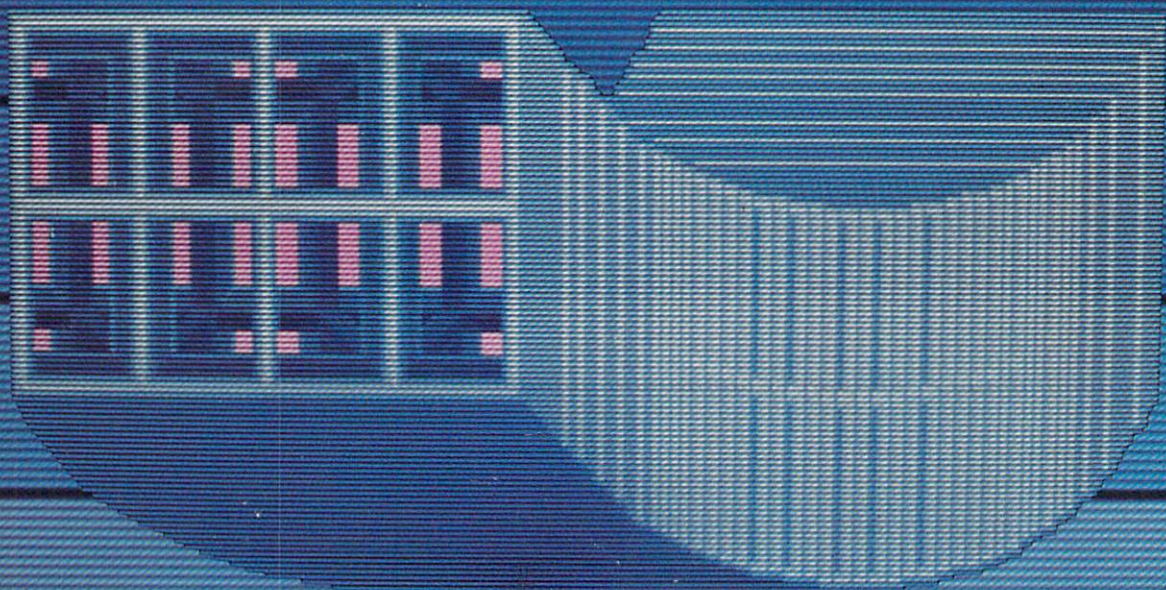


**The**

**Amiga Microsoft BASIC**

**Programmer's Guide**



**William B. Sanders**





**The**  
**Amiga Microsoft BASIC**  
**Programmer's Guide**





**The**  
**Amiga Microsoft BASIC**  
**Programmer's Guide**

**William B. Sanders**

**Scott Foresman and Company**  
Glenview, Illinois • London

Amiga ASCII characters reprinted by permission of Commodore-Amiga, Inc.  
Cole Porter's *Anything Goes* used by permission of Warner Bros. Inc.

Microsoft BASIC is a trademark of Microsoft Corporation.  
Commodore Amiga is a registered trademark of Commodore Business  
Machines, Inc.  
IBM is a registered trademark of the International Business Machines  
Corporation.  
Apple is a registered trademark of Apple Computer, Inc.  
Atari is a registered trademark of Atari Corporation.

#### **Library of Congress Cataloging-in-Publication Data**

Sanders, William B., 1944-  
The Amiga Microsoft BASIC programmer's guide.

Includes index.

1. Amiga (Computer)—Programming. 2. BASIC  
(Computer program language) I. Title.  
QA76.8.A46S26 1987 005.265 86-17867

1 2 3 4 5 6 KPF 91 90 89 88 87 86

ISBN 0-673-18523-0

Copyright © 1987 William B. Sanders.  
All Rights Reserved.  
Printed in the United States of America.

#### **Notice of Liability**

The information in this book is distributed on an "As Is" basis, without warranty. Neither the author nor Scott, Foresman and Company shall have any liability to customer or any other person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by the programs contained herein. This includes, but is not limited to, interruption of service, loss of data, loss of business or anticipatory profits, or consequential damages from the use of the programs.

Scott, Foresman Professional Publishing Group books are available for bulk sales at quantity discounts. For information, please contact Marketing Manager, Professional Publishing Group, Scott, Foresman and Company, 1900 East Lake Avenue, Glenview, IL 60025.

# CONTENTS

<b>1</b>	<b>Introduction to Basic Programming</b>	<b>1</b>
	Welcome to Programming	2
	A Little Philosophy	4
	Loading BASIC	5
	LIST and OUTPUT Windows	6
	Statements, Functions and Commands	8
	Working with Text and Numbers	12
	Editing Programs	13
<b>2</b>	<b>Starting Operations</b>	<b>19</b>
	Fundamental Formats: Commas, Semicolons and Colons	19
	Tabs, REMs, Ticks and Width	21
	Variables	24
<b>3</b>	<b>Math Operations</b>	<b>35</b>
	Sequential Calculations	35
	Precedence	37
	Math Functions	39
<b>4</b>	<b>Sequential Modular Program Organization</b>	<b>47</b>
	Top Down Programming	48
	Getting Things in Order	49



Interactive Data Entry	51
Getting Information to the Screen Clearly	58
READ That DATA	60
Breaking down Large Problems into Small Problems	62

**5****Loops 67**

The Loop Structure	67
FOR/NEXT Loops	68
Counters	72
WHILE/WEND Loops	73

**6****Branch Structures 77**

Branching	78
Computing with Relationals	82
Subroutines	87
Computed GOTO and GOSUB	88
Strings and Relationals	92
Subprograms	93

**7****Arrays 99**

Arrays as Grouped Variables	99
The DIMension of an Array	101
Keeping in Bounds	102
Buffers and Arrays	104
Multi-DIMensional Arrays	106
Arrays in Subprograms	111

**8****Manipulating Strings 115**

Substrings: Parts and Whole	115
Formatting Strings	116
Time after TIME\$	120
String Searching	122
Converting between String and Numeric Variables	124

---

<b>9</b>	<b>Preparing Data and Formatting Output</b>	<b>129</b>
	Formatting and Manipulating Information	129
	Screen Placement	130
	PRINT USING: Formatting with Style	132
	Highlighting Output with COLOR	135
	How about a DATE\$	137
<b>10</b>	<b>Of Mice and Menus</b>	<b>141</b>
	The Mouse Input	141
	Pull down Menus	141
	Setting up Your Menus	142
	Toggling Menus	145
	Refreshing the Menu	149
<b>11</b>	<b>Screen Control</b>	<b>151</b>
	Scroll Management	151
	Finding the Cursor with POS(0) and CSRLIN	152
	Yes—The Amiga Does Do Windows	155
	Screen Work	159
<b>12</b>	<b>Drawing with Graphics</b>	<b>163</b>
	Pixels	163
	The Amiga's COLORful Palette	164
	Multiple Colors	166
	Getting in Shape	168
	Plots from Last Plot: Relative Plots	174
	Amiga Art	176
	PAINT	176
	Getting Around: CIRCLE	177
	Filling an AREA with PATTERN	179

<b>13</b>	<b>Animation, Sprites and Bobs</b>	<b>185</b>
	Moving Graphics	185
	Moving out with PUT and GET	187
	Mouse Control	193
	Sprites and Bobs	197
	Using ObjEdit	197
	Displaying Sprites and Bobs with OBJECT	198
	Moving Sprites and Bobs	200
	Managing Crashes with COLLISION	202
	Making Bobs	205
<b>14</b>	<b>Sound, Music and Voice Synthesizing</b>	<b>209</b>
	Amiga Sounds	209
	The SOUND Statement	209
	Amiga Music	212
	More WAVE Work	217
	Speech Synthesis	219
	Translating Text to Talk	219
	The SAY Array	220
	Writing Like It Sounds: Phonetic Transcription	222
<b>15</b>	<b>The Disk System and Sequential Files</b>	<b>227</b>
	CHAIN Routines	229
	Sequential Files	232
	Making a Sequential File	233
	Formatting Text and Numbers in Files with PRINT # USING	241
<b>16</b>	<b>Random Access Files</b>	<b>247</b>
	Random and Sequential Files: Differences and Similarities	247
	Random File Buffers	249
	Finding and Changing Records	254



---

<b>17</b>	<b>Printer Control</b>	<b>261</b>
	Text Output to the Printer	263
	Decoding with CHR\$	264
	CHR\$ and Printer Control	267
	Using the LPOS(0) Function	271
	Opening LPT1:	272
	LPRINT USING	273
	Control That Format!	274
<b>18</b>	<b>Telecommunications</b>	<b>277</b>
	Modems	277
	Types of Modems	278
	Null Modem	279
	The COM1: File	280
	Reading from and Writing to the COM1 File	281
	Amiga Terminal Program	283
	Using the Amiga Terminal Program	289
<b>19</b>	<b>Algorithms and Advanced Techniques</b>	<b>293</b>
	A Good Algorithm Is Worth a Thousand Lines of Code	294
	The Bubble Sort	294
	The Shell Sort	298
	Rearrangements for Sorts	301
	Artificial Intelligence and IF...THEN	307
	<b>Appendix A: ASCII and Non-ASCII Character Codes</b>	<b>313</b>
	<b>Appendix B: Using CLI Commands</b>	<b>315</b>
	<b>Glossary</b>	<b>327</b>
	<b>Index</b>	<b>347</b>



# Introduction to BASIC Programming

This book is about a programming language called Microsoft® BASIC on the Commodore Amiga computer. You'll learn how to write computer programs that you can use to do everything from writing text on your screen to creating animated graphics. We'll go slowly and systematically so that if you're new to programming you won't be overwhelmed or lost. If you do have experience programming, take a close look at the next section describing the differences between Amiga BASIC and other versions of the language. There are significant differences, but once understood, experienced programmers can jump to different parts of the book describing specific features of Amiga BASIC or how to take advantage of the many unique Amiga characteristics such as pull down menus, multi-tasking and color graphics.

## Amiga Notes

### Important Notice!!

The Amiga was originally shipped with another version of BASIC called "ABasiC." That version was dropped by Amiga and replaced by a much improved version, Microsoft BASIC. If you do not have your copy of the newer BASIC, contact your Amiga dealer and he will give you a copy at no cost.



## WELCOME TO PROGRAMMING

---

For those of you who are new to computer programming, you are in for a real treat and surprise. The image of genius level IQ computer programmers is a misleading stereotype clouded by ignorance of what a programming language is. To some degree learning how to program is like learning a foreign language. However, there are a lot fewer words in programming, and there are no exceptions to the rules. When you learn how to do something in a certain correct way, you will not encounter "exceptions to the rule" or special idioms that will change how things are done. In addition, your Amiga will tell you if you make a mistake and point out what you did wrong. It's like having the most patient teacher in the world helping you learn how to program.

Besides being easier to learn than a foreign language, Amiga BASIC *does* things. It puts pictures on the screen, analyzes data and even generates a voice synthesizer. Using this book, you can learn how to write a computer operated checkbook, make learning games for your children, create animated graphics and develop many other practical and fun applications. You don't have to stick with the examples in the book. Use the techniques you acquire to write programs for your own needs.

In addition, you will be able to do things with your computer you may not ever have thought about. For example, you'll learn how to write a communications program to connect your Amiga to other computers via a modem. (Modems are simply telephones for computers.) You can call up your bank, a financial news service or send a letter across the country. So besides being a tool to use for tasks you do already by some other method, your BASIC programming skills can be used to create new applications that will enhance your life in ways you did not consider when you bought your computer.

### **Throw Caution to the Wind . . .**

The last thing you want to do with your Amiga and programming is to be overly cautious. The mistaken belief that you can wreck a computer by entering the wrong information on the keyboard is silly. You can wipe out information on a disk or knock out work in memory that you've spent a lot of time developing, but you can't hurt your computer. When you're learning how to program, use a separate disk. Label it clearly as your "Experiment Disk," and then try all kinds of things with your computer. If

you wipe out this disk, you won't lose anything important. If you do write a program that is important for you to keep, just make a copy of that program on another disk. This way, even if you wipe out your "Experiment Disk," you still have copies of your important programs.

Everyone makes mistakes in programming, even the most experienced programmers. (In fact, experienced programmers make more mistakes than beginners since they are not afraid to try more things.) Instead of being a reason for not experimenting with your programming skills, however, it is a good reason to have an "insurance policy," which will guarantee that experimentation will not be a cause for losing important data. So, while you should be innovative, do it with a safety net. (By the way, *nobody* ever listens to this advice initially. Everyone loses something important, before they take it seriously.) Just remember, whether you experiment or not, you'll make mistakes. You'll learn more—and have a lot more fun—by trying out different things, so you might as well enjoy yourself. A "back-up" disk with your important programs will protect you.

## Amiga BASIC and Other BASICs

If you're familiar with BASIC programming on other microcomputers, one of the first things you'll notice about the Amiga Microsoft BASIC is the absence of line numbers. If you think of line numbers simply as labels, you'll understand why they are not needed in Amiga's BASIC. Microsoft BASIC on the Amiga uses descriptive labels followed by colons that sit on lines all by themselves. You can use numbers as labels, but it makes a lot more sense for you to use descriptive labels that tell what your program is doing in different sections. For example:

```
PrintStuff:
```

might be used as a label for a section of the program that prints stuff on the screen. It's a lot clearer than something like:

```
200:
```

even though the latter is perfectly correct to use in a program. Otherwise, it is very similar to other BASICs. If you have used the later version of Microsoft BASIC on the Apple Macintosh computer, you will find them to be almost identical. Likewise, with the exception of line numbers, BASICA on the IBM PC is very close to Amiga's version of BASIC.

Experienced BASIC programmers should concentrate on the following features of Amiga BASIC:

1. Pull-down MENU, MOUSE and SCROLL (screen and program control)
2. Object editor, OBJECT.XX words and COLLISION (animated graphics)
3. LIBRARY, PEEKW and POKEW (machine language routines)
4. SAY (voice synthesizer)
5. Subprograms, SUB, EXIT SUB and END SUB (BASIC subprograms)
6. Sound and graphics words if not used in previous BASICs

The above six categories may or may not be in another version of BASIC you've used, but they should be checked just in case. There are a lot of other things about the Amiga that you will discover in this book. Check out the list above, and don't be surprised if there's more than you bargained for!

## **A LITTLE PHILOSOPHY**

---

For beginners and experienced programmers alike there are different attitudes and approaches to programming. It might sound stuffy and "school marmish" to lecture you on one approach or another before we even get started writing programs. We'll try to understand a style that is both experimental and practical in terms of accomplishing a goal and having fun in the process.

Amiga Microsoft BASIC has a certain built-in structure and philosophy that makes programming a lot easier while not taking anything away from creativity. To some, the term "structured programming" evokes images of weighty flowcharts and tedium. That's really not accurate, and anyone who treats structured programming as a set of stiff rules instead of a practical tool misses the point. Basically, all structured programming attempts to do is to organize the programming task. Let's see how Amiga BASIC does it, and what this means to the novice and experienced programmer alike.

**1. TOP DOWN PROGRAMMING.** This kind of program starts at a given point and sequentially executes one task at a time. The alternative is jumping all around the place instead of putting things in sequential



order. This style of programming makes it easier to find errors (or “bugs” as they’re called in programming) and develop good programs. This does not exclude jumping to modules that perform sub-tasks (called sub-routines) or, in Amiga BASIC, using subprograms and then returning to the main sequence. Likewise, a program can branch out in different directions in structured programming.

**2. MODULAR PROGRAMMING.** An old Chinese proverb states that even a journey of a 1000 miles begins with a single step. The same is true in structured programming. If you think of a program as a series of steps to be taken one at a time instead of a giant step to be taken all at once, the task is much easier and has a far greater chance of success. Each part of a program can be seen as a module that does something for the program. The most experienced and smartest programmers do not work up long complex programming formulas to get a job done. Instead, they take a big complex program and break it down into small, manageable chunks that even a beginner could handle. Each of these chunks is called a “module.” The module is then placed into the proper place in the sequence of the program. So instead of having a program of “100 lines,” they have a simpler one of “10 modules.” Later on you’ll learn how to save certain of these modules that are used over and over as “sub-programs.” When you program, you can just reuse these subprograms instead of having to write them all over again.

## LOADING BASIC

---

To get going, put the Kickstart disk in your drive and then the Workbench disk. If you have a single drive, remove the Workbench disk and insert the “Extras” disk with both MBASIC and the Amiga Tutor. (We strongly advise you to make a copy of MBASIC on a blank disk and use it for an “experiment disk.”) If you have a dual drive, place the disk with MBASIC into the external drive. The internal drive is called ‘df0’ and the external drive ‘df1’ in CLI (Command Line Interface), but you can easily get everything cranked up from the workbench environment on your screen. You will see an icon with the “Extras” or whatever name you gave your “Experiment Disk.” Place the pointer on the disk icon and give it a double click with the left mouse button.

When the window opens, you will see an icon labeled ‘AmigaBASIC’. Place the pointer on that image and double-click it. When the window opens you will see the following message:

```
Commodore Amiga BASIC
Version 1.00
Created Oct. 23, 1985
Copyright (c) 1985
by Microsoft Corp.
226048 Bytes free in System
25000 Bytes free in BASIC
```

The version and number of free bytes may vary depending on the amount of memory and the version of BASIC you have, so don't worry if your screen says something a little different. The 226048 bytes free in this System shows how much memory is free in your computer, while the 25000 free bytes in BASIC show how much room you have available to program.

On the right side of your screen is the LIST window. In the upper left hand corner of the LIST window is the cursor, a straight vertical red line. (Your cursor may not be red if you've adjusted the colors, but don't worry about it.) You should see your pointer on the screen somewhere, too. At this time your LIST window is the 'active' one since the word LIST on top of the screen is clear. The other window, labeled BASIC, is dimmed; so you know it's not active now. Every time you start, this is the way your screen should look.

## **LIST AND OUTPUT WINDOWS**

---

Now that you know what things look like at the beginning, let's move from the LIST window to the OUTPUT window. Move your pointer so that it is outside of the LIST window, and give it a click. The word 'Ok' appears on your screen, and the cursor moves to the window labeled BASIC. The LIST label is dimmed, and the BASIC label is clear, so you know that you've switched windows.

Without further ado, put in your first command, FILES. Just type in the word FILES and press the key marked 'Return':

```
FILES <Return>
```

After every entry is completed in the output window, press the Return key. You'll find 'Return' is used more than just about any other key you have. We'll remind you at first to press 'Return' by the notation

<Return>. When you get used to pressing RETURN, you're on your own and we'll take out the reminders. As soon as you pressed RETURN, your disk light came on and all of the files on your disk were written on the screen. You may have been surprised since there's a lot more on your disk than you saw from the workbench on icons. For example, your "Extras" disk only shows five icons, but from BASIC, you saw the following when you used the FILES command:

```
Directory of:[Extras]
[Trashcan]
.info
AmigaBASIC
Amiga Tutor.info
[t]
[BasicDemos]
AmigaBASIC.info
[Tfiles]
Tfiles.info
BasicDemos.info
Disk.info
Trashcan.info
Amiga Tutor
```

You may think you've seen a lot, but there's still more. Each of the filenames in brackets is actually a whole other set of files, called a "subdirectory." To see what's in there, enter the FILES command and the name of the subdirectory in quote marks. For example, type in the following:

```
FILES "BasicDemos" <Return>
```

You'll see so many files that they'll scroll off your screen. If you have two disk drives, and your disk with BASIC on it is in the external drive (df1), you can see what's on the internal drive by typing in:

```
FILES "DF0:" <Return>
```

If your Workbench disk is in the internal drive (df0), you'll recognize some of the files from that disk. Now you should know how to do the following:









1. See all the files on your disk
2. See the files in a subdirectory
3. See the files from either drive




It's not quite as easy as using the workbench and icons, but you can see a lot more of what's on your disk. Also, you've learned to explore what's on your disk. Later on, we'll be saving and loading files you create in BASIC on the disk, but for now, we just want to show you how to find your way around.

To do some more rudimentary explorations, press the *right* mouse button and look at the top of your screen. You'll see four menu headings for BASIC:

Project      Edit                  Run            Windows

To open a menu, hold the right mouse button and place the pointer over the menu heading. You'll see the following:

<u>P</u> roject	<u>E</u> dit	<u>R</u> un	<u>W</u> indows
New	Cut  X	Start  R	Show List  L
Open	Copy  C	Stop  .	Show Output
Save	Paste  P	Continue	
Save As		Suspend  S	
Quit		Trace On	
		Step  T	

These menus are options available for working with programs in BASIC. There are equivalent BASIC commands for these menu selections, and we'll cover them all as we work our way through this book. The  symbols indicate that the same thing can be done by pressing the outlined  on your keyboard and the letter next to it. For example, to show the list window after you have run a program, you can press  and L together rather than doing it from the menu. With practice and experience, you will find which method of giving commands is the most comfortable for you.

## STATEMENTS, FUNCTIONS AND COMMANDS


BASIC, like other programming languages, has certain 'reserved words.' These words are used in programs and from the direct mode. When we use a word like FILES from the output window, it is a *command*

in that it makes something happen as soon as you press RETURN. Other words are used within program lines in the deferred or program mode. These are called *statements*. Finally, there are *functions* that are used with statements that perform some built in or defined operation. Don't be overly concerned with differentiating between statements, functions and commands at first. Their difference will become clearer as you use them. To get going, though, we'll look at examples of each and start programming.

## PRINT Statement

You'll probably use PRINT more than any other statement. It puts both text and numbers on the screen. To see how it works, type in the following in the output window:



```
PRINT "Amiga" <Return>
```

That works fine, and the word 'Amiga' appears on your screen. Now, move over to the LIST window, and key in the same thing. This time when you press RETURN, nothing happens other than the cursor moves down a line. Press the right mouse key, select the Run Menu Bar and click START. (Also try pressing the  and R keys to see what happens.) That will execute your program, clear the screen and print the word 'Amiga' on your screen.

## LIST command

Now that you've written and run your first BASIC program, you will want to learn how to get your LIST window back. From the output window, type in:

```
LIST <Return>
```

Your LIST window will reappear along with your little program. You will be using the LIST command a lot since every time you run a program from the output window, the LIST window disappears. You can also click 'Show List' from the Windows menu or use L to get LIST. Find which way is easiest for you; one method of LISTing your program is as good as another. (If you run a program from the LIST window using the pointer or R, your LIST window will automatically return after you've run the program. When you're developing a program and have to add more to it,

it's a good idea to run the program from the LIST window to save a few steps in getting back to it.)

One more command you'll use a lot is NEW. The NEW command removes the BASIC program currently in memory. Since you'll be writing many programs, you'll want to get the old one out of the way before starting a new one. So just type in NEW and press RETURN or click 'New' from the Project window. As soon as you do that you'll get the following message box:

```
Current program is not saved          YES
Do you want to save it before proceeding?  NO
                                           CANCEL
```

Click NO unless you have written a program that's important to you. Now when you start writing a different program, your LIST window will be clear and ready to go.

## The SQR Function

Now let's take a look at a function. Functions are used with statements or commands to perform some operation. For example, on just about any calculator you can find the square root function with the ( $\sqrt{\quad}$ ) symbol on the key. Your Amiga has no such key, but BASIC has a built-in square root function you can use to get the same results. Pop back into the LIST window by placing the pointer in the window and clicking it. (If there's anything in the LIST window, get rid of it with NEW.) Combining the PRINT statement with the SQR, type in the following:

```
PRINT SQR(4)
```

Run the program using any of the three methods you've learned, and you'll get a '2' in the output window. Since 2 is the square root of 4, you've probably guessed that the square root of any number that is in parentheses will be found by the SQR function. Go back to the LIST window and add some more lines with different values in the parentheses to generate any square root you want.

