations and searches using complex mathematical comparisons using over 80 functions and constants.



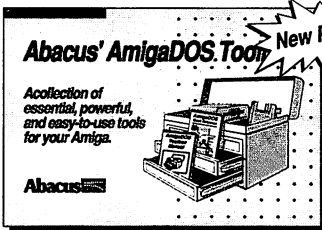
## Professional DataRetrieve's features:

- Maximum size of a data field 32,000 characters (text fields only)
- Maximum number of data fields limited by RAM
- Maximum record size of 64,000 characters
- Maximum number of records disk dependant (2,000,000,000 max.)
- Up to 80 index fields per file
- Up to 6 field types - Text, Date, Time, Numeric, IFF, Choice
- Up to 8 files can be edited simultaneously
- Unlimited number of searches and sub-range criteria
- Integrated list editor and full-page printer mask editor
- Index accuracy selectable from 1-999 characters
- Multiple file masks on-screen
- User-programmable pulldown menus
- Operate the program from the mouse or from the keyboard
- IFF Graphics supported
- Mass-storage-oriented file organization

ISBN 1-55755-028-X. Suggested retail price: \$295.00

## AmigaDOS Toolbox

**Abacus' AmigaDOS Toolbox** has the tools you need to make your Amiga computing easier and more productive. Whether you are a beginner or an advanced Amiga user, you'll find the AmigaDOS Toolbox to be just what you need.



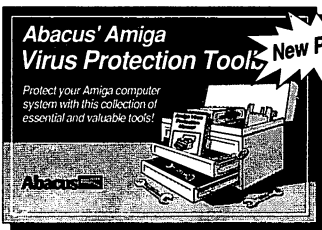
Some of our best tools included are:

- **DeepCopy**- one of the fastest FULL disk copiers
- **Speeder**- a data speedup utility (more than 300%) not a disk cache
- **BTSnap**- a screen grabber deluxe
- **Diskmon**- a full-featured disk editing tool
- **Fonts**- eleven new originals you can use in your Amiga text and many additional tools that every Amiga owner can use.

ISBN 1-55755-053-0. Suggested retail price: \$39.95

## Amiga Virus Protection Toolbox

The **Virus Protection Toolbox** describes how computer viruses work; what problems viruses cause; how viruses invade the Libraries, Handler and Devices of the operating system; preventive maintenance; how to cure infected programs and disks. Works with Workbench 1.2 and 1.3! Tools included are:



- **Boot Check**- to prevent startup viruses
- **Recover**- to restore the system information to disk
- **Change Control Checker**- to record modifications to important files
- **Check New**- to identify new program and data files

ISBN 1-55755-055-7. Suggested retail price: \$39.95

★★★★★ Five Star Rating- *Info Magazine*

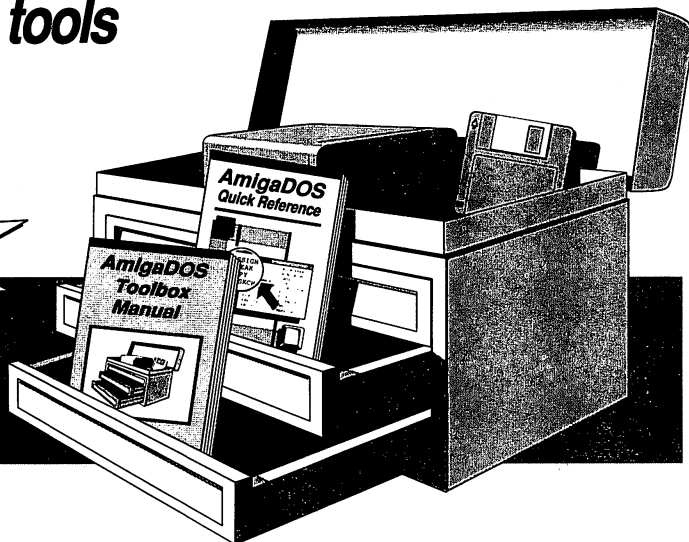
See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

..... Presenting .....

# Abacus' AmigaDOS<sup>®</sup> Toolbox

*A collection of essential, powerful  
and easy-to-use tools  
for your Amiga.*

**New Price!**



**Abacus' AmigaDOS Toolbox** has the tools you need to make your Amiga computing easier and more productive. Whether you are a beginner or an advanced Amiga user, you'll find the AmigaDOS Toolbox to be just what you need.