As we mentioned above, don't worry whether a reserved word is a statement, function or command. Just experiment with them to see how they work together, and you'll catch on to what each is and what it does.

In certain cases you'll use a statement like a command, and commands like statements. For example, from the Output Window, if you enter:

```
PRINT "Wackawacka do"
```

and hit RETURN, that banal message will be duly printed on your screen. Thus, PRINT works just like a command from the Output Window. Conversely, if you put in LIST as your last program statement, after you RUN the program (even from the Output Window), your program will be listed on the screen. In this case, LIST works just like a statement. Functions can work in either the LIST or Output Windows when combined with statements. So if you want to do a quick calculation from the Output Window, you can. For example, if you type in:

```
PRINT SQR(55) <Return>
```

you'll get the result (7.416198) immediately. When you do something like that from the Output Window, it's called the "Direct" or "Immediate" mode. The same thing written in the LIST Window and then run is referred to as the "Deferred" or "Program" mode. The direct mode is good for testing little routines to see quickly what the results are going to look like and finding bugs in the program.

## Amiga Notes

### Secret Shortcut

One of the best kept secrets from beginners is the substitution of the question mark for the PRINT statement. If you put a question mark (?) where you would normally place a PRINT statement, it will work just like the PRINT statement. From the LIST Window, type in:

```
? "What happened?"
```

As soon as you press RETURN, the message changes to:

```
PRINT "What happened?"
```

You can see the only question mark that changed was the one that substituted for the PRINT statement. The second question mark that you wanted in your message stayed the way it was. So, instead of having to type five keys for PRINT, you only need one.

## WORKING WITH TEXT AND NUMBERS

---

There are two distinct formats with PRINT. The first involves text, as we have seen, and whatever is placed between quotation marks appears on the screen when the program is run. Anything you put (except more quotation marks) between the quotes gets printed. For example:

```
PRINT "2345DFGSRT+=Y3456WYN{P}Q234ASDF"
```

places 2345DFGSRT+=Y3456WYN{P}Q234ASDF on the output window's screen.

The PRINT statement works differently with numbers. In fact, it works pretty much like a calculator. You saw with the SQR example that PRINT can work with functions. It works the same way with numbers. For example, write in the following program and run it:

```
PRINT 10 + 20
```

There are no quote marks; only numbers and the plus sign are used. When you run the program, instead of printing '10 + 20' on the screen, you get the sum of the two numbers, 30. The plus (+) sign is an operator. You can also do other mathematical operations with numbers and PRINT. Here's a list of the operations you can do and the associated keys.

- |                     |      |
|---------------------|------|
| 1. Addition         | +    |
| 2. Subtraction      | -    |
| 3. Multiplication   | *(X) |
| 4. Division         | /(:) |
| 5. Modulo           | MOD  |
| 6. Exponentiation   | ^    |
| 7. Integer Division | \    |

The addition and subtraction symbols are standard, but an asterisk is used instead of an  $\times$  for multiplication, and a slash is used instead of a division sign. MOD finds the 'remainder' in division. For example, PRINT 10 MOD 8 results in 2 since 8 goes into 10 one time with a remainder (modulo) of 2. The exponentiation refers to the "power of" expression in math. For example, if you want to write two to the third power, you write,  $2^3$ , but on your Amiga, you type in  $2 \wedge 3$ . Finally, integer division uses the



---

opposite type of slash (located in the upper right hand corner of your keyboard). Integer division returns only whole numbers with no fractions. For example  $9 \setminus 4$  returns 2 while  $9/4$  returns 2.25.

From the direct mode, try the following to get a feel for printing numbers and doing calculations:

```
PRINT 15 MOD 4
PRINT 10 * 8
PRINT 88/11
PRINT 3 ^ 2
PRINT 10 * 5 + 4 - 2
PRINT 15 \ 7 (Note direction of slash!)
```

As you can see from the last example, you can put a lot on a single line, both in the direct and program modes.

## EDITING PROGRAMS

---

Everyone who writes programs makes mistakes. Even the best-planned and structured program will have some typing or syntax errors. Rewriting programs from the beginning because of one little error is very time consuming, so you should learn how to use your editor function immediately for correcting mistakes. There are also a lot of shortcuts you can take with your editor to speed up the process of writing programs. We'll finish this chapter by examining all of the tricks you can do with your editor on the Amiga.

### Amiga Editing

If you're used to other computer program editors, you may find the one in Amiga Microsoft BASIC to be very different.

Once you become accustomed to working with it, though you'll appreciate all of its features.

All editing is done in the LIST Window using the mouse and keyboard. To get started, write the following error-riddled program:

```
StartHere:
  PRINT "Amiga Edditor Work"
  PRINT 5 x 5
```

If you run this program, you'll get the following results in your output window:

```
Amiga Edditor Work
5  0  5
Ok
```

Right away you can see that the word 'editor' is misspelled, and that '5 0 5' doesn't make any sense at all. (Assume you wanted to multiply 5 times 5 and you accidentally used the "x" instead of the asterisk. In the next chapter, you will see why you got the zero between the two 5s.)

To fix your program, place the pointer back in the LIST Window and click the mouse. Now you're ready to edit. First, to take care of the misspelled 'Edditor,' place the pointer so that it is to the right of either 'd' and click the left mouse button. (Most editing is done with the left mouse button; so from now on we'll assume, unless otherwise stated, that a reference to 'click' means press the left mouse button.) The cursor appears to the right of the 'd.' Press the BACKSPACE key once to remove the 'd.' Now the word 'Editor' appears spelled correctly.

Next, to use a different technique to do the same thing, find the four arrow keys to the left of the numeric key pad on the right side of your keyboard. Press the arrow keys until you position the cursor to the right of the 'x' between the two 5s. Remove the 'x' by pressing the BACKSPACE key, and then replace it with an asterisk (\*). If you can do that, you've learned about ninety percent of what you will be doing with your editor. Basically, most of the repair work you'll be doing with the editor is fixing up little mistakes and syntax errors. However, there are many more tricks you can do, so let's look at them.

## Cut, Copy and Paste

Sometimes, you'll want to remove, change or duplicate whole lines of your program. To do that you *hold down* the left mouse button and pull the cursor over the lines you want to duplicate. The part selected becomes inverted or highlighted (the letters are black and the background is orange or red). Click the right mouse button to pull down the Edit Window and select either Copy or Cut. If you select Copy, your line will stay there and a copy of the line will be placed in memory. A Cut selection will remove the line from the screen but keep it in memory.

### Amiga Notes


#### Be careful!

If you press BACKSPACE while a segment of a program is highlighted, it will be shuffled off to silicon oblivion. A Cut removes the segment from the screen but keeps it stored in memory, so you can get it back with Paste. A BACKSPACE creams it for good (or for BAD)! Go ahead and try it with a segment to see what happens.

Once you have Cut or Copied a segment, you can Paste it elsewhere in the program or add a duplicate copy of it. Just click Paste from the Edit Window. For practice, select the line:

```
PRINT "Amiga Editor Work"
```

and make ten copies of it. Then run the program to see what happens. If you want, after you've made the first copy, select two lines together and then Paste each two at a time.

With Cut, Copy and Paste, it might be easier to use the  key in conjunction with 'X' for Cut, 'C' for Copy and 'P' for Paste. Then you can work with both hands and save a little time. (Of course if you have trouble chewing gum and programming at the same time, as I do, you may not want to get too complicated with coordinated manipulations right away!) If you want to select a big section from your program, instead of dragging the cursor over an area it might be easier to click the mouse at the beginning of the segment you want and then do a SHIFT-click at the end of the segment. For example, place the cursor on the beginning of the program currently in your LIST Window and click the mouse. Then move the cursor to the very end of the program and press the SHIFT key and click the mouse. The whole program is now highlighted. Also, by placing the cursor at the far left of a line you want to select, holding down the mouse button and moving the mouse downwards, you can easily grab a whole line. The thing to do is to PRACTICE, EXPERIMENT and have some fun trying out your editor. The more you experiment now, the more time you'll save in the long run. Try some of the following exercises:

1. Using your editor, write the following program:

```
PRINT "My name is <put your name here>."
PRINT "My name is <put your friend's name here>."
PRINT "My name is <put your dog's name here>."
PRINT "My name is <put your bookie's name here>."
```

2. Write this line and then get rid of it by both the Cut method and select and BACKSPACE method.

```
PRINT "I love doing homework!!!"
```

3. Using your editor rearrange the first program to look like the second:

```
PRINT "What is 5 + 34?"
PRINT 5 + 34
PRINT "how old are you?"
PRINT 35
```

```
PRINT "How old are you?"
PRINT 35
PRINT "What is 5 + 34?"
PRINT 5 + 34
```

A final editing trick we'll discuss at this point is the use of the SHIFT and ALT keys with the arrow keys. As your programs get longer, they will fill up your LIST Window. To see different parts of your program, it is necessary to scroll down the LIST Window using either the mouse, holding down the left button, or using the arrow keys. If you want to jump a "page," hold down the SHIFT key and press the up or down arrow key. Similarly, to jump a horizontal page, do the same thing with the left and right arrow keys. Type in the following line:

```
PRINT "This will be an extremely long message that will cover
several horizontal 'pages' of your LIST Window."
```

We'll use the above line to show you horizontal scrolling and how to jump whole pages in a single bound. Use your arrow keys to place the

cursor to the left of the PRINT statement. Now, press and hold the SHIFT key and at the same time press the right arrow key. Do that until you have read the whole line. Do the opposite to get back to the beginning of the line. If you do the same thing with the ALT key instead of the SHIFT key, you can jump to the beginning or end of the line. Try jumping back and forth with the ALT and arrow keys. As your programs get bigger, you'll appreciate this shortcut more and more.

## Changing Window Sizes

Since the LIST Window takes up only half of your screen, program lines will quickly scroll and segment so that you can only see part of a program line at a time. To change the horizontal and vertical size of the LIST Window, place the pointer on the top bar and, holding the left mouse button down, drag the window to your left. Then place the pointer in the lower right corner box of the window and pull (hold the button and drag) the window to your right. That will let you see more of your program as you write it. Leave a little space on the left side of your screen so that you can place the cursor into the Output Window if you want. Just remember, Commodore made the Amiga flexible so that it could fit your needs.



---

# CHAPTER 2

## Starting Operations

---

### FUNDAMENTAL FORMATS: COMMAS, SEMICOLONS AND COLONS

---

In Chapter 1 you learned how to print letters and numbers on your screen; now we will look at the first step in formatting what you print in the Output Window. Write the following program and run it:

```
StartHere:
  PRINT 1;2;3;4;5
  PRINT 1,2,3,4,5
  PRINT 1 : PRINT 2 : PRINT 3 : PRINT 4 : PRINT 5
```

Place the pointer in the Output Window and run the program. Your screen should look like this:

```
1 2 3 4 5
1           2           3           4           5
1
2
3
4
5
Ok
```

The first set of numbers are right next to one another, separated by a space. The next set is tabbed at intervals across the screen, and the last set is on separate lines. The formatting characters are the semicolon, comma and colon. The semicolon between numbers or text will place printed characters adjacent to one another (we'll explain why the numbers have a space between them shortly), and the comma separates characters into five even placements across the screen. The colon works just like a separate line, so if there's a colon or nothing on the rest of a line after a character in a PRINT statement, the next character will be on the next line. The colon is really not a formatting character; instead it delineates different statements in a single line.

For the most part, we will be using separate lines instead of the colon at first. It is usually clearer to show each statement on individual lines than to combine a lot of statements on a single line. This is especially true if the program starts scrolling horizontally off the screen. The *major* exception to the use of the colon as a line separator is with line labels. The colon must go after a line label and program statements can go after the line label colon. However, you cannot place a line label and then another line label together on the same line. For example, the following would *not* work:

*Wrong*

```
StartHere: PRINT "Whoops" : SecondLine : PRINT "I goofed"
```

If you try to run that, you'll get an 'Undefined subprogram' error. Go ahead and run it to see what happens. After the error, click the 'OK' in the error box and then fix the program with your editor to look like this:

*Right*

```
StartHere: PRINT "Whoops"
SecondLine : PRINT "I goofed"
```

Using your editor, we'll change the values 1, 2, 3, 4 to the words One, Two, Three, Four and run the program again. Remember to put in quotation marks now, since we're dealing with text instead of numbers.

```
StartHere:
PRINT "One"; "Two"; "Three"; "Four"; "Five"
PRINT "One", "Two", "Three", "Four", "Five"
PRINT "One : PRINT "Two : PRINT "Three : PRINT "Four : PRINT "Five
```



---

Run the program and you'll see one interesting difference between what happened with your numbers. The first line looks like this:

```
OneTwoThreeFourFive
```

There are *no* spaces between the words. The numbers all had spaces between them when a semicolon was used. What's going on? Numbers actually have an "invisible" character in front of them denoting their sign. That is, there's really a plus or minus in front of the number but you cannot see the plus sign when it's there. For example, from the Direct Mode type in:

```
PRINT 4 - 10
```

The result, -6, is presented on the screen with the negative sign clearly visible. The same thing happens with positive numbers except you cannot see the sign, and that's what is in those blank spaces between the numbers. If you change the first program so that the numbers are in quotation marks and treated like text, there will be no spaces between them. Try changing:

```
PRINT 1;2;3;4;5
```

to

```
PRINT "1";"2";"3";"4";"5"
```

and run the program again.

## **TABS, REMS, TICKS AND WIDTH**

---

To keep things clear, you will want to use your TAB key to space parts of your program horizontally in your LIST Window. Usually, it's helpful to have each segment label all the way on the left side of the LIST Window. Each line that is part of a given segment should be one or more tab stops to the right. Each tab stop is three spaces. For example, the following program has three parts with three line labels each. Notice how clear the grouping is:

```
PrintNumbers:
  PRINT 123.45
  PRINT 33
  PRINT 77

PrintText:
  PRINT "What goes up";
  PRINT "might just stay";
  PRINT "there."

PrintBoth:
  PRINT "This is #";5
  PRINT 4; " times";2;
  PRINT "equals";2 * 4
```

At this stage of the game, this kind of formatting might not seem too important since we're dealing with small programs. But later on when you have great big programs, you'll be glad that you kept them in separate modules and used tabs to delineate them. Take a good look at the 'PrintBoth' segment of the program to see how we mixed numbers and text. It's easy to print numbers and text together, if you're careful to place the semicolons and the quotation marks in the right places. Notice that we even printed a computer result ( $2 * 4$ ) mixed in with text.

Using descriptive line and segment labels goes a long way to clarify what a program means. However, there will be times when you want a lot of comments about your program which require more than just a line label. Using either the REM statement or tick mark ('), you can write comment sections of your program that let you know what's going on. The program ignores everything in a line after a REM statement or tick mark. For example type in and run this next program:

```
REM This will not be printed to the screen, but it helps to
REM show you what REM does.
```

```
Comment One:
  PRINT "This is printed to the screen."
```

'The tick or single quote mark has the same effect as REM.

```
Comment Two:
  PRINT "Yet all you see is what follows PRINT."
```

As your programs become larger and you write more programs, these comments are crucial. While you are writing a program, you know what you're doing, but when you look at it later on, you may have no idea of why a certain portion is the way it is. Likewise, if you give your program to someone else, the documentation helps them understand what you've done. Further on in this book you'll see how much this internal documentation helps as the programs become longer. It's not much fun to write extra lines that "don't do anything," but in time you'll be glad you did.

The width of your screen defaults to a width somewhere around infinity. (Actually, it's 255, but if text goes beyond 255 it doesn't "wrap around" but just keeps on going. No one knows where, though!) If the material printed on the screen goes beyond the columns visible on your screen, you can't see the whole line. For example, type in the following line in your LIST Window and run it:

```
PRINT "This message is so long that it will scroll right off the  
side of your screen."
```

When you run this program, the words "of your screen" have been shuffled off to the right and you can't see them. To see the whole thing, write in the following line *above* your print statement. (Use your editor; don't rewrite the entire program.)

```
WIDTH 62  
PRINT "This message is so long that it will scroll right off the  
side of your screen."
```

This time when you run the program, you can see the whole message. The WIDTH statement, in effect, adds a "carriage return" after the maximum number of characters has been printed to the screen. Therefore, when you have lines longer than the screen width, just add a WIDTH statement.

Besides using WIDTH to change the width of your screen, you can use this statement to change the tab stops. You saw how the comma between PRINTed text and numbers placed the characters in about five equal groups across the screen. If you wanted more groupings or *print-zones*, then you'd have to change the tab stops. You use a tab value with WIDTH to do this. For example, type in the following program:

```
WIDTH 62,10 : REM Second value is tab value
PRINT "1","2","3","4","5","6","7","8","9","10"
WIDTH ,3: REM Notice the comma and lack of width value.
PRINT "1","2","3","4","5","6","7","8","9","10"
```

When you run the program, you get two different spacings for the numbers. The first set has ten spaces between tabs, and the second has three. We used the quote marks around the numbers so that we wouldn't get the "invisible" spaces caused by the sign value in front of numbers. Experiment with that program trying out different tab widths. Also, notice how the second WIDTH statement only has a single number after a comma. Since we wanted the Output Window width to remain the same (62), we didn't have to put in a different width value. So we just placed a comma after the position where the width value would normally be and inserted our new tab width.

## VARIABLES

---

Using variables is probably going to be one of the most important things you're going to learn about programming in this book. They are flexible and powerful tools, but they are very simple concepts. Essentially, a variable is just a symbol for a value or text. Variables used with numbers are called 'numeric variables' and ones used with text are called 'string variables.' Variables are called such because they change or vary in content. Think of variables as cartons that hold either numbers or text. As you know, the cartons contain what you put in them, and so do variables.

**NUMERIC VARIABLES.** To get started with variables, you'll need to know how to define or assign values to variables. We'll begin with numeric variables. Clear memory with NEW and type in and run the following program:

```
Numbers:
W=5
X=10
Y=33.3
Z=12 * 2
PRINT W,X,Y,Z
```

As you can see, the values assigned to the letters were printed on the screen. To 'assign' a value to a variable, you use the equal sign (=) and the value you want. You can also use computed values, as we did with the 'Z' variable. In fact, computed values are the most important ones you will use in programming.

Besides defining variables with numbers, you can define them with other variables. For example:

```
X=Y+Z
```

is a perfectly correct way to assign a variable. You can use other variables, a combination of variables and numbers, computations or even the variable itself in variable assignment. For example:

```
C=C+1
```

is a common way of assigning a "counter variable" that increments the value of the variable 'C' each time the program passes through that line. (Later you will see how a program can loop through a line several times.)

**VARIABLE NAMES.** Variable names are almost as flexible as variables themselves, but there are some important rules to remember in connection with them. In summary, variable names must have the following characteristics:

1. Each variable must begin with an alphabetic letter (A-Z).
2. Reserved words cannot be used as variable names, but you can use reserved words as parts of variable names.
3. Numbers and decimal points can be used in variable names, but they cannot be the first character.
4. There is a limit of 40 characters in variable names.

The following examples show correct and incorrect examples of variable names:

Name	Comment
NEW=15	<i>Invalid:</i> NEW is reserved word
RENEW=15	<i>Valid:</i> NEW is only part of name
2X=33	<i>Invalid:</i> First character must be letter
X2=33	<i>Valid:</i> Numbers can be part of variable name
THIS ONE=12	<i>Invalid:</i> Spaces are not allowed in variable name
THIS.ONE=12	<i>Valid:</i> Decimal points are allowed

About the only other thing that's involved in naming variables is upper or lower case. As far as your Amiga is concerned, *case doesn't count*, so don't think that using different cases with the same name will result in different variables. For example if you have:

```
WHOOPIE = 40
```

in one place, and

```
Whoopie = 90
```

further on in your program, the values of WHOOPIE will change from 40 to 90. My own personal preference is to keep all variable names in upper case like key words. Then, anything in lower case is assumed to be text messages or comments. When debugging programs, problems are easier to find if you use a consistent format. Here's a little reminder program to show you what happens with changing case in variable names.

```
Apples = 20  
Oranges = 40  
FRUIT = Apples + Oranges  
PRINT FRUIT  
fruit = 99  
PRINT FRUIT
```

As you can see, the two different results of PRINT FRUIT was due to declaring FRUIT as being the sum of the variables 'Apples' and 'Oranges' and then declaring the lower case 'fruit' to equal 99. So the second time the value of FRUIT was printed, it showed 99 instead of 60.

### Amiga Notes

When experimenting with little programs, it's easier to "clear memory" by selecting material in the LIST Window with the mouse and wiping it out with the BACKSPACE than by using NEW. Then you don't have to go through the interruption of deciding whether to save the program and all that other stuff. Your Amiga is just trying to be 'friendly' and prevent you from screwing up by giving you the save option, but sometimes it just gets in the way. With longer programs, you'll appreciate the chance to save a program before creaming it, but with these little example programs, just blow it out with the mouse drag-BACKSPACE technique.

**VARIABLE PRECISION.** The variables we've discussed so far have all been single precision. Precision refers to the number of digits a variable or constant can handle. Single precision variables are accurate up to seven digits. To see how this works, write and run the following little program:

```
Precision.Test:
  A= 1234.5678
  PRINT A
```

As soon as you press RETURN, a pound sign (#) appears after the last digit. This means the number is a double precision variable. When you run the program, the '7' is rounded up to 8 and the result on your Output Window is:

```
1234.568
```

Using your editor, change the program so that the variable 'A' is followed by a pound sign (#) as follows:

```
A#=1234.5678
PRINT A#
```

Now you get all eight digits. Double precision can handle eight digits. Your Amiga uses different types of variables for different tasks. The less the precision in a variable, the less memory and the faster the execution of a program. The default single precision variable is fine for most applications, but in case you need the others, they too are available. Your Amiga helps to remind you of this by adding the correct sign after double precision numbers and long integers. The following summary shows the different types of variables and their characteristics:

Symbol	Type	Features
A%	Short Integer	Integer values (whole) from -32768 to +32767
A&	Long Integer	Integer values between -2147483648 and +2147483648
A or A!	Single Precision	Precision to seven digits with decimals
A#	Double Precision	Precision to 16 digits with decimals

Practice with the following examples. We'll put them all in a single program and watch the results:

```
Show.Variables:
A% = 32767
PRINT "SHORT INTEGER";A%
A& = 2147483647
PRINT "LONG INTEGER";A&
A = 1234.567
PRINT "SINGLE PRECISION";A
A# = 1234.567890123456
PRINT "DOUBLE PRECISION";A#
```

Notice what happens to the number in your LIST Window with long integers and double precision numbers.

**MASS ASSIGNMENTS AND CHANGES.** Sometimes your program needs nothing but double precision, integer, or long integer numbers. Since it's no fun to keep remembering to put %, # or & after variable names, you can assign all your variables to be a certain type all at once. The keywords DEFINT, DEFLNG, DEFDBL and DEFSTR will declare your numeric variables to be of a given precision. The DEF . . . part of the statements refers to DEFine the variable as a certain type, and the second part of the statements refers to short integer (INT), long integer (LNG), double precision (DBL) and string (STR). (The next section explains string variables, so don't worry about that now.) You don't have to declare all variables to be of a certain type; only the ones you want. You simply type in the range of variable names beginning with a given character that you want declared. For example:

```
DEFINT X-Z
```

would declare all variable names beginning with X, Y and Z to be short integer variables. If you have "A-Z," then all variables would be declared to be integers using DEFINT. You can even have split groupings such as:

```
DEFDBL A-C, M-O, X
```

making variables beginning A, B, C, M, N, O and X to be double precision variables. (This may seem strange now, but in big programs, mass declarations can really help organize things and make life simpler.)