Some of our best tools included are:

- **DeepCopy** - one of the fastest FULL disk copiers
  - **Speeder** - a data speedup utility (more than 300%) not a disk cache
  - **BTSnap** - a screen grabber deluxe
  - **Diskmon** - a full-featured disk editing tool
  - **Fonts** - eleven new originals you can use in your Amiga text
- ...and many additional tools that every Amiga owner can use.

ISBN 1-55755-053-0. Suggested retail price: \$39.95

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

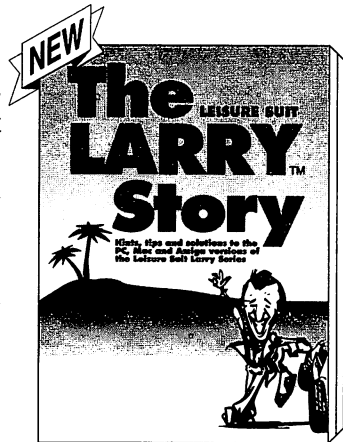
### **The Leisure Suit Larry Story**

Full of game hints, tips and solutions to the mis-adventures of Leisure Suit Larry series from Sierra On-Line. Complete solutions to Land of the Lounge Lizards, Looking for Love, and Passionate Patti. Complete coverage of PC, Amiga, ST, and Macintosh versions. With this book you'll do more than just play the game; you'll live it!

160 pp. Available Now.

ISBN 1-55755-086-7. \$14.95

Canada: 54382 \$19.95



### **Take Off With Microsoft Flight Simulator**

Teaches you quickly and easily the techniques of operating the Flight Simulator to it's fullest. Learn about turns, climbing, diving, takeoffs with crosswinds, landing without engines, navigating and utilizing the autopilot, formation flying and multi-player mode. All the necessary instructions you need to become an experienced PC pilot.

300 pp. Available Now

ISBN 1-55755-089-1. \$16.95

Canada: 54383 \$22.95



To order direct call Toll Free 1-800-451-4319

In US and Canada add \$4.00 shipping and handling. Foreign orders add \$12.00 per item.

Michigan residents add 4% sales tax.





Fold

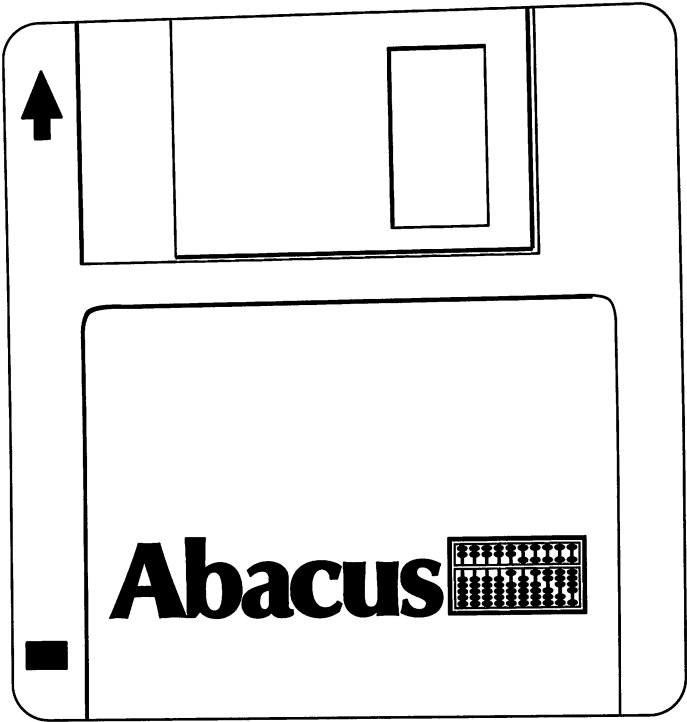
**Abacus** 

Place  
stamp  
here

Abacus  
5370 52nd Street SE  
Grand Rapids MI 49512

Fold

# Companion Diskette Enclosed



## Book/companion diskette packages:

- Save hours of typing in source code from the book.
- Provide complete source code ready-to-compile and executable codes; help avoid printing and typing mistakes.
- The companion diskette contains all of the programs listed in this book..