Besides making group declarations for variables, you can also change variables. The functions CINT, CLNG, CSNG and CDBL change a numeric variable into short integer, long integer and single and double precision variables respectively. For example, run the following program:

```
Double.2.Int:
  A#= 1234.56789
  A% = CINT (A#)
  PRINT A%
```

Since neither short nor long integers accept fractions (decimals), you only get the first four digits. Furthermore, the fractions are rounded up. In programs where you need to round off fractions, but need the fractions in the computations, using CINT can be very handy.

Another function that turns variables and values into integer values is INT. When you use INT, fractions are *not* rounded up, but instead are simply dropped. For example, change the last example program by removing the 'C' in 'CINT' so that it looks like the following:

```
Double.2.Int:
  A#= 1234.56789
  A% = INT (A#)
  PRINT A%
```

Now instead of '1235,' your Output Window shows '1234.' When you need to drop the fraction with no rounding up, use INT; when rounding up is required, use CINT.

**STRING VARIABLES.** Variables that can hold text are called "string variables," and they are denoted by a dollar sign (\$). For example, try the following:

```
ShowString:
Computer$="Amiga"
PRINT Computer$
```

The same naming rules apply to string variables as they do to numeric variables, and you can use all kinds of different symbols, including numbers, in strings. For example:

```
WOW$=" @#$$@#$$%$%^&^*(1234245ASFXCVasfreqr"
```

is a perfectly legitimate string declaration. However, when a number is part of a string, it is not treated as a real number, it's considered text. For example, run the following little program:

```
Number.String:
  TWO$ = "2"
  FIVE$ = "5"
  PRINT TWO$ * FIVE$
```

That'll give you a Type mismatch error since you need numbers for multiplication. Using your editor, change the program so that the asterisk is a plus sign as in the following:

```
Number.String:
  TWO$ = "2"
  FIVE$ = "5"
  PRINT TWO$ + FIVE$
```

Now run it, and see what happens. You get '25' printed to your screen. You have just discovered "concatenation." That means you've tied two strings together using the plus sign. However, you really can't see how concatenation works very well with numbers, so let's look at some examples with text:

```
String.Ties:
  CITY$ = "San Diego," : REM SPACE AFTER COMMA
  STATE$ = "California"
  PRINT CITY$ + STATE$

  LASTNAME$ = "Cadet"
  FIRSTNAME$ = "Space"
  PRINT FIRSTNAME$ + " " + LASTNAME$ : REM SPACE BETWEEN QUOTES

  FIRSTPART$ = "Ahhhh"
  SECONDPART$ = "Chooooo!"
  SNEEZE$ = FIRSTPART$ + SECONDPART$ : REM CREATE THIRD STRING
  PRINT SNEEZE$
```

Concatenation can be very useful when different parts of strings have to be stored separately but brought together in different parts of the

program. You'll find this especially true when writing programs that need separate categories for first and last names. Also, note that new string variables can be created by concatenating two other string variables as we did with SNEEZE\$.

The other basic formatting characteristics that apply to numeric variables also apply to string variables. Semicolons place strings adjacent to each other and commas separate them by tabs. For instance, this next program shows string variable formats:

```
String.Formats:
  DAY$= "Thursday"
  MONTH$="JANUARY" : REM SPACE AFTER 'Y'
  YEAR$="2010"
  MONTHDAY$="25"
  PRINT DAY$,MONTH$;MONTHDAY$,YEAR$
```

Finally, like numeric variables, you can make group declarations. Using DEFSTR. For example, DEFSTR A-Z would make all variables string variables even without the dollar sign. This next little program shows how to use DEFSTR:

```
Gang.Strings:
  DEFSTR J-Z
  A= 99
  J="We want"
  Z="drums in the band."
  PRINT J;A;Z
```

**STRING-NUMBER CONVERSIONS.** Two of the more useful functions you will find in Amiga Microsoft BASIC are STR\$ and VAL. These are used to change string variables into numbers and numeric variables into strings. First, let's look at VAL. It changes string numbers into real numbers:

```
String.2.Num:
  SIX$ = "6"
  THIRTY$= "30"
  SIX=VAL(SIX$)
  THIRTY=VAL(THIRTY$)
  PRINT SIX * THIRTY
```

You may be wondering why on earth anyone would want to go through all the trouble of first defining a string with a number and then converting it to a numeric variable for math functions. Why not just define numeric variables in the first place? There are certain kinds of applications where a number may be part of a string for one situation and then needed as a number for other things. Further on in this book you'll learn how to use parts of strings. Sometimes numbers make up only a part of what needs to be manipulated mathematically. In those cases, you would need first to get the part of the string with the number and then convert it to a real number.

On the other hand, you can convert numbers to strings. Using `STR$`, numeric variables are transformed into string variables. For example, the next program takes a numeric variable and transforms it into a string variable, and then concatenates it with another string in a third string variable.

```
Num.2.String:
  A=10
  A$=STR$(A)
  HOME$=A$ + " Downing street"
  PRINT HOME$
```

Like `VAL`, you may not see right away why `STR$` is useful, but as you progress, you will find it to be a very useful function.

## SUMMARY

---

This chapter took you through the fundamentals of variables and some elementary formatting. The several kinds of variables handle different types of data with varying degrees of precision. The essence of a variable is its ability to *change* in a program. Not only can a variable change in content, as from one string or number to another, but variables can also change in type. We found there were five different types of variables; four numeric and one string. Numeric variables can keep the same contents but be changed in type, as from integer to single precision floating point. Furthermore, strings can be transformed into numeric variables and vice versa.

The flexibility of variables give them their power in programming. We're used to calculating numbers, and numeric variables probably

make more sense to us than string variables. However, just as the content of our talk changes depending on the topic or conversation, string variables can also change depending on what the program does. As you progress, calculations with string variables will become as natural as calculating numbers.

Finally, you should begin to understand the importance of formatting. No matter how good a job you do in writing a program, it's important for your program to show you what the results mean. If all you get is the result of the calculation with no indication of what it is the result of, or if everything is all scrunched together, then you're only doing half of the programming job. As you add more to your programs, formatting will increase in importance, and as you continue in this book you'll learn more words to format with.



# Math Operations

As we saw in Chapter 1, basic mathematics in BASIC is a simple matter. Since your computer can take complex values and quickly return computation results your Amiga is uniquely qualified to perform a wide variety of mathematical tasks. Using the PRINT statement, the outcome of any calculation is sent to the screen. For example:

```
PRINT 8.87 * 390.85
```

returns 3466.84 on your screen. But since that is nothing your calculator can't do, let's see some other more valuable work Amiga programs can do.

## SEQUENTIAL CALCULATIONS

---

The first thing to learn is how to sequentially build a program that will perform several calculations at once. More sophisticated calculators do the same thing, but such calculators are actually performing computer functions in that they are using memory storage. What we will be examining here are very simple methods for computing results; more advanced users may wish to skip this section and go directly to the section on precedence. For those of you new to programming, though, this section will be important for understanding how to organize more complex mathematical operations.

To get started, let's take a look at finding a mean or average. For example, suppose you need to find the average number of disks you used in your business per month during the last fiscal year. Supposing you have records of the monthly volume of disks, you can simply add them all up and divide the total by 12. To make our program clear, we'll use month abbreviations as variable names:

```
Disk.Average:
  JAN=10
  FEB=13
  MAR=15
  APR=20
  MAY=9
  JUN=21
  JUL=6
  AUG=12
  SEP=22
  OCT=33
  NOV=44
  DEC=21
First.Half:
  TOTAL=JAN + FEB + MAR + APR + MAY + JUN
Second.Half:
  TOTAL=TOTAL + JUL + AUG + SEP + OCT + NOV + DEC
Calc.Mean:
  MEAN = TOTAL/12
  PRINT "The average number of disks used was";MEAN
```

The program was relatively long for the simple task it performed, and we could have started off by defining TOTAL as the sum of each monthly value. For example:

```
TOTAL = 10+15+13+20+9+21+6+12+22+33+44+21
```

would have done the same thing with less work. However, there were some other things to learn; so we took a more circuitous route. First, by using descriptive variables, you have a much better idea of what's happening. Thus, the monthly names show you in the listing exactly how many disks were used each month. Second, you can see in the segments labelled 'First.Half' and 'Second.Half' how to combine a value with itself. The variable TOTAL is the sum of the first six months at the end of



'First.Half.' To combine TOTAL with the second six months, you need to start by adding the first six, which in this case, are already accumulated in TOTAL. Then, you add TOTAL to the second six months. If you just defined TOTAL as the second six months, you would simply replace the first six months with the second six months and not combine them.

When you first start programming, it is best to avoid shortcuts and put all of the sequences in clearly labelled paths. This will help you understand the structure of programming and lay out all of the steps of a problem. By breaking down a more complex problem into very simple steps, you can solve just about any programming problem. This takes more time, but as you become more proficient, you will see the value of having taken the extra steps. While more advanced techniques may not take the same number of programming steps, the *logic* of the steps is the same.

## PRECEDENCE

---

As we begin dealing with more and more complex math, we will need to observe a certain order in which problems are executed. This is called "precedence." Depending on the operations we use, and the results we are attempting to obtain, we will use one order or another. For example, let's suppose we want to multiply the sum of two numbers by a third number—say the sum of 7 and 8 multiplied by 3. If you entered

```
PRINT 7 + 8 * 3
```

you would get 3 multiplied by 8 to which 7 is added on. You got 31, and you wanted 45. Here's what actually happened in the form of a program:

```
Addemup:
```

```
A= 3 * 8  
B= A + 7  
PRINT B
```

And you wanted:

```
Add.first:
```

```
A=7 + 8  
B=A * 3  
PRINT B
```

The reason for the sequence of execution in the order it took was precedence—multiplication precedes addition. In general, math is computed from left to right, but the rules of precedence reorder the flow of execution. The following list shows the order:

1. ^ Exponentiation
2. - Minus sign; not subtraction
3. \* Multiplication
4. / Floating point division
5. \ Integer division
6. MOD Modulo calculation
7. + - Addition and subtraction

Try some different problems and see if you can get what you want.

## Reordering Precedence

Once you get the knack of the order in which math operations work, there is a way to simplify their organization. By placing two or more numbers in *parentheses*, it is possible to raise their priority. Let's go back to our example of adding 7 and 8 and then multiplying by 3, but this time we will use parentheses:

```
PRINT (7 + 8) * 3
```

Since the multiplication sign has precedence over the addition sign, without the parentheses we would have gotten 3 times 8 plus 7. However, since all operations inside parentheses are executed first, your computer *first* added 7 and 8 and then multiplied the sum by 3. If more than a single set of parentheses is used in an equation, then the computer executes the innermost first and works its way out.

## Amiga Notes

### Precedence Maze

To help you remember the order in which math operations are executed within parentheses, think of the operations as being caught in the center of a maze. Each maze segment is composed of parentheses, and you must exit each parenthesis from the center outwards. Once outside of the maze, everything works from left to right.

The following examples show you some operations with parentheses:

```
PRINT 20 + 10 * ((5 - 4)+7)
PRINT (12.43 + 92) / 3 ^ (11 - 3)
PRINT (62 - 3.1415) * (91 + 3.1415)
PRINT (((86 + 9) - (4.6 + 5)) / .31) * 2.55
PRINT 6 / 2 * ((51 / 3) - (100 / 14) + 6)
```

It's important to remember to keep the number of left and right parentheses the same. If you run into an error in your program using re-ordered precedence, the first thing you should do is to count your parentheses. Now try some of the following problems using the format expected by your computer:

1. Using a *single* line, rewrite the program that calculated the average number of disks used.
2. Multiply 15 by 15 and the result by 3.1415. This will compute the area of a circle with a radius of 15. To find the area of any other circle, change 15 to another value. (This would be a good little routine to save.)
3. Add up the monthly charges on your utility bills and divide the sum by 12. If you sell your house, you can tell prospective buyers the average amount of your bills.

## MATH FUNCTIONS

---

Math functions are built-in formulas for calculating commonly used math operations. For example, the function COS returns the cosine of a value. Similarly, SQR returns the square root. In addition to the built-in functions, you can also define your own functions for specialized or general needs you may encounter. In this section, we'll look at the built-in functions and how to define your own.

### Built-in Math Functions

Most of the functions in Amiga's BASIC are for mathematics. In this section, we will only be concerned with those used in math (with the exception of those that define variable types, e.g., DEFINT). These func-

tions can be used for calculating straight math problems or for calculating graphics and similar applications.

**SIN.** Returns the sine of a value. You may want double precision with some of these functions, depending on your application. Note the difference in the outputs using single and double precision in the following example:

```
PRINT SIN(4)
B# = SIN(82)
PRINT B# : REM Double precision output
```

Output:

```
-.7568024
.3132287859916687
```

**COS.** Returns the cosine of a value.

```
PRINT COS(4)
C# = COS(82)
PRINT C#
```

Output:

```
-.6536436
.9496777057647705
```

**ABS.** Returns the positive value for any number, including negative numbers.

```
PRINT ABS(-32)
```

Output:

```
32
```

**ATN.** Returns the arc tangent of a value. In calculating circle segments for such applications as pie charts, this function is useful.

```
PRINT ATN (270)
```

Output:

```
1.567093
```

**TAN.** Returns number's tangent value.

```
PRINT TAN(90)
```

Output:  
-1.9952

**EXP.** Returns the base of natural logarithms (e) to e's power. The value of e is 2.718281828490.

```
PRINT EXP(7)
```

Output:  
1096.633

**FIX.** Drops, without rounding, fractions from number. Integer variables will round numbers up, and INT rounds them down. FIX just cuts off the fractions with no rounding whatsoever.

```
PRINT FIX(5.777)
A=5.777
A%=A
PRINT A%
```

Output:  
5  
6

**INT.** Returns the integer of a value rounded down.

```
PRINT INT(9.875)
A%=9.876
PRINT A%
```

Output:  
9  
10

**LOG.** Returns the logarithm of numbers greater than zero.

```
PRINT LOG(8)
```

Output:  
2.079442

**SQR.** Returns the square root of a number.

```
PRINT SQR(16)
```

Output:  
4

The above functions are what you are probably most familiar with if you use math a lot in your work. There are several non-math functions that do everything from returning a value indicating the position of your cursor on the screen, to finding the last file on your disk. We'll get to those later. But now, let's take a look at three more numeric functions closely tied to your computer.

## Random Numbers

Your computer has a pseudo-random number generator. It is not a true random number generator in that some base other than total chance generates the numbers, but it is about as close as you can come. There is a statement, `RANDOMIZE` and a function, `RND`, that when used together can do a very good job of generating random numbers. The statement `RANDOMIZE` seeds the random number generator. You can either put a value after `RANDOMIZE`, enter a value from the keyboard or use `TIMER` that generates the numbers of seconds after midnight from your Amiga's internal clock. To get started, enter the following program that uses a fixed random seed:

```
RANDOMIZE 55  
A=RND(1)  
B=RND(2)  
C=RND(3)  
D=RND(4)  
PRINT A,B,C,D
```

Run the program a few times, and you will see that the four values are always the same. Now change the first line to read:

```
RANDOMIZE TIMER  
Etc . . . .
```

Again, run the program several times. Each time you run it, the values change since the random number seed changes. Finally, delete TIMER so that the first line simply reads

```
RANDOMIZE  
Etc . . . .
```

This time when you run the program, you are asked to provide a number. Go ahead and do it to see what happens. Depending on what you need in a program, you can seed your random number generator in any of these three ways.

You probably noticed that all of your numbers were fractions. What if you wanted to write a program that needed whole numbers? Suppose you wanted to make a simple math skill building program for your child using integers and you wanted to randomly generate different values to be added. All you have to do is to multiply the random values by 100, and then use the INT function to change the number into an integer. For example, run the following program several times to see the different whole numbers it can generate:

```
RANDOMIZE TIMER  
A=RND (1)  
A= A * 100  
A=INT(A)  
PRINT A
```

A more compact way of doing that would be:

```
RANDOMIZE TIMER  
A=INT(RND(1) * 100)  
PRINT A
```

Compare the two programs to make sure you understand how the sequence of the first program was maintained in the second. Remember the rules of precedence and how they were ordered in the second example.

To generate a range of numbers, you would use the following formula:

```
A= INT(RND * (H + 1-L)) + L
```

The variable H is the high limit and L is the low limit. For example, if you wanted to generate numbers between 5 and 25, you would use the following:

```
A = INT(RND * (25+1-5)) + 5
```

Pick any two whole numbers you want, and see if you can write a program that randomly generates values between the low and high values you chose.

## Hexadecimal and Octal Numbers

This section describes numbers that have a base different than decimal. This will be only an illustrative introduction; in later chapters we will describe different number bases and how they relate to your computer.

Hexadecimal numbers are base 16; octal numbers are base 8. To convert decimal values into hexadecimal or octal, use the HEX\$ and OCT\$ functions, respectively. For example, the decimal value 10 in hexadecimal is 'A' and in octal, it is 12. The following program shows how to make the conversion:

```
Hex.Oct:
  PRINT HEX$(10)
  PRINT OCT$(10)
```

When you run that program, you'll get an 'A' and '12.' Both represent integer numbers in another counting system. Also, both octal and hexadecimal values are stored as strings. Therefore you must use string variables for storage of octal and hex values.

## Defined Functions

The final functions we will cover in this chapter are those that you create yourself. To define your own functions, use DEF FN. For example, if you want a function that will find the square of a number (to compliment your built-in SQR function) you can define it with DEF FN. Here's how:



```
Def.Square:
  DEF FNSQ(X) = X^2
  PRINT FNSQ(4)
  PRINT FNSQ(22)
```

The function FNSQ acts just like any of the other functions we've discussed except instead of being built into your BASIC, it's one you've created yourself. In the appendix of your *Amiga BASIC* manual that comes with your computer, there's a whole list of functions you may want to define.

## SUMMARY

---

One of the easiest things for your computer to do is mathematical calculations. The most important thing to remember about your calculations is the order in which you put everything. Initially, it is best to lay all of your calculations out in separate operations on separate lines. As you become more adept at programming, you will be able to combine several steps into a few. However, remember that even the most complex problem can be broken down into several small steps.

Precedence is an important element to keep in mind when creating equations. The order of precedence can be changed, and by doing so you can write more compact programs. When you develop more sophisticated programs, precedence will become an increasingly important aspect to watch.

Finally, you learned about the major functions available in BASIC. These are actually little formulas that have been written for you, and as you saw with defined functions, you can even make your own functions. As you learn more and more about programming, you will learn a lot of tricks where these arithmetic computations do much more than just print numbers on the screen.



# **Sequential Modular Program Organization**

This chapter will introduce you to program structure. As you begin to program more, there will come a point where you will want to test your creativity. In many ways, creative programming is just like creative writing: you will want to pour out your program ideas just as a novelist wants to create characters and a story. However, good novelists work within a structure to provide a guide and sense to their story. The reader can thus follow the development and benefit from the author's creative efforts.

Since your Amiga provides a creative environment, there's a tendency for new programmers to confuse creativity with getting a little routine to run. For example, by the time you are finished with this chapter, you will be able to write an interactive program. You can be as inventive and experimental as you wish. (In fact, you should experiment and try things on your own! If you're not doing that already, begin doing so right away.) As you progress in your learning and understanding, you will want to tackle larger projects that require several different techniques and routines you've developed. If you have no organized procedure or structure for arranging all of the creative routines you've written, you can become snarled in an impossible tangle of program codes which will

eventually get in the way of creativity. To avoid that problem, we will begin here with procedures to help you keep everything running smoothly and your creativity flowing.

## TOP DOWN PROGRAMMING

---

What follows will seem so simple that you may wonder why we bother pointing it out. As you do more programming, the importance of this chapter will become more obvious, but right now, it may appear that we are belaboring the obvious. First, let's address the question of what is 'Top Down Programming.' With the few programming statements now at your command, there may appear to be a single way to write a program—a sequence with each part in place beginning at the top and going to the bottom. That's essentially what we mean by 'Top Down,' proceeding from the beginning in an orderly fashion from the beginning to the end. For example, this next program is a Top Down one:

```
Show.Functions:
Def.Vars:
  SAMPLE=25
  SAMPLE$="Sample="
  SROOT$="Squareroot="
  TANGENT$="Tangent="
  COSINE$="Cosine="

Screen.Output:
  PRINT SAMPLE$;SAMPLE
  PRINT SROOT$; SQR(SAMPLE)
  PRINT TANGENT$; TAN(SAMPLE)
  PRINT COSINE$; COS(SAMPLE)
```

The 'top' of the program was the first labelled line. That gives an idea of what the program does. Next, all of the variables are defined with names which tell something about what they do. Finally, we send the information to the screen. Even before you run the program, you have a pretty good idea of what to expect.

Now compare it to the following:

```
S=25 : S$="Sample=" : PRINT S$;S  
R$="Squareeroot=" : PRINT R$; SQR(S)  
T$="Tangent=" : PRINT T$; TAN(S)  
C$="Cosine=" : PRINT C$;COS(S)
```

Both programs do the same thing, but the second is not as clear or as well organized as the first. It may seem better than the first one since it takes less space to write and the variable names are not as long. But suppose the program was over a hundred lines long and there was a problem or 'bug' in it. You would not be sure where the problem lies because everything is mixed together. That is, you could not easily isolate the problem. Instead of being able to go to the block that defines variables or the block that prints things to the screen, you'd have to wade through the combined definitions and output. A less simple program would be a real mess to fix.

Your Amiga provides a way of helping you. When most beginners get going, they like to see how compact they can make their programs. Instead of having several lines, they like to cram everything into a single line. On some computers, the program listing will wrap around. This means that if your program line scrolls off the screen to the right, it 'wraps around' and puts the line on the left. In that way, very long program lines can be viewed without scrolling. However, on your Amiga, as soon as your program line goes off the right of your LIST Window, the window scrolls. You can make very long program lines, but if you do, it's necessary to scroll horizontally to see the whole line. In debugging a program, this is a real problem. Therefore, expand your LIST Window to fill the screen, and if your line starts scrolling horizontally, it's a subtle hint from your Amiga that you'd be better off writing more lines instead of trying to put everything on a single line.

## GETTING THINGS IN ORDER

---

A key element of Top Down programming is planning. Instead of sitting down at the keyboard and keying in whatever comes to mind, it's a better idea to plan ahead. You don't have to make an elaborate flow chart; rather, you should just jot down what you want your program to do and the order in which it is to perform various tasks.

Let's take a practical task for your computer and see how to write a Top Down program. First, we'll list the things necessary to organize a Top Down program:

1. State the goal of the program
2. Arrange the logical order of procedures to achieve the goal
3. Write the program

**THE GOAL.** Let's write a program that adds up the monthly expenses you have for major items in your family. We'll include expenses for the following:

1. Housing
2. Transportation
3. Food
4. Clothing
5. Utilities

**LOGICAL ORDER.** Now let's see in what order we would have to arrange a program so that it would achieve the goal:

1. Enter the data.
2. Find the sum of the data.
3. Label the results.
4. Send the results to the screen.

The arrangement here may appear obvious, and largely, it is, but later on this step will be the most important thing to do correctly. At this stage, we have very few programming statements and a relatively simple goal. However, as you progress, you may have to juggle more information and programming considerations. At that time you will be glad you spent a little time laying out a program's order before you actually start writing the program.

**WRITE THE PROGRAM.** Finally we come to the program. We can now just follow the logical steps and crank out the program:

```
Fam.Expense:
Ent.Data:
HOUSE=755
```

```
TRANS=233
FOOD=532
CLOTH=124
UTIL=154

Find.Sum:
SUM = HOUSE + TRANS + FOOD + CLOTH + UTIL

Make.Label:
SUM$="Total monthly expenses="
To.Screen:
PRINT SUM$;SUM
```

After having taken a few organizational steps, the program almost writes itself. Save this program on your disk; you'll need it later.

## INTERACTIVE DATA ENTRY

---

Now that you can see how to enter data into your program through declarations of variables, let's see how it can be done dynamically and interactively while your program is running. This will make your programs far more flexible since you are not stuck with the data you entered when equating variables. (Of course, you *could* rewrite the part of the program where the variables are defined.)

**INPUT.** The first method we'll examine for getting data into your program while it's running is INPUT. This statement stops the program and waits until the RETURN key is pressed. Whatever you write before you press RETURN is entered into one or more variables. There are two important rules to remember:

1. Do not key in non-numbers using numeric variables.
2. Do not enter commas.

Now let's write a simple program that shows what INPUT can do:

```
INPUT A
PRINT A
```

Run that little program. You will be presented with a question mark on your screen. This is the 'prompt' for you to enter a value. Since 'A' is a numeric variable, you must type in a number. Go ahead and pick any number you want and press RETURN. When you do so, the number you typed in is printed on the screen. Now, run the program again, but this time instead of typing in a number, type in a letter to see what happens. You'll get something like the following on your screen:

```
? W
?Redo from start
```

The "?Redo from start" message means you put in the wrong thing and it gives you another chance to do it correctly. Just enter any number and your computer will be as happy as a clam.

Change your program so that A is changed to A\$.

```
INPUT A$
PRINT A$
```

Now when you run it, you can type in anything except commas. If you enter numbers, though they are treated like text and cannot be used for mathematical calculations. Run it a few times to try it out.

As you can well imagine, if all you saw on your screen was a question mark, you may not know what to do next in a program. For example, if your program wanted you to first enter a string and then a number, you could not tell from just a question mark what you should do. There are two ways to deal with this problem. First, you can simply use your PRINT statement to write a message preceding the INPUT line. For example, this program clearly shows you what the program expects:

```
Cost.Getter:
  PRINT "Enter item";
  INPUT ITEM$
  PRINT "Enter cost";
  INPUT PRICE
  PRINT ITEM$, PRICE
```

When you run that program, you can tell exactly what type of information your computer wants. Furthermore, by using descriptive variable names in the program, you can tell from the listing what each variable does.



---

Another way to do the same thing with INPUT is to place the prompt message directly in the INPUT line. You can see how that's done in another version of the last program:

```
Cost.Getter:
    INPUT "Enter item";ITEM$
    INPUT "Enter cost";PRICE
    PRINT ITEM$, PRICE
```

That saves time, but it does not make the program any less clear. In fact, the program is probably clearer than the first method since you can tell at a glance what needs to be entered and the name of the variable that will store the information.

Using these new statements, let's rewrite the household expense program so that you can easily enter any amounts you want without having to change the program. (HINT: Don't rewrite the whole program; load the old one from your disk and just make changes with your editor.)

```
Fam.Expense:
    Ent.Data:
        INPUT "Mortgage/Rent"; HOUSE
        INPUT "Transportation"; TRANS
        INPUT "Food costs"; FOOD
        INPUT "Clothing";CLOTH
        INPUT "Utilities";UTIL

    Find.Sum:
        SUM = HOUSE + TRANS + FOOD + CLOTH + UTIL

    Make.Label:
        SUM$="Total monthly expenses="

    To.Screen:
        PRINT SUM$;SUM
```

That was easy; you just had to make a few changes from the old program with your editor. Notice that the 'Find.Sum:' routine worked the same with variables whose values you entered from the keyboard as with those you placed in the program with equates. This version, though, is a lot easier for entering different data than the first one.

### Amiga Notes

#### Saving New Versions and Keeping Old Ones

Since Murphy's Law ("If something can go wrong, it will") seems to work overtime with computer programs, it's a very good idea to make progressive copies of your programs. Then if you goof up part of the program and it chokes your Amiga to reset, you won't lose everything. To save both the old version and new version of your program, all you have to do is to choose the 'Save As' option from your project menu. Then save the new version under a different name. The old version stays intact on your disk and you have the new version as well. Once you have the final version completed, you can get rid of the old ones taking up disk space.

Another way you can put INPUT to good use is by letting it insert a pause in your program. If you want different messages sent to the screen to be read separately, you can use INPUT to stop the program until the user presses the RETURN key. The following program gives the user two messages presented separately:

First.Info:

```
PRINT "This is to inform you that your lottery ticket"  
PRINT : PRINT : PRINT  
INPUT "Press RETURN to continue";RT$
```

Second.Info:

```
CLS  
PRINT "was eaten by the lottery computer."  
PRINT  
PRINT "(The computer was not an Amiga.)"
```

In this case, INPUT and the variable RT\$ were just used to halt the program; no information was stored in RT\$. When you have long messages that fill up the screen or you want different segments of messages presented separately, such as in guessing games, this application of INPUT can be very useful.

### Amiga Notes

#### 'Illegal' Input

One trick to use to input anything you want in a string variable is to place a quote mark as the first thing you type in when prompted for input. You can insert commas with no problem and they are stored in the string variable. Go ahead and try it out. Just remember to put the quote in first.

**INPUT\$.** Besides using INPUT to stop your program and get information, you can use the keyword INPUT\$. This word is used more extensively with files (as discussed further on in the book), but it can also be used with information from the keyboard. The only major problem with INPUT\$ is that you cannot see the keys you press on the screen. (In computer talk, this means the keypresses are not 'echoed' to the screen.) The format for INPUT\$ is as follows:

```
V$=INPUT$(N)
```

The variable (V\$) must be a string, and the program will wait until N characters are received. For example the following program expects three keypresses:

```
PRINT "Press any three keys"  
THREE$=INPUT$(3)  
PRINT THREE$
```

The advantage of INPUT\$ over INPUT lies in the control of the number of characters to be entered. But, it really is not a good substitute for most INPUT applications since you cannot see what you write until a PRINT statement sends the information to the screen. Also, INPUT\$ takes off as soon as the indicated number of keys are pressed, while using INPUT, if you make a mistake, you can change it before pressing RETURN. INPUT\$ is better used as a temporary program pause where any key can be pressed to continue the program flow. This next program illustrates INPUT\$ as a "hit any key" pause:

```
Hit.Me:
  PRINT "What's a female friend in Spanish?"
  PRINT : PRINT : PRINT : PRINT : PRINT
  PRINT "(Hit any key for answer.)"
  HIT$=INPUT$(1)
```

```
Answer:
  CLS
  PRINT "An Amiga!"
```

Whatever key is pressed is stored in the INPUT\$ variable. In the above program, it is HIT\$. To better illustrate how INPUT\$ stores information in variables, we'll use a simple program that lets you press three keys and then prints them on the screen:

```
Three.Key:
  PRINT "Enter any three letter word"
  THREE$=INPUT$(3)
  PRINT "Your word is =>"; THREE$
```

The program creates a strange sensation by having you "type in the dark" and then pop your word to the screen. However, it does show how INPUT\$ is used to store information in variables. Using several INPUT\$ statements in a program you can create your own echo. (Hint: When you write this program use the copy and paste functions of your editor to create those parts of the program that are redundant.)

```
Echo.Back:
  PRINT "Enter any three letter word"
First.Let:
  THREE$=INPUT$(1)
  PRINT THREE$;
  WORD$=WORD$ + THREE$
Second.Let:
  THREE$=INPUT$(1)
  PRINT THREE$;
  WORD$=WORD$ + THREE$
Third.Let:
  THREE$=INPUT$(1)
  PRINT THREE$;
  WORD$=WORD$ + THREE$
```

```
Whole.Thing:
  PRINT : PRINT : PRINT : PRINT
  PRINT "Your word is =>"; WORD$
```

Each time **THREE\$** is printed, it effectively echos back what you just typed. Thus, while the program seems relatively long, it executes commands so fast that you cannot see all of the work being performed to get your typing “out of the dark.”

**INKEY\$.** We’re getting ahead of ourselves on this keyword, but it is important to include it in this section about entering data interactively. **INKEY\$** works a lot like **INPUT\$(1)** except that it requires a routine with a loop and branch. In the next two chapters we’ll discuss loops and branches. Right now just use the format for **INKEY\$** and wait until later to see what it all means. The following example shows a possible application:

```
Yes.No:
  PRINT "Answer [Y]es or [N]o"
Get.Key:
  ANSWER$=INKEY$: IF ANSWER$="" THEN Get.Key
  PRINT ANSWER$
```

As you can see, it has the same effect as **INPUT\$(1)**, but **INPUT\$** is a lot easier to use.

**UCASE\$.** The final keyword we’ll introduce in this section changes keyboard letters to uppercase. Later on when we get into branching, we’ll be dealing with evaluations of what key was last pressed. Since your Amiga differentiates between upper and lower case letters when evaluating strings, one way to save time and avoid confusion is to have all entries of a certain type changed to upper case. (With numeric variables case is ignored.) In that way, if you enter an upper or lower case letter, it will be treated the same. For example, the query:

```
Do you want to continue?(Y/N)
```

is a common one. If your **CAPS LOCK** key is on, then you will automatically enter an upper case Y or N, but if it’s not, you’ll have to remember to use a shifted Y or N. This may seem like a petty point, but when your program bombs because you forgot to enter an upper case letter, you’ll be

glad you met UCASE\$. The following program illustrates its use. When you run it, make sure your CAPS LOCK key is off (the red light should be out).

```
Auto.Cap:
  PRINT "Do you want to continue?(Y/N)"
  REPLY$=INPUT$(1)
Up.it:
  REPLY$=UCASE$(REPLY$)
  PRINT REPLY$
```

Whether you run it with the CAPS LOCK on or off, it returns a capital letter. This handy little function will crop up again and again, so keep it in mind.

## GETTING INFORMATION TO THE SCREEN CLEARLY

---

We saw that it is important to clearly label prompts so that the user will know what to do next with input. The same is true for output. By clearly labelling what you send to the screen with PRINT you know what is going on. All along in this chapter, we've been using *clear* labels so that you know the result of a calculation. A lot of times when we write programs for ourselves, 'we know' what the output is for, so we don't bother with clear labels. For example, if you write a simple calculator program that does nothing but addition, you know the results are going to be the sum of what you last entered. However, if you don't use the program for a while and run it at a later date, you may have no idea of what the output means. And when you become more skilled, you'll have programs that can do a lot of different things. Compare the following two simple calculator programs to see the important difference output labels make:

```
Poor.Output.Labels:
  INPUT "Enter a number: ";N
  PRINT TOTAL= TOTAL+N
  PRINT TOTAL
  INPUT "Enter a number: ";N
  PRINT TOTAL= TOTAL+N
```

```
PRINT TOTAL
INPUT "Enter a number: ";N
PRINT TOTAL= TOTAL+N
PRINT TOTAL
```

By making some very small changes in the program, you can see exactly what your output is. Change it to look like the following:

```
Clear.Output.Labels:
INPUT "Enter a number: ";N
PRINT TOTAL= TOTAL+N
PRINT "Running total is =>";TOTAL
INPUT "Enter a number: ";N
PRINT TOTAL= TOTAL+N
PRINT"Running total is =>"; TOTAL
INPUT "Enter a number: "; N
PRINT TOTAL= TOTAL+N
PRINT "Final total is =>";TOTAL
```

The change is so little that it takes almost no time at all to include in your program, but it makes a big difference in what your program does.

**WRITE.** Besides using PRINT for screen output, you can use another statement called WRITE. The useful thing about WRITE is that it will send anything to the screen. For example, enter the following from the immediate mode:

```
WRITE "The absent are always in the wrong."
```

As you will see, the whole phrase, quotes and all, is sent to the screen. If you need quotes in your output, you can use WRITE, but there are some drawbacks using this word. Try the next line in the immediate mode:

```
WRITE 2,4,6,8
```

Using PRINT, those numbers would be spaced in columns across the screen, but with WRITE, you have no formatting with commas, and the commas are simply plopped on your screen. In some applications, that may be just what you need, but it is not a good idea to substitute WRITE

for PRINT except in situations where you need the special characteristics of WRITE. This next one will show you something strange:

```
WRITE WEIRD$
```

(Put that in a program to confuse someone!)

## READ THAT DATA

---

Up to this point, we have four ways of putting information into variables:

1. Equate (e.g., AMIGA\$="Computer")
2. INPUT
3. INPUT\$
4. INKEY\$

Now we're going to learn a fifth way that uses two words in combination with one another, READ and DATA. This method makes it relatively easy to store information within a program for later use. When you become familiar with disk files, you will probably want to use them instead, but for a simple data storage technique, READ/DATA is handy. The information for the READ variable is in a program line beginning with DATA. Try this little program:

```
Watch.This:
```

```
READ A  
READ A$  
READ B  
PRINT A,A$,B
```

```
Info.Here:
```

```
DATA 23,Skidoo,88
```

The first READ statement loads the variable 'A' with the first *element* in the DATA line. In our example, it's 23. Then the invisible pointer moves to the next element, 'Skidoo.' Finally, the third READ statement looks at the third element in the DATA line, 88. You have to keep your READ variables straight with the kinds of data they will read. For



example, if you had a numeric variable for the second element in the DATA line, "Skidoo," you'd get an error message. Change the variable A\$ to X and see what happens. You'll get a Type mismatch error. That's just like trying the write:

```
X="Skidoo"
```

and it doesn't work any better.

Change the X back to A\$ and insert the following line in the program right after READ B:

```
READ X$ : PRINT X$
```

When you run the program, you'll get an 'Out of DATA' error. That means there were not enough DATA elements for the READ statements. There must be *at least* the same number of DATA elements as there are READ statements or a RESTORE statement. The RESTORE statement simply places the pointer back at the beginning of the DATA elements. Fix up your program so that it looks like this:

Watch.This:

```
READ A
READ A$
READ B
RESTORE
READ X$
PRINT A,A$,B
PRINT X$
```

Info.Here:

```
Data 23,Skidoo,88
```

Your screen will now show:

```
23          Skidoo          88
23
```

Here's what happened. The 23 was read into the variable 'A,' the text "Skidoo" into the variable 'A\$' and the number 88 into the variable 'B.' The pointer was restored to the beginning of the DATA elements with

RESTORE so that 23 became the next value to be read. The 23 was loaded into a *string* variable this time. Notice in the output that the first 23 was one space out from the screen's side and the second one was right next to the side. Do you know why? Remember, the first time the 23 was defined in a numeric variable, so the invisible sign is taking up one space there. When the 23 was read into 'X\$', it treated the number as though it were text; therefore there's no invisible sign taking up a space.

## **BREAKING DOWN LARGE PROBLEMS INTO SMALL PROBLEMS**

---

An element of structured programming is organizing the parts sequentially and clearly. Implied in that organization is the creation of "modules" that are separately developed yet interwoven with the rest of the program. One way to envision this kind of programming is as compartments on a ship. Each compartment is watertight so that if one part is breached, the rest of the compartments will not be flooded. However, each compartment is part of the whole ship; even if one compartment is flooded, it will be necessary to repair it.

The modules or compartments of a program have an analogous function in debugging programs. When a program doesn't work, it must be fixed. With large programs that are *unstructured*, there's no easy way of isolating, identifying and fixing the problem. That's because everything is crammed together in a way that's like a ship with just one big compartment. If there's a hole, the entire ship is flooded. With modules, on the other hand, the problem can be identified in terms of where the program crashes. For example, if something in the data entry portion of the program bombs, then you simply go to the module that handles data entry and look for the problem. Thus, instead of dealing with a giant complex problem, you deal with a small simple one.

At the same time that it is easier to fix a structured program with discrete modules, it is also easier to build a program with modules. As we have seen already, the actual parts of a program deal with fairly simple things: entering data, calculations and output to the screen. Big complex programs have to deal with those parts too, but by breaking them down into modules, even the most forbidding program can be taken one simple step at a time.

## Labelling the Parts

In any manufacturing process, especially ones involving the assembly of many different parts, the various parts must be well labelled. Even the smallest screw is in a bin with a label describing it. Up to this point we've been labelling our programs without really going into the labelling process in detail. This last section will cover the importance of labels as both module headings and program remarks.

**MODULE HEADINGS.** If you're used to programming with line numbers, then you may be familiar with creating module headings by blocking program segments by 100's. For example, the following is a typical block structure with line numbers:

```
100 REM *****
110 REM INPUT
120 REM *****
130 INPUT "What would you like to say";MESSAGE$
200 REM *****
210 REM OUTPUT
220 REM *****
230 PRINT MESSAGE$
```

With the ability to use labels instead of line numbers, we can save a lot of typing and still have clearly labelled segments. By using the TAB key, it is very easy to label your segments in Amiga BASIC. The same program on your Amiga would be written as:

```
Input.Data:
    INPUT "What would you like to say"; MESSAGE$

Output.Data:
    PRINT MESSAGE$
```

Using spaces between your modules further helps the readability of your program listing and aids in finding errors and debugging. To some degree, treat your program like an outline, using the tabs and vertical spacing to add clarity to what you are doing.

**REM AND TICK(').** Now that you can create module headings that describe what's being done, what about REM and the tick mark (')? These too, have their place, especially when you have something tricky you want to describe in your program. For example, you may use a single precision variable in your program, but you can make a note to yourself or another user that the variable should be changed to a double precision one if a greater degree of precision is required. The next program shows how this might look:

Input.Number:

```
INPUT "Please enter value to be divided"; VALUE
    REM - For greater precision change VALUE to VALUE#.
```

Calculate:

```
RESULT = VALUE/23.212
    ' RESULT is single precision and should not be changed.
```

The above two modules illustrate uses for REM and tick (') in addition to the labelled modules. Remember, labels are *to help you*, so use them any way you want that would be helpful. They don't have to get in the way of your programming, and if you prefer, you can add them *after* you've completed the program. You won't appreciate labels until you have a long complex program to wade through where they serve as guideposts.

## SUMMARY

---

The purpose of this chapter is to start you programming in a way that will make it easy for you to create what you want. Structured programming should not be viewed as a straightjacket to hinder creativity, but instead as a tool which will help you realize your creativity. To the extent that structured programming methods help you realize your goals, they are useful. If you find a better way to get the job done, by all means do so. The structured method is simply one way programmers have found to help make the programming task simpler.

All programs can be broken down into some version of: 1) entering information; 2) analyzing information; and 3) displaying information. In this chapter you learned how to enter information with INPUT, INPUT\$,

INKEY\$ and READ/DATA. Before, all data entry was by equating variables (e.g., A=4). Now you can either store data in your program or enter it as it runs. We did little new for analyzing data in this chapter, but we did learn that in addition to PRINT you can also use WRITE to send information to your screen.

Finally, we reemphasized the importance of breaking down a big problem into smaller problems. The module concept is built into Amiga BASIC in the form of using labels instead of line numbers. Each label describes and sets off a module. In later chapters we will see the increasing importance of modulely formed routines, but even relatively simple programs are made easier by using discrete simple modules for getting tasks completed.



# Loops

---

## THE LOOP STRUCTURE

---

In the last chapter we looked at how to set up a modular sequential program. The 'Top Down' structure is an aid to organize our programs so that we can see where we're going and later see what we did. The sequence in top down programming is simply a logical progression through the various tasks. However, there are a lot of times when we need to do the same thing more than once, such as enter names, read data or make calculations. The loop structure allows us to do this without having to rewrite the same thing several times. Diagrammatically, it looks like this:

```
Begin Loop
  Task A
  Loop Limit? (Yes/No)
  If = Loop Limit then go to Next Task
  If <> Loop Limit then go back to Task A
End Loop
Task B
etc.
```

We will look at a number of different loop structures and see how they fit into structured programming and what uses they have. They will significantly add to your programming tools.

## FOR/NEXT LOOPS

---

With the FOR/NEXT loop you can go through a determined number of steps, at variable increments if desired, and execute them until the total number of steps is completed. Let's look at a simple example to get started:

```
=>Click NEW
  Show.Loop:
    CLS
    AM$ = "Amiga"
    FOR X = 1 TO 15: REM BEGIN LOOP
    PRINT AM$
  NEXT X : REM END LOOP
END
```

Now run the program and you will see 'Amiga' printed 15 times along the left side of the screen. Notice also that at the end of the program, we had END. The END statement stops a program. Since your program stops at the end anyway, you may wonder why use it all. As we progress, we'll be ending the program in different modules, depending on the kind of program application. Introducing END at this time is preparatory for later stages.

Let's look at another simple illustration to show what's happening to "X" as the loop is being executed:

```
=>Click NEW
  Count.Loop:
    CLS
    FOR X = 1 TO 15
      PRINT X
    NEXT X
```

As we can see when the program is run, the value of "X" changes each time the program proceeds through the loop. Think of a loop as a child on a merry-go-round. Each time the merry-go-round completes a revolution, the child gets a gold ring, beginning with one and ending, in our example, with 15. Having seen how loops function, let's do something practical with a loop. We'll write a "Checkbook" program.



In our program, we are going to use variables in many ways. First, our FOR/NEXT loop will use a variable. Let's use 'X' as the loop variable name. Second, a variable will be used to indicate the number of loops to be executed. We will use N%, an integer variable. Third, we will use variables for the balance, the amount of the check and the new balance. This program is going to be a little longer, so be sure to SAVE it to disk every 5 lines or so.

=>Click NEW

```

Check.Book:
CLS
CB$ = "Checkbook"
PRINT : PRINT : PRINT CB$
REM *****
REM INPUT INITIAL INFORMATION
REM *****
INPUT "HOW MANY CHECKS"; N%
INPUT "YOUR CURRENT BALANCE" ;BALANCE
REM *****
REM BEGIN LOOP
REM *****
FOR X = 1 TO N%
    PRINT "BALANCE NOW=$";BALANCE
    PRINT "AMOUNT OF CHECK #";X;
    INPUT CHECK : REM VARIABLE FOR CHECK
    BALANCE = BALANCE - CHECK : REM RUNNING BALANCE
NEXT X
REM *****
REM TOP OF LOOP
REM *****
REM
REM *****
REM PRINT OUT FINAL BALANCE
REM *****
CLS : REM CLEAR SCREEN WHEN ALL CHECKS ARE ENTERED
PRINT : PRINT : PRINT
PRINT "YOU NOW HAVE $"; BALANCE ; "IN YOUR ACCOUNT"
PRINT : PRINT " THANK YOU AND COME AGAIN "
END

```

That program would have taken far more steps had we not used the loop structure.

## Nested Loops

With certain applications, it is going to be necessary to have one or more FOR/NEXT loops working inside one another. Let's look at a simple application. Suppose you had two shelves with 10 books on each shelf. You want to make a shelf roster indicating the shelf number (#1 or #2) and book number (#1 through #10). Using a nested loop, we can do this in the following program:

```
=>Click NEW
  CLS
    FOR S = 1 TO 2 : REM S FOR Shelf #
      FOR B = 1 TO 10 : REM B FOR BOOK NUMBER
        PRINT "Shelf #" ; S ; "Book #" ; B
      NEXT B
    NEXT S
  END
```

In using nested loops, it is important to keep the loops straight. The innermost loop (the "B loop" in our example) must not have any other FOR or NEXT statement inside of it. Think of nested loops as a series of fish eating one another, the largest fish's mouth encompassing the next largest and so forth on down to the smallest fish.