If you bought this book without a diskette, call us today to order an economical companion diskette and save yourself valuable time.

# Abacus

5370 52nd Street SE • Grand Rapids, MI 49512  
Call 1-800-451-4319

## Companion diskette contents

### Chapter 2 (dir)

2.3 Startup-Sequences (dir)

2.4.3\_Printer-Spooler (dir)

### Chapter 3 (dir)

3.2.1\_Draw\_modes (dir)

3.2.2\_Style (dir)

3.2.3\_Move (dir)

3.2.4\_FAST-GFX.Amiga (dir)

3.2.5\_BRUSH-TRANSFORMER (dir)

3.2.6\_FLOOD+WindowManip. (dir)

3.3 Fading (dir)

3.4 3D Vector Graphics (dir)

3.5 Fonts (dir)

3.6 PRINT (dir)

### Chapter 4 (dir)

4.1 Input Gadgets (dir)

4.2 Rubberbanding (dir)

4.3 DualBitMap (dir)

### Chapter 5 (dir)

5.1 File Monitor (dir)

5.2 BASIC structure (dir)

5.3 Utility Programs (dir)

5.3.1 Data Generator (dir)

5.3.2 Cross Reference (dir)

5.3.3 BlankLine (dir)

5.3.4 REMarks (dir)

5.3.5 Variables (dir)

5.3.7 Modification (dir)

### Chapter 7 (dir)

7.2.6 IconAnalyzer

7.3.3 Iconeditor

### Chapter 8 (dir)

8.2.1A FileTestBASIC

8.2.1B FileTestDOS

8.2.1C Requester

8.2.2 Input.rev

8.3.1Pulldowns

### Chapter 9 (dir)

9.1 Division by Zero (dir)

9.2 Virus Alert! (dir)

9.3 ML and BASIC (dir)

### Chapter 10 (dir)

Direct disk access

Memhandler

Printer-Data

### Chapter 12 (dir)

. IconInstall

WindowTitle

### Chapter 13 (dir)

. ieee-library.bas

### bmaps (dir)

diskfont.bmap

dos.bmap

exec.bmap

graphics.bmap

ieee-library.bas

intuition.bmap

mathieeedoubtrans.bmap

ZZZ



# The <sup>Best</sup> Amiga<sup>®</sup> Tricks & Tips

The **Best Amiga Tricks & Tips** is a great collection of Workbench, CLI and BASIC programming "quick-hitters", hints and application programs. You'll be able to make your programs more user-friendly with pulldown menus, sliders and tables. BASIC programmers will learn all about gadgets, windows, graphic fades, HAM mode, 3-D graphics and more.

The **Best Amiga Tricks & Tips** includes a complete list of BASIC tokens and multitasking input and a fast and easy print routine. If you're an advanced programmer, you'll discover the hidden powers of your Amiga.

The **Best Amiga Tricks & Tips** will teach you how to allocate memory - trap errors - use Amiga fonts - mix machine language with BASIC - write your own computer virus checker - disable fast RAM - upgrade to a faster processor - use the NewCon and Pipe devices - access machine language and C programs from AmigaBASIC - find "secret" messages built into the Amiga's operating system. You'll learn how to utilize 3D programming and fading graphics - text input and output - BASIC benchmarks (speed tests) - vector graphics - multitasking input - analyze files - write self-modifying programs — all this and more.

US \$29.95

ISBN 1-55755-107-3



9 781557 551078

## A valuable collection of useful and productive hints for using your Amiga

Topics include:

- Using the new AmigaDOS, WorkBench and Preferences 1.3 and Release 2.0
- Tips on using the new utilities on Extras 1.3
- Customizing Kickstart for Amiga 1000 users
- Enhancing BASIC using ColorCycle and mouse sleeper.
- Disabling Fast RAM and disk drives
- Using the Mount command
- Writing an Amiga virus killer program
- Changing type-styles
- Learn kernal commands
- BASIC benchmarks
- Disk drive operations and disk commands
- Learn machine language calls

The **Best Amiga Tricks & Tips** is packed with dozens of hints and applications — for all Amiga owners.

**Abacus** 

5370 52nd Street SE • Grand Rapids, MI 49512

Amiga is a registered trademark of Commodore-Amiga Inc.