Look at the following structure of nested loops:

```
For A = 1 TO N
  FOR B = 1 TO N
    FOR C = 1 TO N
      FOR D = 1 TO N
        NEXT D
      NEXT C
    NEXT B
  NEXT A
```

Note how each loop begins (a FOR statement is executed) and is terminated (encounters a NEXT statement) in a "nested" sequence. If you have ever stacked a set of different sized cooking bowls, each one fits

---

inside the other; that is because the outer edge of one is larger than the next one. Likewise, in nested loops, the “edge” of each loop is “larger” than the one inside it and “smaller” than the one it is inside.

## Loop Steps

Loops can go one step at a time, as we have been using them, or they can step at different increments. The following program “steps” by 10:

```
=>Click NEW
  Step.Loop:
  CLS
  FOR JUMP = 90 TO 200 STEP 10
    PRINT JUMP
  NEXT JUMP
```

This allows you to increment your count by whatever amount you want. You can even use variables or anything else that has a numeric value. For example:

```
=>Click NEW
  Variable.Step:
  CLS
  K = 5 : N = 25
  FOR VAR = K TO N STEP K
    PRINT VAR
  NEXT
```

Go ahead and run the program. “Now just a darn minute,” you may well be thinking to yourself. After the word NEXT, there should be a “VAR” but there is none, right? Well, actually, in Amiga BASIC you really do not need it, and you can save a little memory if you use NEXT statements without the variable name. Even in nested loops, as long as you put in enough NEXT statements, it is possible to run your program without variable names after each NEXT statement. However, it is good programming practice to use variable names after NEXT statements, especially in nested loops, so that you can keep everything straight.

It is also possible to go backwards. In case you draw the last number to be served, try this program:

```
=>Click NEW
Line.Cut:
FOR PLACE = 9 TO 1 STEP -1
  PRINT "Position in line =";PLACE
NEXT PLACE
```

As we get into more and more sophisticated (and useful) programs, we will begin to see how all of these different features of Amiga BASIC are used. You may not see the practicality of a statement initially, but when you need it later on, you will wonder how you could program without it!

## COUNTERS

---

Often you will want to count the number of times a loop is executed and keep a record of it in your program for later use. For example, if you run a program that loops with a STEP of 3, you may not know exactly how many times the loop will execute. To find out, programmers use “counters,” variables which are incremented, usually by +1, each time a loop is executed. The following program illustrates the use of a counter. [Notice: Up to now you’ve been reminded to Click NEW in the Project Menu Bar before entering a new program. Now you’ll have to remember to either do that or enter NEW and press RETURN on your own.]

```
Count.Em:
CLS
FOR X = 4 TO 120 STEP 3
  PRINT X
  COUNT=COUNT + 1 : REM THIS IS THE COUNTER
NEXT X
PRINT : PRINT "LOOP EXECUTED "; COUNT ; "TIMES."
```

The first time the loop was entered, the value of “COUNT” was 0, but when the program got to the fifth line, the value of 1 was added to COUNT to make it 1 (i.e.,  $0 + 1 = 1$ ). The second time through the loop, the value of COUNT began at 1, then 1 was added, and at the top of the loop, the value of COUNT was 2. This continued until the program exited the loop. After all the looping was finished, your COUNT automatically told you how many times the loop was executed. Of course,

---

counters are not restricted to counting loops and they can be incremented by any value, including other variables, you may need. For example, change the fifth line to read:

```
COUNT=COUNT + (B * 3)
```

Run your program again and your “counter total” will be a good deal higher.

## WHILE/WEND LOOPS

---

Another type of loop available in Amiga BASIC is the WHILE/WEND loop. This loop starts with WHILE and keeps looping through WEND (the “top” of the loop) until a zero value (not true) is encountered. This kind of loop is valuable for programs where you do not know the number of times you will need to execute the contents of the loop. Let’s start off with a simple example that illustrates the structure of the WHILE/WEND:

```
Count.Down:
CLS
COUNT = 15 : REM Initialize 'Count'
WHILE COUNT
  PRINT COUNT
  COUNT = COUNT - 1
WEND
```

When you run the program, the numbers from 15 to 1 will be printed to your screen. Essentially the program says, WHILE the value of COUNT is not zero, execute the loop, but when it is zero, exit the loop. After the value of COUNT reached 1, which was then printed, it was decremented to 0 and went to the WEND statement. The WEND statement tested it and found it was zero, so the program exited the loop and ended.

Let’s take a look at a practical application using WHILE/WEND. Let’s say you have just taken a trip, and you have written down all of your expenses, but with the stack of notes and receipts, you do not want to bother counting them. All you want to do is to add them all up and get a total. Using the WHILE/WEND loop, you can do this and get a final result. When you are finished, you just enter 0.

```
Mad.Adder :
CLS
Rem *****
REM SET UP LOOP
REM *****
ADDEMUP = 1 : REM MUST BE NON-ZERO
WHILE ADDEMUP
  INPUT "COST OF ITEM OR SERVICE"; ADDEMUP
  TOTAL = TOTAL + ADDEMUP
WEND
Rem *****
REM PRINT TOTAL
REM *****
CLS
PRINT : PRINT
PRINT "Your total expenses were $"; TOTAL
```

This handy little adding machine will add up your figures, and as soon as you enter a zero, it prints out your total. You may be wondering why your TOTAL was not one more than the actual total. After all, the variable ADDEMUP was initialized as 1; therefore, there should be that extra 1 to your total. Since the first value of the variable ADDEMUP that was entered into TOTAL was what you INPUT, the initial value was never entered into the running total.

### Amiga Notes

#### Don't Jump!!!

If you have to jump out of a loop, don't jump out of a FOR/NEXT loop. There are ways to exit FOR/NEXT loops before the end of the loop using a conditional branch that we'll learn about in the next chapter. For example, if your loop goes to 20, you can jump out before that point. Suppose you wanted to end the looping after only 10 loops, you could do it. However, you'll mess up some invisible things in BASIC that may really louse up your program. So, when you want to jump out of a loop, use a WHILE/WEND loop. (You wonder why anyone would want to jump out of a loop? Read on!)

Another way to use WHILE/WEND is to use the '<>' (not equal to) sign with a string. When something is *not true* it is usually flagged as a

---

'0,' but if the true condition is defined as not true, then the not true becomes true. (That makes weird sense, but don't think about it. We'll look at an example instead.)

```
Goodbye.Loop:
  WHILE SAAY$ <> "GONE"
    BEEP : REM A New Word!!!
    INPUT "Whatdaya say ";SAAY$
    CLS
    PRINT SAAY$
    SAAY$=UCASE$(SAAY$)
  WEND
```

We put in a new word, BEEP, that beeps your Amiga. That is one way to get your attention in addition to a letter prompt. (Also, it nearly drove our program testers nuts, so you can get rid of it if you want.) OK, let's take a look at what we've wrought in this program. We used the variable name 'SAAY\$' since SAY is a reserved word. The WHILE condition states, "As long as SAAY\$ is not the word GONE, wave a 'true' flag, but if SAAY\$ is 'GONE' then wave a 'false' flag." We used the UCASE\$ function to make SAAY\$ upper case so that no matter how you enter the characters, they can be compared with the word GONE. If you had used the word "Gone" or "gone," instead of the all caps, "GONE," you must use the UCASE\$ function to effectively make all versions of the word "gone" acceptable.

In just about every respect, the WHILE/WEND loop acts like the FOR/NEXT loop. It is necessary to have a "flag" to exit the loop, but otherwise you can have nested WHILE/WEND loops and execute various statements and functions within the loop. Just for fun, why don't you change our Checkbook program to use a WHILE/WEND loop instead of a FOR/NEXT loop so that you can put in as many or as few checks as you want.

## SUMMARY

---

This chapter introduced a new structure, the loop. You have begun to see the power of your computer, and we have really begun programming. The loop structure allows us, with a minimal amount of effort, to tell the computer to go through a process several times with a single set of instructions. With FOR/NEXT loops, we can set the parameters of an

operation at any increment we want, and then sit back and let the Amiga go to work for us. Using WHILE/WEND loops, we can tell the computer when to exit the loop without having to set limits at the outset.

However, we have only just begun programming! In the next chapter we will learn the third major programming structure, the branch. Then we will be able to combine all of the structures—sequence, loop and branch—into very sophisticated programs. The more commands we know, the less work it is to write a program.



# Branch Structures

In this chapter we will explore new programming structures that will greatly increase your programming tools. We will be examining some sophisticated techniques, but by taking each a step at a time, you will begin using them with ease. Later, when you develop your own programs, be bold and try out new statements.

One problem new programmers have is a tendency to stick with the simple statements they have learned to get a job done. After all, why use “complicated” statements to do what simpler ones can do? Well, the answer to that has to do with simplicity. If one “complicated” statement can do the work of 10 “simple” statements, which is actually simpler? As you get into more and more sophisticated programming applications, your programs can become longer and subject to more bugs. The more statements you have to sift through, the more difficult it is to find the bugs; therefore, while it is perfectly all right to create a long program using a lot of simple statements as you’re learning, you should begin thinking about trying shortcuts through the use of the more advanced statements.

Related to this issue of maximizing your knowledge of different statements is that of letting the computer perform the computing. This may sound strange at first, but often novices will figure everything out for the computer and use it only as a glorified calculator. In the last chapter you may remember we set up a counter to count the times a loop was executed when we used a STEP 3 loop. We could have figured out how many loops were executed instead of letting the computer do it with the

counter, but that would have defeated the purpose of programming! So, as you learn new statements, see how they can be used to perform the calculations you had to work out yourself.

## BRANCHING

---

So far, all of our programs have gone straight from the top to the bottom with the exception of loops. If our Amiga is to do some real decision making, we must have some way of giving it options. When a program leaves a straight path, it is referred to as either "looping" or "branching." We already know the purpose of a loop, but what is a branch? By using the IF/THEN/ELSE and GOTO statements, we will see. Consider the following program:

```
Start.Up:
  CLS
  PRINT "CHOOSE ONE OF THE FOLLOWING BY NUMBER"
  PRINT
  PRINT "1. Enter Names"
  PRINT "2. Sort Lists"
  PRINT "3. Send to Printer"
  PRINT "4. Find Name"
  PRINT
  INPUT "WHICH? "; X
  CLS
  IF X = 1 THEN GOTO Get.Names
  IF X = 2 THEN GOTO Sort:
  IF X = 3 THEN GOTO Printer:
  IF X = 4 THEN GOTO Find:

Trap:
  GOTO Start.Up
  REM Above LINE IS A 'TRAP' TO MAKE SURE
  REM THE USER CHOOSES 1, 2, 3, OR 4

REM *****
REM BRANCHES
REM *****
```

```
Get.Names:
  PRINT "Enter Names" : END
Sort:
  PRINT "Sort Lists" : END
Printer:
  PRINT "Print on printer" : END
Find:
  PRINT "Find Name" : END
```

As you can see, your computer “branched” to the appropriate place, did what it was told and ended. Not very inspiring, I admit, but it is a clear example. Now, let’s try something a little more practical for your kids to play with in their math homework. While we’re at it, we will introduce another use of TIMER. Notice the ‘Pause’ subroutine. Using a long integer variable, we load the value of TIMER into the variable A&. Then, the time is loaded into B&. In a loop, the subroutine keeps updating B& and comparing it with A&. Where there is a difference of 2, indicating that 2 seconds have passed, the program returns to the main program. By changing the comparable value, it’s possible to make pauses as long as you like. If you want fractions of seconds, use double precision variables and compare fractions.

```
CLS
INPUT "What's your name"; NA$
AG$=" Addition Practice ": PRINT AG$
Get.Problem:
  PRINT : PRINT
  INPUT "ENTER FIRST NUMBER -->" ; A
  PRINT
  INPUT "ENTER SECOND NUMBER-->" ; B
  PRINT
Answer:
  PRINT "WHAT IS "; A ; "+" ; B ;; INPUT C
  IF C = A + B THEN Got.It
  PRINT : PRINT "That's not it."
  PRINT "Try again." : PRINT
GOSUB Pause
GOTO Answer
```

```

Got.It:
  REM *****
  REM CORRECT ANSWER
  REM *****
  PRINT " Your answer is right, "; NA$
  PRINT "That's good work."
  PRINT
Do.It.Again:
  PRINT "Do you want more questions? (Y/N): ";
  AN$ = INPUT$(1)
  AN$=UCASE$(AN$)
  IF AN$= "Y" THEN CLS : GOTO Get.Problem
  CLS : PRINT : PRINT : PRINT
  PRINT "I look forward to seeing you again "
END
Pause:
  REM *****
  REM Two Second Pause
  REM *****
  TIMER ON
  A& = INT(TIMER) : REM Use long integer variable w/ TIMER.
  Check.It:
    B&=INT(TIMER)
    IF(B&-A&) < 2 THEN Check.It
  RETURN

```

The more statements we learn, the more flexibility we have. See if you can change the program so that it will handle multiplication, division and subtraction.

Let's look carefully at our program to learn something about IF/THEN statements. First, note in the 'Do.It.Again' routine, the branch is to clear the screen (CLS) if AN\$ = "Y". If any other response is encountered it ends the program. You may ask why the program did not branch to 'GetProblem' regardless of the response since the "GOTO GetProblem" statement is after a colon, making it a new line. Good point. The reason is that after an IF statement, when the response or condition is null, the program immediately drops to the next LINE. That is, any statement after a colon in a line beginning with an IF statement will only be executed if the condition of the IF statement is met. Secondly, the condition of AN\$ is queried as being a "Y" and not simply a Y without quotes. Since the user

enters a Y and not a "Y," we assume that the program will accept a Y. But remember AN\$ is a "string" and not a numeric variable. Therefore in the setting of the conditional, we must remember what kind of variable we are using. On the other hand, if we used a numeric variable, such as AN or AN%, we could have entered a line such as:

```
IF AN = 1 THEN . . .
```

It is also possible to have an alternative branch with ELSE. Using ELSE is an exception to the rule that if the 'true' condition of an IF is not met, the program drops to the next line. Thus, if you want one of two branches, you can use ELSE for another branch or statement. Look at the following program:

```
CLS
INPUT "CAN YOU SAY 'YEAH' ";Y$
IF Y$="YEAH" THEN E1.Branch ELSE Other.Branch
END
E1.Branch:
  REM *****
  REM BRANCH THEN
  REM *****
  PRINT "YEAH, YEAH, YEAH"
  END
Other.Branch:
  REM *****
  REM BRANCH ELSE
  REM *****
  PRINT "WHAT'D YOU SAY THAT FOR?"
  END
```

Of course, ELSE does not have to branch to a new line. It can execute a statement on its own. For example:

```
CLS
INPUT "ENTER 1 OR ELSE!"; A
REM *****
REM One Line Two Branches
REM *****
  IF A=1 THEN PRINT "ONE" ELSE PRINT "NOT ONE"
```

## COMPUTING WITH RELATIONALS

---

There are several different states of relation or "relationals," that we can use for determining a branch. The following list of relationals can be used with branches:

Symbol	Meaning
=	equal to
<	less than
>	greater than
<>	not equal to
>=	greater than or equal to
<=	less than or equal to

Now let's look at some of these, and see what might be done with them. We'll start with some simple illustrations:

Compare.Size:

```
CLS
INPUT "NUMBER 1-->";A
INPUT "NUMBER 2-->";B
IF A > B THEN Greater:
IF A < B THEN GOTO Less:
IF A = B THEN Equal:
```

Greater:

```
PRINT "NUMBER 1 IS GREATER THAN NUMBER 2" : END
```

Less:

```
PRINT "NUMBER 1 IS LESS THAN NUMBER 2" : END
```

```
Equal : PRINT "THE NUMBERS ARE EQUAL" : END
```

Bar.Hopper:

```
CLS
INPUT "HOW OLD ARE YOU?"; AGE%
IF AGE% >=21 THEN Come.On.In
CLS : PRINT
PRINT "Sorry you're too young!"
END
```

```
Come.On.In:
  REM *****
  REM OLD ENOUGH
  REM *****
  CLS : PRINT : PRINT "Do you come here often?"
  PRINT "I'm a Virgo. What's your sign?"
```

OK, you have an idea how they can be used. Note that relationals work with strings as well as numeric variables and with IF/THEN/ELSE statements. There is also another way to use relationals. Try the following from the immediate mode:

```
A = 10 : B = 20 : PRINT A = B
```

Your computer responded with a 0, right? This is a logical operation. If a condition is false, your Amiga responds with a "0," but if it is true, it responds with a "-1." Now try the following little program:

```
CLS
A = 10
B = 20
C = A > B
PRINT C
```

When you RUN the program, you again get a 0. This is because the variable C was defined as A being greater than B. Since A was less than B the variable C was 0 or "false." Now, let's take it a step further:

```
Relational.Branch:
  CLS
  A = 10
  B = 20
  C = A > B
  IF C = 0 THEN PRINT "A is less than B" : END
  IF C = -1 THEN PRINT "A is bigger than B"
```

Later, we will see further applications of these logical operations of the Amiga. For now, it is important to understand that a true condition is represented by a "-1" and a false condition by a "0."

Sometimes we need to set up more than a single relational. Suppose, for example, that you are organizing your finances into 3 categories of expenses: 1) under \$10; 2) between \$10 and \$100; and 3) over \$100. With our relationals it would be simple to compare input under \$10 and over \$100. But if we wanted to do something in between, we might have some difficulty without added statements. The AND, OR and NOT statements allow us to set ranges with our relationals.

AND. If all conditions are met then true  
 OR. If one condition is met then true  
 NOT. If condition is not met then true

For example:

```
Start:
  CLS
  INPUT "How much -->$"; A
  IF A < 10 THEN Small
  IF A > 10 AND A <= 100 THEN Medium
  IF A > 100 THEN Large
Small:
  REM *****
  REM LESS THAN BRANCH
  REM *****
  PRINT "Piggy Bank Money" : GOTO What.Next
Medium:
  REM *****
  REM IN BETWEEN BRANCH
  REM *****
  PRINT "A night on the town" : GOTO What.Next
Large:
  REM *****
  REM GREATER THAN BRANCH
  REM *****
  PRINT "A trip to Europe"
What.Next:
  REM *****
  REM WHAT NEXT? BRANCH
  REM *****
```



```
PRINT " DO YOU WISH TO CONTINUE? ";
AN$ = INPUT$(1)
IF AN$ <> "Y" AND AN$ <> "N" THEN Listen.Up
IF AN$ = "Y" THEN Start
CLS : PRINT "Goodbye" : END
Listen.Up:
BEEP
PRINT "ANSWER 'Y' OR 'N' PLEASE "
GOTO What.Next
```

In the fifth line we set the conditional branch to be BOTH greater than 10 and equal to or less than 100. The variable “A” had to meet both conditions to branch. Similarly, in the ‘WhatNext’ routine, again using the AND statement, we made sure that the response had to be either “Y” or “N.” If you are very perceptive, you may have asked yourself if there is some fishy format in the program. There are conditional IF/THEN lines that simply say ‘THEN Start’ and things like that. What’s going on? Shouldn’t there be a GOTO statement?

Again, we have slipped in another feature of Amiga BASIC. When using IF/THEN statements, it is possible to drop the GOTO on a branch and simply put in the line number. However, note that we have used GOTO statements elsewhere in the program where no conditional is used within the same line or within a single set of colons. Until you become more familiar with programming you might want to keep your GOTO statements after IF/THEN statements, but that is not required.

You may have another question involving the AND statement in the ‘WhatNext’ routine. In normal English if we say something is not “Y” or “N” sometimes we mean that it must be one or the other, exclusively. However, in programming, if we use OR, we are telling the program to branch if either condition is met. Thus, if we wrote the line as:

```
IF AN$ <> "Y" OR AN$ <> "N" THEN Listen.Up
```

the program would have branched if AN\$ was not equal to *either* “Y” or “N.” Thus if we responded with a “Y,” that “Y” would NOT have been equal to “N” and so the program would have branched to “ANSWER ‘Y’ OR ‘N’ PLEASE”—not what we intended. To check this, change the AND to an OR in the line and RUN the program.

Now, let's use the OR and NOT statements in a program:

```
Find.True:
  CLS
  A=30
  B=40
  C=70
Find.Truth:
  IF A + B = C OR A < B OR A - B = C THEN Truly
  END
Truly:
  REM *****
  REM Anything Can be True
  REM *****
  PRINT "Truth, at last!!"
```

Looking at the 'Find.Truth' routine, we see that  $A - B$  does not equal  $C$ . However,  $A + B$  does equal  $C$  and  $A$  is less than  $B$ . Using the OR statement, only one statement has to be true to branch. Now, let's try the following program:

```
  CLS
  X=15
  Y=20
  Z=97
Go.Figure:
  W=X + Y = Z
  IF NOT W THEN No.No
  END
No.No:
  REM *****
  REM NOT BRANCH
  REM *****
  PRINT "Of course it isn't true"
```

As can be seen from the example, it is possible to use the "negation" of a formula to calculate a branch condition. In most cases, you will use  $\langle \rangle$  (not equal) or the positive case, but at other times it will be simpler to employ NOT.

---

## SUBROUTINES

---

Often there is some operation you will want your computer to perform at several different places in the program. Now, you can repeat the instructions again and again or use GOTO's all over the place to return to your original spot after branching to the operation. On the other hand, you can set up "subroutines" and jump to them using GOSUB and get back to your starting point using the RETURN statement. Up to a point, the GOSUB statement works pretty much like the GOTO statement since it sends your program bouncing off to a line out of sequence. Also, the RETURN statement is something like GOTO since it also sends your program to an out-of-sequence line. However, the GOSUB/RETURN pair is unique in what it does. Let's take a look at a simple example to see how it works:

```
Main.Body:
  CLS
  AMIGA$ = "This is your Amiga" : GOSUB Print.Em
  AMIGA$ = "sending you all kinds of" : GOSUB Print.Em
  AMIGA$ = "Greetings!!!" : GOSUB Print.Em
  END
  REM Be sure to have your END Statement
  REM between your subroutines and end of the body of
  REM your program!!!

Print.Em:
  REM *****
  REM SUBROUTINE CITY
  REM *****
  PRINT AMIGA$
  RETURN
```

Our example shows that a GOSUB statement works exactly like a statement on the line itself except that it is executed elsewhere in the program. The RETURN statement brings it back to the next statement after the GOSUB statement. Using the GOSUB/RETURN pair it is much easier to weave in and out of a program than to use GOTO since the RETURN automatically takes you back to the jump-off point. To better illustrate the usefulness of GOSUB, let's change the 'Print.Em' routine to

something else. Try the following. [NOTE: We will be getting ahead of ourselves a bit with this example, but it will illustrate something very useful about GOSUB's.]

```
Replace: PRINT AMIGA$  
With: SAY TRANSLATE$(AMIGA$)
```

Turn up the sound on your monitor or TV and your Amiga will talk to you. As you can see, a single routine handled all of the talking, and by changing only a single line, you were able to dramatically change what the program did.

### Amiga Notes

#### Now You Can See

By this stage, you should begin noticing how the structure of a program really helps you to understand, debug and create the program. We're getting into longer and longer programs, and we'll be adding other variations on the basic structures and more modules as well. Without clear organization, things would start to get out of hand at this point. Besides, it really isn't that much more work to do it right in the first place.

---

## COMPUTED GOTO AND GOSUB

Now we're going to get a little fancier, but in the long run, it will result in clearer and simpler programming. As we have seen, we can GOTO or GOSUB on a "conditional" (e.g., IF A = 1 THEN PrintIt). The easier way to make a conditional jump is to use "computed" branches using the ON statement. For example:

```
CLS  
Get.Val:  
  INPUT "ENTER A NUMBER FROM 1 TO 5 " ; A  
  IF A < 1 OR A > 5 THEN Get.Val : REM TRAP
```

```

Com.Sub:
  ON A GOSUB One,Two,Three,Four,Five
Query:
  PRINT "Do it again? (Y/N)" ;
  AN$=INPUT$(1) : AN$=UCASE$(AN$)
  IF AN$ <> "Y" THEN END
  GOTO Get.Val
Subs:
  REM *****
  REM FIVE SUBROUTINES!
  REM *****
One:
  PRINT "ONE" : PRINT : RETURN
Two:
  PRINT "TWO" : PRINT : RETURN
Three:
  PRINT "THREE" : PRINT : RETURN
Four:
  PRINT "FOUR" : PRINT : RETURN
Five:
  PRINT "FIVE" : PRINT : RETURN

```

The format for a computed GOSUB/GOTO is to enter a variable following the ON command. The program will then jump the number of “commas” to the appropriate line number. If “1” is entered, it takes the first line number, “2,” the second, and so forth. It’s a lot easier than entering:

```

IF A = 1 THEN GOSUB One
IF A = 2 THEN GOSUB Two
etc.

```

However, it is necessary to use relatively small numbers in the “ON” variable since there is a limited number of subroutines. If your program is computing larger numbers, all you have to do is to convert the larger numbers into smaller ones by changing the variables. For example:

```

Start:
  CLS
  INPUT "ENTER ANY NUMBER--> "; A

```

```

    IF A < 100 THEN B = 1
    IF A >= 100 AND A < 200 THEN B = 2
    IF A >= 200 THEN B = 3
Com.Sub:
    ON B GOSUB Uno,Dos,Tres
    REM B is compute variable
Query:
    PRINT "Do it again?(Y/N)";
    AN$ = INPUT$(1)
If AN$ <> "Y" THEN END ELSE Start
E1.Subos:
    REM *****
    REM SUBROUTINE NET
    REM *****
Uno:
    PRINT "LESS THAN 100" : RETURN
Dos:
    PRINT "MORE THAN 100 BUT LESS THAN 200 " : RETURN
Tres:
    PRINT "MORE THAN 200" : RETURN

```

RUN the program and enter any number you want. Since the program is branching on the variable B, and not on A (the INPUT variable), you will not get an error since the greatest value of B can only be 3.

Now let's get back to relationals and see how they can be used with computed GOSUB's. Remember, in using relationals, the only numbers we get are 0's and 1's for false and true respectively. However, we can use these 0's and 1's just like regular numbers. Try the following:

```

CLS
Equates:
    X = 1 : Y = 2 : Z = 3
    A = X < Z
    B = Y > Z
    C = Z > X
Results:
    PRINT "A + A =" ; A + A
    PRINT : PRINT "A + B =" ; A + B
    PRINT : PRINT "A + B + C =" ; A + B + C
END

```

Now before you RUN the program, see if you can determine what will be printed in the 'Results' segment. Once you have made a determination, RUN the program and see what happens. Go ahead and see how you do. Let's go over it step by step.

1. Since X is less than Z, A will be "true" with a value of one (-1). Therefore  $A + A$  ( $-1 + -1$ ) will equal  $-2$ .
2. Since Y is not less than Z ( $Y = 2$  and  $Z = 3$ , remember), B will be "false" with a value of 0. Therefore,  $A + B$  ( $-1 + 0$ ) will total  $-1$ .
3. Since Z is greater than X, C will be "true" with a value of  $-1$ . Therefore  $A + B + C$  ( $-1 + 0 + -1$ ) will equal  $-2$ .

If you got it right, congratulations! If not, go over it again. Remember, very simple things are happening, and so don't look for a complicated explanation! Now that we see how we can get numbers by manipulating relationals, let's use them in computed GOSUB's. The following program shows how:

```

Bean.Calc:
  CLS
  INPUT "How many beans in the jar?";HM
Get.Beans:
  BEANS = 1 + (HM >= 500) + (HM >= 1000)
  IF BEANS = 0 THEN BEANS = 2
  IF BEANS = -1 THEN BEANS = 3
  ON BEANS GOSUB Some,More,Most
Again:
  PRINT : INPUT "More bean counting (Y/N) "; AN$
  AN$=UCASE$(AN$)
  IF AN$ <> "Y" THEN END ELSE Bean.Calc
Bean.Subs:
  REM *****
  REM SUBROUTINES
  REM *****
Some:
  CLS : PRINT "Not many beans- Less than 500"
  RETURN
More:
  CLS : PRINT "Nice bunch of beans - between 500 and 1000."
  RETURN

```

Most:

```
CLS : PRINT "That's over 1000 beans"  
RETURN
```

The program is hinged on the 'Get.Bean' formula or algorithm. Let's see how it works:

1. There are 3 conditions:
  - a. HM is less than 500
  - b. HM is 500 or more but less than 1000
  - c. HM is 1000 or greater
2. If the first condition exists, both  $HM \geq 500$  and  $HM \geq 1000$  would be false. Thus  $1 + 0 + 0 = 1$ . Therefore BEANS = 1.
3. If HM is  $\geq 500$  but less than 1000, then  $HM \geq 500$  would be true but  $HM \geq 1000$  would be false. Thus we would have  $1 + (-1) + 0 = 0$ . Convert the value of BEANS to 2.
4. Finally if HM is both  $\geq 500$  and  $\geq 1000$ , then our formula would result in  $1 + (-1) + (-1) = -1$ . Convert the value of BEANS to 3.

### Amiga Notes

#### Right is Right

We've discussed using structured programming to help in your programming tasks, but *remember* that structured programming is a means to an end. If there is ever a choice between getting a program to work and having things in structured programming format, most programmers will choose getting it done. Sometimes we get so wrapped up in structured programming that we tend to forget it is a way to help us. As long as a program does what it's supposed to do, whether it follows the tenets of structured programming or not, it is right.

---

## STRINGS AND RELATIONALS

Before we leave our discussion of computed GOTO's and GOSUB's with relationals, let's take a look at how relationals handle strings. Try the following:



```
A$ = "A" : B$ = "B" : PRINT B$ > A$ <RETURN>
```

Surprised? In addition to comparing numeric variables, relationals can compare alphabetic string variables with “A” being the lowest and “Z” the highest. (Actually, any string variables can be compared, but we will just look at the alphabetic ones here.) So if we ask if B\$ is greater than A\$, we get a “-1” (true) since B\$ was a B and A\$ was an A. Now you might be wondering what on earth you could possibly want to do with this knowledge. Well, in sorting strings (like putting names in alphabetical order) such an operation is crucial. Later on we will show you a routine for sorting strings, but for now let’s make a simple string sorter for two strings:

```
CLS
INPUT "WORD" #1 --> " ; A$
INPUT "WORD" #2 --> " ; B$
PRINT : PRINT : PRINT
IF A$ < B$ THEN PRINT A$ : PRINT B$
IF A$ > B$ THEN PRINT B$ : PRINT A$
```

Just what you needed! A program that will put two words in alphabetical order!

## SUBPROGRAMS

---

This section will *introduce* the concept of subprograms and show some new tricks. However, we’re starting to get into some advanced concepts at this point, so instead of using more complex program examples, we’ll use very simple subprograms to focus on how subprograms work. Thus, while the examples may appear to be trivial, we hope you can grasp the significance of subprograms. Finally, we will only show how to use BASIC subprograms even though it is possible to write machine language subprograms.

To begin we’ll show a simple subprogram that uses two variables to keep a running total of numbers passed to it.

```
Enter.Num:
  INPUT "Enter value"; VALUE
  CALL Addemup (VALUE,TOTAL)
```

```
PRINT "Total=" ; TOTAL
IF VALUE <> 0 THEN Enter.Num
REM *****
REM SUBPROGRAM
REM *****
Run.Total:
SUB Addemup (A,B) STATIC
  B=B+A
END SUB
```

In order to understand the significance of what happens in the above program, consider the variables 'VALUE' and 'TOTAL' that are part of the 'Enter.Num' routine and the variables 'A' and 'B' that are part of the 'Run.Total' routine. Step by step, let's see what happens:

1. You input a number in the variable 'VALUE.'
2. The program calls the subprogram 'Addemup' with the variables VALUE and TOTAL.
3. The subprogram uses the *parallel* variables A and B to calculate VALUE and TOTAL. Variable 'A' parallels VALUE and 'B' parallels TOTAL.
4. The subprogram calculates 'B' as a running total and passes the value of 'B' back to the parallel variable TOTAL.
5. The value of TOTAL is printed to the screen.
6. The program loops back to 'Enter.Num' until you enter a zero.

The interesting thing about subprograms is the use of *local variables*. The variables in the subprogram that are declared STATIC are affected only by the most recent call and retain their values. For example, in our example, the variables 'A' and 'B' only use the last values passed to them. They remain, in effect, zero, until they are called. This is useful in programs that have *different* variables used in calling the *same* subprogram. With subroutines, it is necessary to use a common variable name, but with subprograms, you can use different variable names, and then pass their values to parallel variables in the subprogram. For example, add the following routine to the above program. (Just tack it on to the end of the program.)

```
Loop.Add:
FOR X = 1 TO 4
  CALL Addemup (X,SUM)
  PRINT"Sum=" ; SUM
NEXT X
```

In the 'Loop.Add' routine, the variables 'X' and 'Sum' are used with the parallel variables 'A' and 'B', and they work just fine. So the subprogram can deal with any variables sent their way as long as the subprogram is correctly called.

When passed from the main routine, variables are called *arguments*; parallel values in the subprogram are called *formal parameters*. The following outlines the connection between the two:

<b>Main Program</b>	<b>Subprogram</b>
Arguments (W,X,Y\$)	Formal Parameters (A,B,C\$)

By aligning the arguments in the main program with the formal parameters in the subprogram, all operations are easily handled by the subprograms. It requires careful planning and a well structured program, but in the long run, it saves a lot of unnecessary redundancy. Just remember to line up the variables so that they are parallel.

**SHARED VARIABLES.** Besides having local or static variables in subprograms, it's possible to have 'global' or shared variables as well. These are variables that have the same values in the main program and subprogram. The values in both the main and subprograms are affected by one another. Any changes in one part will reflect changes in the other. Let's look at a program that uses *both* shared and static subprograms to see what happens differently between shared and static variables.

Two.Types:

```
CALL ONE (A,B)
PRINT A,B
```

```
CALL TWO
PRINT X,B
```

```
SUB ONE (X,Y) STATIC
  X=10
  Y=20
END SUB
```

```
SUB TWO STATIC
  SHARED B
  X=30
  B=B + 40
END SUB
```

When you run the program, you will see,

```
10      20
0       60
```

The first subprogram passed values between A and B and X and Y in the pattern of local variables we've already seen. However, when we attempted to pass the value of the variable 'X' from SUB TWO, we got a zero. That's because there was no parallel variable for X in SUB TWO as there was in SUB ONE. However, by making 'B' a shared variable, we took the value it generated in the main program in interaction with the first subprogram and passed it back and forth between the main and subprogram in SUB TWO.

This may be a bit confusing at first, and the difference between shared and static variables in subprograms may not be clear at this point. However, with practice and experimentation, you will soon find that you can put subprograms to very good use. Also note that both shared and static subprograms have the word `STATIC` in them. That is more a matter of form than description. It is possible to have a combination of static and shared variables in the same subprogram. Only those declared as shared will be so while the rest are assumed to be static.

### Amiga Notes

#### Experimenting with New Tools

The only way to really understand new statements and tools in programming is to try things out with them. Let's face it, at this point it's a lot easier to do calculations or other subroutine chores with something else than subprograms. Besides, you can probably write a more efficient program using non-subprograms now anyway. However, unless you 'test drive' the new programming techniques you have available, you may not remember or understand them when you do need them. Keep refining your skill by testing out every single programming statement, function and algorithm until you at least know what it does. Then when you do need them, at least you'll remember to give them a trial run and you won't be stuck without the right tool.

There are more features to subprograms, and we will cover these features as we progress. At this time, it is enough to see how local

variables operate within a program, and how shared variables differ from local variables. The final feature of subprograms we will deal with in this section is the combination of shared and local variables within the same subprogram. In 'SUB ONE' and 'SUB TWO' in the above program, we looked at two subprograms using exclusively shared or static variables within the same program, and now we'll see how that is done within the same subprogram.

```
Dual.Sub:
  INPUT "Enter Name"; NA$
  INPUT "Enter Address":AD$
  INPUT "Years at address";YEARS
  CALL Go.Fig (NA$,YEARS)

SUB Go.Fig (WHO$,TIEMPO) STATIC
  SHARED AD$
  CLS
  PRINT WHO$;" lives at ";AD$
  PRINT "They've been there for";TIEMPO';" years."
END SUB
```

The 'Go.Fig' subprogram has the static variables WHO\$ and TIEMPO aligned with NA\$ and YEARS, respectively. However, AD\$ is declared to be a shared variable in the same subprogram; so the subprogram has both static and shared variables.

## SUMMARY

---

We covered a good deal in this chapter, and if you understood everything, excellent! If you did not, don't worry, for with practice, it will all become very clear. Whatever your understanding of the material, though, experiment with all the statements. Remember to experiment with your computer's commands, and as long as you have a practice disk on which to try out your skills, the worst that can happen is that you will erase a few programs.

We saw how your Amiga computer can "make decisions" using the IF/THEN/ELSE statements and relationals. Using subroutines it is possible to branch at decision points to anywhere we want in our program. Computed GOTO's and GOSUB's allow the execution to move ap-

propriately with a minimal amount of programming. Subroutines also help us organize our programs more efficiently and save time.

Finally, we examined subprograms. Unlike subroutines, subprograms can employ *local variables*, and so different parts of your program can use the same subprogram even though the variable names are different. That way it's possible to have the same routine take care of more chores. Since subprograms can also handle shared variables, you can pass values back and forth between your subprograms and main program. There's a lot more to subprograms than we discussed in this chapter, but we'll get to these other parts as we go along.

# Arrays

---

## ARRAYS AS GROUPED VARIABLES

---

Sometimes novice programmers have problems understanding arrays, so we'll take it slowly in this chapter. Usually, the confusion about arrays stems from the novice overcomplicating things. In fact, arrays are very simple, and if you keep that in mind you'll be using them with ease in no time at all.

The best way to think about arrays is as a kind of variable. As we have seen, we can name variables SUM, Nairobi\$, KK%, X1 and so forth. An array uses a single name with a number to differentiate different variables. Consider the following two lists, one using regular string variables and the other using a string array:

<b>Variable</b>	<b>Array</b>
Tree\$="Pine"	Plant\$(1)="Pine"
Bush\$="Hedge"	Plant\$(2)="Hedge"
Flower\$="Rose"	Plant\$(3)="Rose"
Fruit\$="Plum"	Plant\$(4)="Plum"

If we print 'Bush\$' or 'Plant\$(2),' the Amiga would respond with 'Hedge' since in both cases, the word 'Hedge' is stored in those variables. Likewise, we could use arrays for numeric variables such as:

```
NUM(1) = 2
NUM(2) = 4
NUM(3) = 6
NUM(4) = 8 etc.
```

Again, you may well ask, "So what? Why not just use regular numeric or string variables instead of arrays?" Well, for one thing, it can be a lot easier to keep track of what you're doing in a program by using arrays, and it can also save a lot of time. Consider the following program for INPUTting a list of 10 names using a string array:

```
Quick.Name:
  CLS
Name.In:
  FOR X = 1 TO 10
    PRINT "Name #"; X ;
    INPUT NAM$(X)
  NEXT X
Name.Out:
  CLS
  FOR X = 1 TO 10
    PRINT NAM$(X)
  NEXT X
```

Now, write a program that does the same thing using non-array variables. It would take a lot more coding to do so, but go ahead and try it. Use the variables N0\$ through N9\$ for the names. If you rewrote the program, you saw how much time was saved by using arrays. But before going on, let's take a closer look at how the program worked with the FOR/NEXT loop and array variable:

1. The FOR/NEXT loop generated the numbers sequentially so that the array would be the following:

```
FOR X = 1 TO 10
  NAM$(1) <--First time through loop
  NAM$(2) <--Second time through loop
  NAM$(3) <--Third time through loop
  NAM$(4) etc.
  NAM$(5)
  NAM$(6)
```



```
NAM$(7)
NAM$(8)
NAM$(9)
NAM$(10)
NEXT X
```

2. Each string INPUT by the user was stored in a sequentially numbered array variable.
3. Output, using the PRINT statement, was generated by the FOR/NEXT loop sequentially supplying numbers to be entered into array variables. Now, to get used to the idea that an array variable is a variable, enter the following:

```
V(10) = 432 : PRINT V(10) <RETURN>
XYZ(9) = 2.432 : PRINT XYZ(9) <RETURN>
TOOT$(1) = "BEEP!": + CHR$(7) : PRINT TOOT$(1) <RETURN>
J%(5) = 321 : PRINT J%(5) <RETURN>
```

It's easy to forget and think of arrays as something more exotic than they are. Just remember they're nicely organized variables.

## THE DIMENSION OF AN ARRAY

In our array examples we haven't gone over the number 10. The reason is that once our array is larger than 10 we have to use the DIM (dimension) statement to reserve space for our array. (To be exact, 11 array elements are automatically numbered by dimension from 0 to 10.) The following is an example of the format for DIMensioning an array:

```
Big.Array:
DIM NUMS(220)
FOR X = 1 TO 220
  NUMS(X) = X
NEXT X
FOR X = 1 TO 220
  PRINT NUMS(X);
NEXT X
```

Run the program as it is written; it should work fine. Now delete the line with the DIM statement and run the program again. You will get a 'Subscript out of range' error for not DIMing the array. That means your array went over 10 with no DIM statement, and thus it is 'out of range.' If your array went over 220 in the above program it would also be 'Out of range' since the DIM statement only gives it an upper boundary of 220. Whenever your arrays are going to have more than 11 values from 0 to 10, be sure to DIM them, and make sure you keep within the limits of the programs you do DIM.

### Amiga Notes

#### DIM It All

Many programmers always DIM arrays, regardless of the number of elements in the array. It is perfectly all right to do so, and statements such as DIM X\$(3) or DIM N% (5) are valid. Often when copying programs from books or magazines you may run across these lower level DIM statements because the programmer thinks it's a good idea to DIM all arrays as part of programming style and clarity. Furthermore, you can save memory space by using the minimal amount of DIMension space, and if the program is large enough, it may be necessary to DIM and array at less than 11. Finally, some old versions of BASIC require all arrays to be DIMensioned.

---

## KEEPING IN BOUNDS

A couple of possibly useful functions for working with arrays are LBOUND and UBOUND. They stand for 'lower bounds' and 'upper bounds' respectively, and they return the lowest and highest value for an array. Above we noted that arrays begin at 0 (zero) instead of 1 (one). Actually, you can set the base of an array to be 0 or 1 using OPTION BASE. It would be correct to say the *default* base value of an array is 0. The following is the entire range of options for OPTION BASE:

```
OPTION BASE 0
```

```
OPTION BASE 1
```

You can save a byte or so of memory using `OPTION BASE 1` since most programmers start with 1 instead of 0 in using their arrays, but for the most part, we will simply go with the default of 0. (So we'll waste a byte or so.)

Since `LBOUND` and `UBOUND` return the *values* of the array boundaries, you can use them to automatically fill the limits of your array. The general format is:

```
LBOUND(Array.Name)
```

For example, the following program “automatically” cranks out the entire array simply by placing the `LBOUND` and `UBOUND` functions in a loop:

```
Unbounded.Passion:
  DIM Love(100)
  WIDTH 62 : REM Keep it on the screen
  FOR X= LBOUND(Love) TO UBOUND(Love)
    Love(X)=X
    PRINT Love(X);
  NEXT
```

Notice that the array generated 101 values (0–100) since the upper boundary is 100 and the default lower boundary is zero. Now, insert `OPTION BASE 1` right above the line with the `DIM` statement and run the program again. This time, there's only 100 values (1–100).

Using the `LBOUND` and `UBOUND` functions, you can link your loops directly to the size of your array. It may be more convenient to store their values in variables right after you dimension the array. Furthermore, you can use a variable to dimension an array. The following program shows you how:

```
Shop.List:
  OPTION BASE 1
  INPUT "How many items to buy ";ITEMS
  DIM SHOP$(ITEMS) : REM Using variable for DIM
  LOSHOP = LBOUND(SHOP$)
  HISHOP = UBOUND(SHOP$)
```

```

Make.List:
  FOR X = LOSHOP TO HISHOP
    INPUT "Item to buy ";SHOP$(X)
  NEXT X

Print.List:
  CLS
  FOR X = LOSHOP TO HISHOP
    PRINT SHOP$(X)
  NEXT X

```

Using the above method makes it a little easier to use the boundary functions. The variables 'LOSHOP' and 'HISHOP' are descriptive, making it simple to remember to what array they belong.

## BUFFERS AND ARRAYS

Just about any variable or array is a buffer. Buffers are simply temporary storage places used until you print something on your screen or put it somewhere more permanently. (Later, we'll see how to store data on disks or send it over the phone lines by moving data into and out of buffers.) Modular structured programming lends itself to using arrays as buffers since each module is a task unto itself. As each task is being performed, information can either be taken from or placed into a buffer. If an array is used as a buffer, it can be maintained as a set of unique variables to be used whenever needed. Consider the difference between using a single array buffer and single variable buffer:

Single Variable	Single Array
Data to variable	All data to buffer
Data to screen	All data from buffer to screen
Data to variable	
Data to screen	
Data to variable	
Data to screen	
etc., ad nauseam	

To see how useful an array buffer is, look at this next program. It is divided into three main modules:

1. Data entered into buffer
2. First task with buffer
3. Second task with buffer

As you will see, once the data is in the array buffer, you can do different things with it:

Show.And.Find:

```
INPUT "How many names to enter "; N
OPTION BASE 1
DIM Buffer1$(N),Buffer2$(N)
LOBUFF=LBOUND(Buffer1$)
HIBUFF=UBOUND(Buffer1$)
```

Enter.Data:

```
FOR X=LOBUFF TO HIBUFF
  INPUT "Name ";Buffer1$(X)
  INPUT "Phone Number ";Buffer2$(X)
NEXT X
```

Show.Data: 'Task 1

```
CLS
FOR X=LOBUFF TO HIBUFF
  PRINT Buffer1$(X),Buffer2$(X)
NEXT X
```

```
PRINT : PRINT
PRINT "Hit any key to continue "
HIT$=INPUT$(1)
```

Find.Data: 'Task 2

```
CLS
INPUT "Find what person's number ";NA$
L=LOBUFF : H=HIBUFF
WHILE Flag$ <> "Gone"
  IF L > H THEN GOSUB Not.Found
  IF NA$=Buffer1$(L) THEN GOSUB Found
  L=L+1
WEND
```

```
END
```

```

Found:
  Flag$="Gone"
  PRINT NA$;" 's number is ";Buffer2$(L)
  RETURN

Not.Found:
  Flag$="Gone"
  L=L-1
  PRINT NA$;" 's number is not here."
  RETURN

```

That was a long program, but with everything in blocks, it's easy to see what happened. First, all the data were placed in an array. After that, the information in the array was printed on the screen in the 'Show.Data' block, and then searched for in the 'Find.Data' block. Notice how the search routine worked. It was simple, yet efficient. By comparing the name you entered with the names in the buffer, it was able to quickly find the phone number. Later on in the book when we examine files and how to store information in them, we will use buffers to read and write data to your disk.

## MULTI-DIMENSIONAL ARRAYS

So far, all we have examined are single dimension arrays. However, it is possible to have arrays with two or more dimensions. Let's begin with two-dimensional arrays and examine how to use arrays with more than a single dimension. The best way to think of a two-dimensional array is as a matrix. For example, if our array ranged from 1 to 3 on two dimensions the entire set would include: V(1,1) V(1,2) V(1,3) V(2,1) V(2,2) V(2,3) V(3,1) V(3,2) and V(3,3). By laying it out on a matrix, we can think of the first number as a row and the second as a column. This makes it much clearer:

	Column #1	Column #2	Column #3
ROW #1	V(1,1)	V(1,2)	V(1,3)
ROW #2	V(2,1)	V(2,2)	V(2,3)
ROW #3	V(3,1)	V(3,2)	V(3,3)

Again, it is important to remember that each element in the array is simply a type of variable. To drum that into your head do the following:

```
TWODEE$(3,1)= "Two dim array" : PRINT TWODEE$(3,1)
<RETURN>
ING%(2,2) = 81 : PRINT ING% <RETURN>
FP(1,1) = 3.212 : PRINT FP(1,1) <RETURN>
```

To use arrays to their fullest advantage in programs, they must be envisioned as an orderly set of variables and not something else. Now, let's use a two dimension array in a program. This first one will sequentially fill the array with numbers and print them on the screen. We will use the variable names 'ROW' and 'COLUMN' for the loops and the array name 'RC':

```
Two.Dim.Array:
FOR ROW = 1 TO 2
  FOR COLUMN = 1 TO 3
    COUNT=COUNT+1
    RC(ROW,COLUMN)=COUNT
    PRINT RC(ROW,COLUMN),
  NEXT COLUMN
  PRINT : REM Start new row
NEXT ROW
```

As you will see when you run the program, the values generated by the variable 'COUNT' were stored in the two dimension array 'RC.' We could have used variables or a single dimension array to do the same thing since all an array does is to give the programmer an easier and more systematic way of naming variables. When you need a way of changing or defining a value in a more complex program, these multiple dimensions will be very valuable. Let's look at another program that uses a two dimension array to organize data. Note how the DIM statement is used with two dimensions and how the DATA statements are 'reused' by resetting them with RESTORE. Our program will line up 16 playing cards in four columns. We'll use the ace and three face cards of the four suits:

```
Dealer:
DIM CARD$(4,4) : 'Four suits and four cards
```

```
Read.Data.Into.Array:
  FOR S=1 TO 4 : 'Suits
    FOR FC=1 TO 4 : 'Face cards
      READ CARD$(S,FC)
    NEXT FC
  RESTORE
NEXT S

DATA Jack,Queen,King,Ace

Shuffle:
  FOR S=1 TO 4: 'Suits
    ON S GOSUB Clubs,Diamonds,Spades,Hearts
    FOR FC=1 TO 4: 'Face cards
      PRINT CARD$(S,FC),
    NEXT FC
  PRINT
NEXT S
REM To be continued . . .
END

Clubs:
  PRINT "Clubs",
  RETURN
Diamonds:
  PRINT "Diamonds",
  RETURN
Spades:
  PRINT "Spades",
  RETURN
Hearts:
  PRINT "Hearts",
  RETURN
```

When you RUN this program, all of your cards will be lined up. However, you could have done the same thing with variables or a single dimension array since all that “lines them up” is the use of the comma to format the PRINT statement. To see how the two dimension array may be more useful, let’s see how it can be used to “pick a card.” Insert the following lines to your program right after the line



```
REM To be continued . . . .
```

and run it again.

```
PRINT "Hit any key to continue"  
HIT$ = INPUT$(1)  
CLS  
INPUT "Enter row and column number ";ROW,COLUMN  
CLS  
PRINT "That card is the "; CARD$(ROW,COLUMN); " of "  
ON ROW GOSUB Clubs,Diamonds,Spades,Hearts
```

Note how we used a single INPUT to enter two values. We used the format:

```
INPUT A, B
```

and as long as the operator (program user) is told to enter the appropriate number of responses and separate each response with a comma, everything will work fine. Remember that you must place a comma between the values you enter when prompted to provide a row and column number (e.g., you would enter 2,3 and then press RETURN). Now you can locate the value or contents on a specific array on two dimensions.

The use of two-dimensional arrays in problems dealing with matrixes is an important addition to your programming commands. It is also possible to have several more dimensions in an array variable. As you add more and more dimensions, you have to be careful not to confuse the different aspects of a single array. Sometimes, when a multi-dimensional array becomes difficult to manage, it is better to break it down into several one- or two-dimensional arrays. Just for fun, let's see what we might want to do with a three-dimensional array with the following program:

```
Librarian:  
INPUT "How many cases ";CASES  
INPUT "How many shelves on each case ";SHELVES  
INPUT "How many modules per shelf ";MODULES  
DIM BOOK$(CASES,SHELVES,MODULES)
```

Place.Books:

```
FOR C=1 TO CASES
  FOR S=1 TO SHELVES
    FOR M=1 TO MODULES
      PRINT "What book is in case";C;
      PRINT " shelf";S;
      PRINT " module";M;
      INPUT BOOK$(C,S,M)
    NEXT M
  NEXT S
NEXT C
```

Find.Book:

```
CLS
INPUT "Which book to find"; TITLE$
FOR C=1 TO CASES
  FOR S=1 TO SHELVES
    FOR M=1 TO MODULES
      IF BOOK$(C,S,M)=TITLE$ THEN GOSUB Found.It: Flag=1
    NEXT M
  NEXT S
NEXT C
IF Flag=0 THEN PRINT "Title not found" : GOTO Another
END
```

Found.It:

```
PRINT TITLE$; "is in case";C; "shelf";S;
PRINT " module";M
```

Another:

```
PRINT : PRINT : Flag=0
PRINT "Find another book(Y/N) ";
```

```
ANS=INPUT$(1)
ANS=UCASE$(ANS)
IF ANS="Y" THEN Find.Book ELSE End
```

Now that was a pretty long program, so go over it carefully to make sure you understand what it is doing. Again, let me remind you that a three-dimensional array is simply a variable with a lot of numbers in

---

parentheses. It would be a good idea to save this program on a disk as an example of a multi-dimensional array.

## ARRAYS IN SUBPROGRAMS

---

Arrays in subprograms work pretty much like variables with subprograms but the format is different. The array is set up in the main program, just as you would any array. You must DIM your array in the main program no matter how many elements it has, but whatever variable name is used in the subprogram does not have to be dimensioned again. In other words, all arrays must be defined with a DIM statement in the main program, but not in the subprograms.

When you call the subprogram, the array name is followed by a pair of parentheses. For example, if you use the single dimension array 'A,' it would be formatted as follows in a CALL:

```
CALL ASUB (A())
```

In the subprogram, the number of dimensions in the array is declared in parentheses after the array name. For example, a single dimension array named 'W' would appear as follows:

```
SUB MARINE (W(1))STATIC
```

To see how an array might look and work, the following program has two different arrays in the main program that use a single subprogram array:

```
WIDTH 62
First.One:
  DIM MAIN(20)
  CALL SUBBY (MAIN())
  FOR X=1 TO 20
    PRINT MAIN(X),
  NEXT X
  PRINT :PRINT

SUB SUBBY (ASUB(1)) STATIC
C=0
```

```
FOR S=2 TO 40 STEP 2
  C=C+1
  ASUB(C)=S
NEXT S
END SUB
```

Second.One:

```
DIM OTHER(20)
CALL SUBBY(OTHER())
FOR X=1 TO 20
  PRINT OTHER(X);
NEXT X
```

In the subprogram, the variable 'C' was set to zero at the beginning. This was necessary since 'C' was not called for by either main routine. Had we not declared "C" as zero at the start of the subprogram, the second time the subprogram was called, its value would have begun at 20, since that's what it was after the first main routine called it. You may wonder why it did not reset to zero since all the variables in the subprogram are local. If any variable had been passed to 'C' in the second CALL, it would have reverted to zero, but since it was left alone, it maintained its value after the first call. This is useful to know in case you want to use the changed value of a local variable after an initial CALL by another part of the program, or if you want to avoid problems caused by the same thing. To see what happens if 'C' is not reset to zero, remove the line C=0 to see what happens.

Multi-dimensional arrays and subprograms work just like single dimension arrays except that the array number is changed from 1 to something else. This next program shows you how to set up and use multiple dimension arrays with subprograms:

Doubasub:

```
DIM DOUB(2,3)
CALL TWODEE (DOUB())
FOR X=1 TO 2
  FOR Y=1 TO 3
    PRINT DOUB(X,Y),
  NEXT Y
PRINT
NEXT X
```

```
SUB TWODEE (OK(2)) STATIC
FOR L=1 TO 2
  FOR M=1 TO 3
    OK(L,M)=L+M
  NEXT
NEXT
END SUB
```

## SUMMARY

---

Arrays allow us to enter values into sequentially arranged variables (or elements). By using FOR/NEXT and WHILE/WEND loops it is possible to quickly program multiple variables up to the limits of a specified dimension. Not only do arrays assist us in keeping variables orderly, they eliminate a good deal of work as well. In the next chapter, we will begin working with commands that help arrange everything for us. As our programs become more and more sophisticated, we will need to keep better track of what we're doing. By organizing our programs into small, manageable chunks, we can create clear, useful programs.



# Manipulating Strings

---

## SUBSTRINGS: PARTS AND WHOLE

---

Up to this point our programming has involved “whole” strings. That is, whatever we define a string to be no matter how long or short, it can be considered a “whole” string. For example, if we define JUMP\$ as “Jump” then we can consider “Jump” to be the whole of JUMP\$. If we defined JUMP\$ as “I have a lot to say about that,” then “I have a lot to say about that” would be the whole string of JUMP\$. There will be occasions, however, when we want to use only part of a string or tie several strings together. (When you work with files, you will find this to be very important.) Also, there are applications where we will need to know the length of strings, find the numeric values of strings and even change strings into numeric variables and back again.

### Amiga Notes

#### These Are Important

At this point in your programming career, you may not appreciate how important partial strings or “substrings” are. It would seem that if you need a part of a string, then just define a whole string that would take care of the part. That makes sense, but you actually save time by having strings broken up and put back together again. Your computer will do most of the work for you, and once you see what can be done with substrings, you’ll be glad you have them.

## FORMATTING STRINGS

We will divide our discussion of string formatting into four parts: 1) calculating the length of a string; 2) locating parts of strings; 3) changing strings to numeric variables and back again; and 4) tying strings together (concatenation).

### How Long Is the String?

Sometimes it is necessary to calculate the length of a string for formatting output. Your Amiga is very good at telling you the length of a particular string. By the statement, PRINT LEN (A\$) you will be told the number of characters (including spaces) your string has. Try the following little program to see how this works:

```
Get.String:
CLS
INPUT "String Name-> "; SN$
PRINT SN$; " has "; LEN(SN$); " letters"
PRINT : PRINT " Another string?(Y/ N) ";
AN$ = INPUT$(1)
AN$=UCASE$(AN$)
IF AN$ = "Y" THEN Get.String
```

Now, to see a more practical application, let's see how to center a string on your screen:

```
WIDTH 62
String.Em.Along:
PRINT "Enter string with less than 63 characters"
INPUT "-> ";CEN$
HALF=31-LEN(CEN$)/2
PRINT TAB(HALF);CEN$
More.Stuff:
PRINT : PRINT "Hit any key for more or 'E' to exit"
HIT$=INPUT$(1) : HIT$=UCASE$(HIT$)
IF HIT$ <> "E" THEN String.Em.Along
```

Now that we can see how to compute the LENGTH of a string and then use that LENGTH to compute our tabbing, let's see how we can



---

control the input with the LEN statement. Suppose you want to write a program that will print out mailing labels, but your labels will only hold 33 characters. You want to make sure all of your entries are 33 or fewer characters long, including spaces. To do this we will write a program that checks the LENGTH of a string before it is accepted:

```
Check.Size:
  PRINT
  INPUT "Enter name: 33 characters or less";NA$
  IF LEN(NA$) > 33 THEN GOSUB Too.Long : GOTO Check.Size
  PRINT NA$; " is an acceptable size.

MORE:
  PRINT "More(Y/N)";
  MORE$=INPUT$(1) : MORE$=UCASE$(MORE$)
  IF MORE$="Y" THEN Check.Size ELSE END

Too.Long:
  PRINT : BEEP
  PRINT "Please use only 33 characters or less"
  RETURN
```

Now the first thing you should do is to break the rule!!! Go ahead and enter a string of more than 33 characters to see what happens. (If your computer gets snotty with you, you can always give it a lobotomy with NEW. It helps to remind it of that fact periodically.) If the program was entered properly, it is impossible to enter a string of more than 33 characters. From the above examples, you can begin to see how the LEN statement can be useful in several ways. There are many other ways that such statements can be employed to reduce programming time, clarify output and compute information. The key to understanding the usefulness of a LEN statement is to experiment with it and see how other programmers use the same statement.

## **Finding The MIDDLE\$, LEFT\$, and RIGHT\$ Parts of a String**

Suppose you want to use a single string variable to describe three different conditions, such as "Rain,Snow,Hail," but you want to use only part of that string to describe an outcome. By using MID\$, LEFT\$ and RIGHT\$, it is possible to PRINT only that part of the string you want. For

example, the following program lets you use a single string to describe three different conditions:

```
Weather.Man:
  Weather$="RainSnowHail"
  PRINT "What's the weather (R)ain (S)now (H)ail?";
  Predict$=INPUT$(1) : Predict$=UCASE$(Predict$)
```

```
Substring:
  PRINT
  PRINT "It's going to";
  IF Predict$="R" THEN PRINT LEFT$(Weather$,4)
  IF Predict$="S" THEN PRINT MID$(Weather$,5,4)
  IF Predict$="H" THEN PRINT RIGHT$(Weather$,4)
```

Realistically, you could have done the same thing with less trouble using programming techniques we already know. But no matter, it was for purposes of illustration and not to optimize program organization. Let's see what the new statements do:

Statement	Meaning
<b>MID\$(A\$,N,L)</b>	Finds the portion of A\$ beginning at Nth character L characters long
<b>LEFT\$(A\$,L)</b>	Finds the portion of A\$ L characters long starting at the LEFT side of the string
<b>RIGHT\$(A\$,L)</b>	Finds the portion of A\$ L characters long starting at the RIGHT side of the string

To give you some immediate experience with these statements, try the following:

```
Shoe$="Loafers" : PRINT LEFT$(Shoe$,4) <RETURN>
G$ = "Gypsy Rose Lee" : PRINT MID$(G$,7,4) <RETURN>
OK$= "All right!" : PRINT RIGHT$(OK$,6) <RETURN>
```

It's even possible to take a long string, break it up into parts and create an entirely new string. For example, try the following:

Riddle:

```
Whole$="When the going gets tough, the tough get weird"
Part1$=MID$(Whole$,16,3)
Part2$=RIGHT$(Whole$,5)
Part3$=LEFT$(Whole$,4)
Part4$=RIGHT$(Part3$,3)
Part5$=MID$(Whole$,19,1)
S$=SPACE$(1)
```

```
Dumb$=Part 1$ + S$ + Part2$ + S$ + Part4$ + Part5$
PRINT Dumb$
```

If you can guess what the output will be before you run the program, you have a good understanding of how substrings work!

While we're at it, why not do this next one?

WIDTH 62

Inside.Out:

```
INPUT "What's your name ";na$
CLS
FOR X=LEN(na$) TO 1 STEP -1
  PRINT MID$(na$,X,1);
NEXT X
```

Slow.Mo:

```
PRINT : PRINT
FOR X=LEN(na$) TO 1 STEP - 1
  FOR Pause=1 TO 500 : NEXT Pause
  PRINT MID$(na$,X,1);
NEXT X
```

Back.Name:

```
PRINT : PRINT
FOR X=LEN(na$) TO 1 STEP - 1
  Bname$= Bname$ + MID$(na$,X,1)
NEXT X
```

One.Line.Center:

```
PRINT TAB (31-LEN(Bname$)/2) Bname$
```

The above exercise did a couple of things for you besides giving you a chance to have a little fun. First, it demonstrated how loops and partial strings (or substrings) can be used together to format output. Second, it showed how output could be slowed down for either an interesting effect or simply to give the user time to see what's happening. Third, it showed how to build a string one character at a time in the 'Back.Name' subroutine. By rebuilding a new string it was simple to then center it using a centering routine.

## TIME AFTER TIMES

---

Try the following in the immediate mode:

```
PRINT TIME$
```

Now wait a few seconds and enter

```
PRINT TIME$ <RETURN>
```

The value of TIME\$ changed. The longer the interval between when you printed the first and second TIME\$, the bigger the difference between the times. TIME\$ is set from the Workbench file "Preferences" but you can access the various parts of TIME\$ just like any other string. The layout of TIME\$ is in hours, minutes and seconds as:

```
hh.mm.ss
```

with each variable representing a two digit substring. For example, type in the following:

```
PRINT RIGHT$(TIME$,2)
```

That returns the current number of seconds. The entire string can be broken down into hours, minutes and seconds using the following formats:

<b>Time Substrings</b>	
Hours:	LEFT\$(TIME\$,2)
Minutes:	MID\$(TIME\$,4,2)
Seconds:	RIGHT\$(TIME\$,2)

You can use time substrings to determine time in program execution. For example, the following little program shows you how to see how many times a WHILE/WEND loop is executed in a 10 second period:

```
WIDTH 62
PRINT TIME$
SEC$=RIGHT$(TIME$,2)
SEC1=VAL(SEC$)
FLAG=1
WHILE FLAG
  SEC$=RIGHT$(TIME$,2)
  SEC=VAL(SEC$)
  IF (SEC-SEC1)> 10 THEN FLAG=0
  COUNT=COUNT + 1 : PRINT COUNT;
WEND
PRINT : PRINT "Ten seconds are up"
END
```

This program generated a count of 345 in 10 seconds. You can test other aspects of your programs using the TIME\$ function. Further on in the book, we'll be seeing some very useful applications for TIME\$ in our programs. For example, we'll put together a communications program that you can use to call up commercial computer networks. Since commercial networks charge by the minute, it will be very helpful to have a built-in time display in the program that will tell us how long our computer has been on the phone.

Using the TIMER function is another way to use the built-in clock in our programs. However, using TIMER never requires a special setting of the clock or substrings. By placing the number of seconds in parentheses after TIMER, you can set your Amiga to jump to a timing routine anywhere in your program once the timer routine has been set up. Use the following sequence to set up the routine:

```
ON TIMER (N) GOSUB Subroutine.X
TIMER ON
```

If you ever need to write a program that should remind you of something, time a response or even time an external device (like a robot!), this function is handy to have. Since it can go anywhere in your program, it can be very useful. For example, this next program shows how to use TIMER to time the response to a question:

```
ON TIMER (5) GOSUB Time.check
TIMER ON
PRINT "Question: Your computer was named after someone's ";
PRINT "Spanish girlfriend?"
PRINT "Answer please (T/F)"
Flag=1
WHILE flag
  AN$=INKEY$
  IF AN$ <> "" THEN Flag=0
WEND
AN$=UCASE$(AN$)
IF AN$="F" THEN PRINT "The answer is false"
IF AN$="T" THEN PRINT "The answer is true"
END

Time.check:
  PRINT "Your time is up:"
  Flag=0
  RETURN
```

To turn off the event trapping, use `TIMER OFF`, and to suspend it temporarily, use `TIMER STOP`.

## STRING SEARCHING

---

Some programs require finding one string inside another string. (Sure it sounds weird, but there are a lot of really neat things you can do with it. Honest.) Using the `INSTR` statement along with the substring statements (`MID$` and the like), it is possible to find parts of a string and then do something useful with them. We'll start off by seeing how `INSTR` works, and then we'll do something practical with it.

To begin with, `INSTR` finds the beginning position in a string of the search string. For example, let's see where `CUP` is in `HICCUP`:

```
A$="HICCUP"
PRINT INSTR(A$,"CUP")
```

When you `RUN` the program, you get a '4' indicating that the fourth letter of the word "HICCUP" begins the word "CUP." Since it is unlikely that

you will need our example application anytime soon, let's see how it might be used in a practical program.

Suppose you have a bunch of strings that are arranged with last name first and first name last. For instance, the name 'Sam Spade' is in a string as:

```
Spade Sam
```

It is not unusual to arrange strings this way for purposes of alphabetical sorting. However, when you want to create lists or mailing labels, it just doesn't look right having people's names backwards. We'll create a little program that will fix things so that first names come first and last names last and then go through it and explain how it works.

```
Name.Flip:
CLS
INPUT "Last name and first name";NA$
SP$=SPACE$(1) : REM A SINGLE BLANK SPACE
L=INSTR(NA$,SP$)
NF$=MID$(NA$,L+1)
NL$=LEFT$(NA$,L-1)
PRINT NF$;SP$;NL$
```

When you RUN the program, be sure to put a space between the last and first name when prompted. Stepping through the program, we find:

- Step 1.** We will look for the space between the last and first names with SP\$ which has been defined as a blank space.
- Step 2.** Using INSTR, we store the starting position of the space in the variable L.
- Step 3.** The first name is everything to the right of the space; by using MID\$, we define NF\$ as everything from the right of the space to the end of the string. Remember, if we do not put a second parameter value in for MID\$, it defaults to everything from the first parameter (starting position) to the end of the string.
- Step 4.** Conversely, everything to the left of the space is the last name, so we load that into the string variable, NL\$ using LEFT\$.
- Step 5.** All we have to do now is to rearrange things in the order we want and PRINT them out.

### Amiga Notes

#### A Very Short Sermon

By breaking down a problem into simple little tasks, it is a lot easier to write programs.

## CONVERTING BETWEEN STRING AND NUMERIC VARIABLES

---

**STRINGS TO NUMBERS.** Now we're going to learn more about changing strings to numbers and numbers to strings. In Chapter 2, you were introduced to VAL and STR\$. Now that you understand substrings, these functions will mean more to you. To get started, let's RUN the following program:

```
Score:
FOR X=1 TO 10
  READ Na$(X)
  Score$(X)=RIGHT$(Na$(X),2)
  Score(X) =VAL(Score$(X))
  SUM=SUM+Score(X)
NEXT
Calculate:
Mean=SUM/10
PRINT "The group's average is"; Mean

DATA Smith 76,Jones 85, Butler 97,Roosevelt 94
DATA Wong 83,Kimberly 71,Allen 79,Kim 63
DATA Jackson 90,Kelly 67
```

Using DATA that were originally in a string format, we were able to change a portion of the string array to a numeric array. By making such a conversion, we were able to use our mathematical operations to determine the mean score for the group. In the 'Score' routine, we used the following steps:



1. Read the DATA into Na\$ array.
2. Put the numeric substring into Score\$ array.
3. Placed the Score\$ string array into Score numeric array.
4. Tallied the Sum in the SUM variable.

All that was left was to divide the SUM by 10 to determine the average score.

Enter these from the immediate mode to get the feel of how they work.

```
Tede$ = "222" : PRINT VAL(Tede$) + 22 <RETURN>
Enough$ = "1025" : PRINT VAL(Enough$) * 10 <RETURN>
Deal$ = "44.95" : PRINT "Deal for you! ->$"; VAL(Deal$) / 2
<RETURN>
```

**NUMBERS TO STRINGS.** All right, now let's go the other way. We saw why we might want to change strings to numbers, but we may also want to change numbers to strings. To make the conversion we use the STR\$ statement. For example, look at the following program:

```
Zipper.Fixer:
CLS
INPUT "What's your Zip Code"; ZIP
ZIP$ = STR$(ZIP)
PRINT : REM Just put in a vertical space
ZIP$ = RIGHT$(ZIP$,5)
PARCEL$ = LEFT$(ZIP$,3)
PRINT ZIP$,PARCEL$
```

This program did two things. First of all, it took the invisible space out of the zip code number by using the right five characters of the string. That's important since numbers, you'll remember, have that blank space in front of them. Also, since some shipping companies use the first three digits of a Zip Code to determine shipping costs, PARCEL\$ shows how to strip off the first three digits of a Zip Code. Now, let's get some practice in the immediate mode:

```
B = 22.00 : B$ = STR$(B) : PRINT B$ <RETURN>
V = 2345 : V$ = STR$(V) : PRINT V$ <RETURN>
Money = 22.36 : Money$ = STR$(Money) : PRINT LEFT$(Money$,2)
<RETURN>
```

## Combining Strings with Concatenation

We have seen how we can take a portion of a string and PRINT it on the screen. Now, we will tie strings together. This is called CONCATENATION and is accomplished by using the “+” sign with strings. For example:

```
CLS
INPUT "Your City -> ";City$
INPUT "Your State -> "; State$
Send$ = City$ + "," + State$
PRINT Send$
```

By adding the comma (,) as a string between City\$ and State\$, you can put both into a single string, Send\$. When this is printed, you'll get something like:

```
San Diego, CA
```

Thus, you can see how to concatenate both strings in variables and strings directly. To see something a little more practical and a nifty trick to boot, try the following program:

```
String.Line:
  WIDTH 62
  FOR X=1 TO 62
    LINE$=LINE$ + "-"
  NEXT X
  PRINT : PRINT
  PRINT LINE$
```

Until you get to the graphics chapter, you have something with which to make lines.

## Padding and Parsing

Two important functions in string manipulation are “padding” and “parsing.” Sometimes when creating files and formatting output, you will want to have everything a certain size. For example, suppose you wanted every string to be exactly 25 characters long. Those strings that are too long, you want to cut or “parse,” and those that are too short, you want to

“pad.” By inserting spaces in front of or at the end of a string, it is possible to make it line up any way you want. For example, the following program shows how to “right justify” strings of any length using padding and parsing:

Right.Just:

```
FOR X=1 TO 5
  INPUT "Write something"; A$(X)
  IF LEN (A$(X)) > 25 THEN GOSUB Parse
  IF LEN (A$(X)) < 25 THEN GOSUB Pad
NEXT

FOR X= 1 TO 5
  PRINT A$(X)
NEXT

END
```

Parse:

```
A$(X) = LEFT$(A$(X),25)
RETURN
```

Pad:

```
WHILE Flag <> 1
  A$(X) = SPACE$(1) + A$(X)
  IF LEN (A$(X))= 25 THEN Flag= 1
WEND
Flag=0
RETURN
```

We introduced a new word, `SPACE$(N)`, which makes a space `N` characters long. For example, `SPACE$(5)` would create five spaces or, if defined as part of a string, it would have a length of five.

## SUMMARY

---

By this point, you should begin to see why substrings are important. Your computer can do all kinds of string manipulation tricks, and you need to take advantage of them when you can. The more you program,

the more you should let your Amiga do the work. Make it *smart* by using the powerful programming statements you've learned, and then once it's really smart, it can do all of the hard work.

Most of the work you will do with strings and substrings is to set up interesting and clear output. There will be many calculations used to set up the output, and while it is important to understand the use of substrings for output, it is also important to remember that substrings are subject to calculations as well so that they can be formatted for correct output. This again reminds us of the importance of having clear, structured programs. The next chapter will provide more tricks and techniques to format output that may be used with the substring functions you learned in this chapter.

# Preparing Data and Formatting Output

---

## FORMATTING AND MANIPULATING INFORMATION

---

Your Amiga has several statements that will clearly arrange your output for clear communication. This chapter concentrates on a number of these statements and shows you how to use them to arrange things on your screen for interesting and lucid presentations. We will write a fairly large program that uses these new tricks to illustrate their functions.

Preparing information for output requires two types of data manipulation:

1. Numeric: manipulating numeric data with mathematical operations
2. String: manipulating strings with concatenation and substring statements

Numeric and string data manipulations often go hand in hand. To use any kind of mathematical calculations, strings must be transformed into numeric properties using such functions as `LEN` and `VAL`. To put the

calculations into a useful format, the information then has to be transmitted into strings. Using values and strings, the final step is to put everything on the screen, placing things so that they make sense. This involves planning the position of things on the screen in relationship to each other.

## SCREEN PLACEMENT

We've used the comma, semicolon and even the PRINT statement to format text on the screen. We also used TAB in an example of centering text and how to use SPACE\$. Here, we're going to take a close look at these and other keywords used to place text on your screen. First, let's take a look at some statements and functions:

Screen Placement Words	Function
SPACE\$(N)	Creates a string with N number of spaces
TAB (N)	Used within PRINT statement to place next character N spaces from left margin
SPC (N)	Used within PRINT statement to create specified number of spaces. (SPC starts printing non-space one space after N.)
LOCATE vertical, horizontal	Used to place cursor (and next screen output) at specified row and column on the screen

First, let's see how SPACE\$, TAB and SPC are used within print statements. They do not have to be separated from the text with commas, colons or semicolons:

```
Spacer:
WIDTH 62
CLS
PRINT SPACE$(10)"Space String"
PRINT TAB (10)"TAB is here"
PRINT SPC(10)"SPC is here"
LOCATE 1,1
PRINT "Top!"
LOCATE 20,50
PRINT "Down and Out";
Hold$=INPUT$(1) : REM Keeps it from scrolling
```

The last line prevents the output from scrolling off the top of the screen. (Pressing any key will cause the program to end.)

That little program did a lot. First, notice how SPACE\$ and SPC placed the text in the same column while TAB put it one space to the left. Both SPACE\$ and SPC planted 10 spaces and then the text. The TAB function placed the text right on column 10. Also note how the word 'Top' was put on the same line as 'Space String' without erasing the screen. The nice thing about LOCATE is its ability to "plant" a character right where you want it without scrolling the screen. However, if you put a character or string too close to the right edge of the screen, with LOCATE it will scroll to the next line. Change the horizontal value in the second LOCATE statement from 50 to 60 and see what happens.

Now let's have some fun with our statements. Here's a little program that will give you an idea of how to place text within your program. In addition, it allows you set your WIDTH:

```
Madison.Ave:
  INPUT "Screen width ";SW
  WIDTH SW
  CLS : PRINT : PRINT
  PRINT : PRINT : INPUT "Your Ad Here "; Commercial$
  PRINT : INPUT "Column"; X
  PRINT : INPUT "Row"; Y
  REM *****
  REM Put your Ad on the Screen
  REM *****
Billboard:
  CLS
  LOCATE Y,X : PRINT Commercial$;
  PRINT : PRINT : PRINT "Press";
  WRITE "M"
  PRINT "for more"
  AN$ = INPUT$(1) : AN$=UCASE$(AN$)
  IF AN$ = "M" THEN Madison.Ave ELSE END
```

As you can see, variables can be used with formatting statements. Thus, LOCATE Y,X is read in the same way as if we had used numbers. Using the above program, what do you think would happen if you entered "This is a very long and boring commercial," in column 50 with a screen width of 62? Since the maximum horizontal position is 62 before text starts disappearing off the screen the string will run out of horizontal room. Go ahead and see what happens, and when you use

these statements in your programs, you will have a better understanding of their parameters. Once you see how far you can push the horizontal, see what the vertical limits are.

## **PRINT USING: FORMATTING WITH STYLE**

---

You may have noticed by now that if you print a number on the screen with a trailing zero after a decimal point, the zero gets dropped. Likewise, if you try to put a dollar sign in front of a numeric variable, there is a space between it and the number. For example, enter the following:

```
PRINT "$";9.60
```

You want to get \$9.60, but instead you get

```
$ 9.6
```

The PRINT USING statement solves these problems by allowing you to format the output. It's a strange term, but once you start using it, you won't care what it's called. Type in the following:

```
PRINT USING "$$.##";9.6
```

The pound (#) signs refer to the number of digits to be printed out minus one. (Three pound signs make room for four digits.) The double dollar signs align the output dollar sign adjacent to the first digit, and the period between the dollar and pound signs shows the position of the decimal point relative to the rest of the number. Try the following program to see what happens when there are more digits than spaces for them:

```
Too.Much:  
CLS  
FOR X = 1 TO 200 STEP 45  
PRINT USING "$$#";X  
NEXT X
```



The results show:

```
$1
$46
$91
%$136
%$181
```

Now, change the PRINT USING line to:

```
PRINT USING "$$##";X
```

This time you got:

```
$1
$46
$91
$136
$181
```

If you add more pound (#) signs, you will just get more spaces to the left of the dollar sign. Notice how all of the numbers are right justified also. Now change the line by adding about five more pound (#) signs to see what happens.

The dropped zeros problem was solved by adding the two pound signs after the decimal point in the PRINT USING line. All you have to do is to include a decimal point (.) among the pound signs (#) where you want your decimal points. Watch this when you run it:

```
Save.Zero:
CLS
FOR X = 10 TO 250 STEP 20
  PRINT USING "##.##";X/5
NEXT X
A$ = INPUT$(1)
```

You got all your trailing zeros, and everything was lined up nicely.

Try using a single dollar sign to see what happens:

```
One.Buck:
  FOR X=1 TO 130 STEP 30.5
  PRINT USING "$###.##"; X
  NEXT
```

Notice how with a single dollar sign, all of the dollar signs line up evenly and all of the numbers are right justified. That looks a little neater in some applications, so depending on whether you want your dollar signs lined up evenly or you want them directly next to the numbers, use one or two dollar signs in your PRINT USING statements.

Now that you have an idea how to use PRINT USING with dollars and cents, let's take a look at what else you can do with this statement. Enter the examples below to get used to each format:

<b>PRINT USING Formats</b>	<b>Example</b>
# One digit position for each pound sign (#).	PRINT USING "###"; 689
##.## Places decimal point in position relative to pound signs and number of pound signs.	PRINT USING "##.###"; 24.67
\$ Places left justified dollar sign next to number.	PRINT USING "\$#.##"; 8.90
\$\$ Places dollar sign adjacent to number.	PRINT USING "\$\$###.##"; 23.45
! Prints only the first character of a string.	PRINT USING "!"; "Amiga"
& Prints the whole string when string length is variable.	PRINT USING "&"; Some\$
\ \ Prints the number of characters plus two based on the number of spaces between the back slashes. (NOTE: The back slash is adjacent to the BACK SPACE key.)	PRINT USING "\ \"; "COMPUTER" PRINT USING "\ \ \"; "Amiga"; "Computer"
, Placed at the last position or last position before a decimal point, places commas every third position in numbers.	PRINT USING "#####"; 1234567891 PRINT USING "#####.##"; 5555.55
# Placed before character, that character is placed as literal (i.e., any character you want in the indicated position).	PRINT USING "#%"; 8 PRINT USING "#( "; 66
+ Outputs a plus or minus sign before positive or negative numbers.	PRINT USING "+#####.##"; 22.33;-87
- Outputs a trailing minus sign after negative numbers.	PRINT USING "#####.##-"; 102; -42
^^^ Placed at the end of a PRINT USING format, it results in an exponential output	PRINT USING "#####^^^"; 525

## Strings in PRINT USING

Another way to get the PRINT USING formats set up is to put them into string variables. For example, when formatting dollars and cents or percentages, you might want to do the following:

```
Tax.Calc:
  Buck$= "$#####.##"
  SalesTax$= "##.##%"
  INPUT "Amount of purchase";Amount
  INPUT "Sales tax (Enter decimal values) "; Tax
  Total=Amount + (Amount * Tax)
  Tax=Tax * 100
Out.Put:
  PRINT "Your purchase is "; : PRINT USING Buck$; Amount
  PRINT "Your sales tax is "; : PRINT USING SalesTax$;Tax
  PRINT "It will cost you "; : PRINT USING Buck$;Total
```

Pay close attention to the last three lines in the “Out.Put” module showing how to use PRINT USING in combination with PRINT. In big programs where various formats are used, you can define your PRINT USING formats at the beginning of your program, making it a lot easier to use the different outputs where required. Also, be sure to use a variable name for your string that will be easily remembered.

## HIGHLIGHTING OUTPUT WITH COLOR

In Chapter 12, we’ll really get into graphics and color, but there is a graphic statement you need now. It gives you the ability to create inverse output which is important in formatting output. That statement, COLOR, can be used to ‘toggle’ white characters on a blue background with blue characters on a white background. For example, run this program:

```
Toggle:
  PRINT "Regular"
  COLOR 0,1 : REM White background
  PRINT "Inverse Video"
  COLOR 1,0 : REM Back to normal
  PRINT "Back to normal"
```

The first value in color is the foreground color and the second is the background color. There are several more color combinations we could use, but that will have to wait until Chapter 12. In the meantime, try this next program that demonstrates using variables with COLOR:

Flipper:

```
Message$= "This important message"
FOR x=1 TO 9
  Foreground=0
  Background=1
  COLOR Foreground,Background
  PRINT Message$
  Foreground=1
  Background=0
  COLOR Foreground,Background
  PRINT Message$
NEXT
COLOR 1,0
```

You can also get a flashing effect by toggling normal and reverse video using LOCATE. Look at this next program to see how:

Good.Eats:

```
WIDTH 62
Joe$=" Eat at Joe's"
L=LEN(Joe$)
Center=31 - L/2
```

Flasher:

```
WHILE Halt$ = "" : REM Loop until a key is pressed.
  Halt$=INKEY$
  COLOR 0,1
  LOCATE 10,Center
  PRINT Joe$
  COLOR 1,0
  LOCATE 10,Center
  PRINT Joe$
WEND

COLOR 1,0
REM Remember to return to normal video.
```

While you're at it, see how the centering routine was used with LOCATE and how it was set up to work until you pressed any key.

## HOW ABOUT A DATES

The last screen presentation word we'll introduce in this chapter is DATE\$. It works something like TIME\$, but instead of the time you get the date set from the Workbench 'Preferences.' To see what date your Amiga is set to, key in the following:

```
PRINT DATE$
```

Your output will look something like:

```
06-07-1988
```

depending on the setting of your built-in calendar. (You can get clocks with batteries to attach to your Amiga so that you do not have to update your clock and calendar information.)

Since DATE\$ is set up as a string, you can take the substrings and use it to provide a descriptive date. By replacing the values in the calendar (01-12) with the names of the months, January-December, you can make a text calendar from your numeric calendar. The following is one example:

```
Date.Maker:
```

```
  DIM Month$(12)
  FOR X=1 TO 12
    READ Month$(X)
  NEXT X
```

```
Make.My.Date:
```

```
  D$=DATE$
  CMonth$=LEFT$(D$,2)
  Day$=MID$(D$,4,2)
  Day$=Day$ + ", "
  Year$=RIGHT$(D$,4)
  YEAR=VAL(Year$)
  CMonth = VAL(CMonth$)
```

```
PRINT "Today is ";Month$(CMonth);SPC(1);Day$;Year
```

```
DATA January,February,March
DATA April,May,June
DATA July,August,September
DATA October,November,December
```

Now, let's take all the tricks you've learned in this chapter and make one big program. We'll create an interesting menu for you to use in your programs.

Snazy.Screen:

```
COLOR 0,1 : REM reverse colors
CLS : REM Turn screen white
WIDTH 62
COLOR 1,0
BAR$=SPACE$(58)
FOR X=9 TO 2 STEP-1
  LOCATE X,2
  PRINT Bar$
  LOCATE 19-X,2
  PRINT Bar$
NEXT
```

Calendar:

```
Today$="Today is " + DATE$
Halfway = 31 - LEN(Today$)/2
LOCATE 19,Halfway
PRINT Today$;
```

Menu.A.la.Carte:

```
COLOR 0,1
FOR X=1 TO 7
  LOCATE X*2+1,5
  PRINT USING "#";X
NEXT
COLOR 1,0
FOR X=1 TO 7
  READ Choice$
  LOCATE X*2+1,7
  PRINT Choice$
```

```
NEXT
LOCATE X*2+1,6
PRINT "Choose a number";
That.One$=INPUT$(1)
```

Your.Choice:

```
DATA Whatever,You,Want,In,Your,Menu,Here
```

You can put anything you want in the DATA statements, depending on what your program does. You can also make more or fewer choices available by changing the size of the loop.

## SUMMARY

---

Formatting programs makes the difference between a useful and not-so-useful application of your computer. The more organized and clearer your program is, the better the chances are for simple yet effective programming. Formatting is more than an exercise in making your input/output fancy or interesting. It is a matter of communication between your Amiga and you! After all, if you can't make heads or tails out of what has been computed, the best calculations in the world are of absolutely no use.

Formatting statements, such as TAB, SPC, SPACE\$ and PRINT USING give you a great deal of control over how things appear on the screen. These are handy tools for organizing the various parts in a manner which gives you complete control over your computer's output. What may at first seem like a petty, even silly command in Amiga BASIC can be appreciated as an excellent tool upon useful application. The more you apply these new tools, the better you will become at using them and discover new applications for them.





---

## CHAPTER 10

# Of Mice and Menus

---

### THE MOUSE INPUT

So far the main input device we've used is the keyboard. The interactive programs we've examined have been based on some response triggered by pressing one or more keys. However, besides having everything the user wants to enter from the keyboard, the 'mouse' is an important tool as well. You've probably used the mouse and pull-down menus a lot with the Workbench disk or with BASIC-like saving programs, and if you have some commercial programs, chances are they took advantage of the mouse as well. This chapter will show you how to use the mouse, pointers and pull down menus for interactive input with programs you write. This chapter will concentrate on using the mouse with the menu option, and in later chapters, primarily those dealing with graphics, we'll see more the mouse can do on its own.

---

### PULL DOWN MENUS

To get started, press the right button on your mouse and look at the different menus from BASIC. The Project, Edit, Run and Windows menus all represent "main choices" available to you. We'll refer to the main choices simply as "menus." By holding down the right mouse button and

placing the pointer over one of the menus, you can see the “menu options” or simply “options.” We’ll see how to use these menus and options for getting things done from our own programs. To get started, we’ll review the various keywords in Amiga BASIC that use the pull down menus and then get going on our own menus.

**MENU N,O,S,N\$.** The keyword MENU has four parameters: (N)umber from 1–10; (O)ption from 0–19; (S)tate from 0–2; and an optional string identification. Arranged horizontally from left to right each menu is identified by this first number. Thus, if the first number is 2, then it will be the second menu from the left. The second number is the option once the menu is pulled down. The identifying menu (the one on the horizontal bar) has an option value of zero. The other ones are arranged vertically with the lowest number in the highest vertical position. The third parameter indicates whether the menu is off (0), on (1) or on and checked (2). Finally, there is an optional string that identifies the menu.

**ON MENU.** ON MENU works like ON M GOSUB. The variable M is read from MENU(0) or MENU(1) *only* or a variable that has been defined as either. The main menus are based on the value of MENU(0) and the option choices are based on MENU(1). The first branch is to the main menu subroutine and the second branch is to the option within the menu.

**MENU RESET.** This command resets all menus to default conditions of BASIC. This should always be executed in BASIC programs using pull down menus before the program stops. It can be initiated from the immediate mode if the program bombs and you need to reset the menus.

**MENU ON/OFF/STOP.** These options are used to enable, suspend or halt menu options.

## SETTING UP YOUR MENUS

---

The first thing to do is to decide what your main menus will be and what options you want them to have. The following program is a simple menu with four main menus with different kinds of options. Let’s look at it and then see why it works the way it does:

Menu.Define:

REM First, set up the menus and options.

```
MENU 1,0,1, "Group 1"  
MENU 1,1,1, "Choice 1"  
MENU 1,2,1, "Choice 2"
```

```
MENU 2,0,1, "Group 2"  
MENU 2,1,1, "Choice 1"  
MENU 2,2,0, "Choice 2"
```

```
MENU 3,0,1, "Exit"  
MENU 3,1,2, " Get me outta here"
```

```
MENU 4,0,1, " " : REM Turn off Window 4
```

CLS

PRINT "Press the RIGHT mouse button and choose a menu."

Menu.Read:

```
REM Loop to scan menu choice.  
CHOOZEIT=MENU(0)  
ON CHOOZEIT GOSUB GROUP1,GROUP2,TERMINATE  
GOTO Menu.Read
```

GROUP 1:

```
ON MENU(1) GOSUB g1,g2  
RETURN
```

g1:

```
PRINT "Group1/Choice1"  
RETURN
```

g2:

```
PRINT "Group1/Choice2"  
RETURN
```

GROUP2:

```
ON MENU(1) GOSUB g3,g4  
RETURN
```

```

g3:
  PRINT "Group2/Choice1"
  RETURN

g4:
  PRINT "Group2/Choice2"
  RETURN

TERMINATE:
  PRINT "You have ended the program with the mouse!"
  REM Be sure to get the menus back to normal.

MENU RESET
END

```

The program first defines four menus. Menu 1 is defined as “Group 1,” Menu 2 as “Group 2,” Menu 3 as “Exit” and Menu 4 as a blank. Run the program and look at the Menu Bar by pressing the right mouse button. You just see:

```
Group 1  Group 2  Exit
```

You saw no fourth menu since it was used as a “cover” for the “Windows” menu in the BASIC default Menu Bar. (Just for fun, change the name of the fourth menu to see it on the Menu Bar.)

Now pull down each of the menus. With them all down at once, they'd look like this:

```
Group 1   Group 2   Exit
Choice 1  Choice 2   ✓ Get me outta here
Choice 2  (Choice2)
```

In the Group 1 menu, everything is active, and if you place the pointer over any of the options and release the right mouse key, the program will branch to the appropriate subroutine. However, in Group 2, the second choice is “ghosted,” meaning it is not currently active. If you click it, nothing happens. Notice in the program listing the option is stated as:

```
MENU 2,2,0, "Choice 2"
```

The third parameter is zero, making it inactive. You may wonder why bother with a menu option that you cannot use. The reason is that

instead of a number you may have a variable in that parameter position which can be changed depending on what the program is doing. We'll examine that later.

Finally, if you look at the single Exit menu option, you will see a check mark next to it. That check was created by placing a 2 in the third parameter position. The listing shows:

```
MENU 3,0,1, "EXIT"  
    MENU 3,1,2 " Get me outta here"
```

The main menu can only have a 1 or 0 in the third parameter position, but the option menu can have a 2 as well. When you use the 2, it creates a check mark next to the option. Be sure to include two spaces in front of the string name as was done in the example. Otherwise, the string and check will crash into one another.

## TOGGLING MENUS

---

The basic menu above sets the menus in a permanent configuration. However, you can toggle menu labels, check marks and even "ghost" images. First type in and run this next program, and then we'll see how to use some of the features it has.

```
Header:  
Vflag$="N"  
Cflag$="A"  
Gflag$="Ghost On"  
GOSUB Menu.Set  
WIDTH 62  
ON MENU GOSUB Get.Menu  
MENU ON  
Writer:  
CLS  
PRINT "Start typing"  
SubFlag=0  
WHILE 1  
    IF SubFlag THEN Writer  
        Type$=INKEY$  
        PRINT Type$;  
WEND
```

Get.Menu:

```
Mainmenu=MENU(0)
ON Mainmenu GOSUB Video,Calculate,Finish
RETURN
```

Video:

```
OptionSet 1=MENU(1)
ON OptionSet1 GOSUB Reverse,Normal,Ghost
RETURN
```

Reverse:

```
COLOR 0,1
CLS
Vflag$="R"
GOSUB Menu.Set
SubFlag=1
RETURN
```

Normal:

```
COLOR 1,0
CLS
Vflag$="N"
GOSUB Menu.Set
SubFlag=1
RETURN
```

Ghost:

```
IF Gflag$="Ghost On" THEN GOSUB Ghostoff : RETURN
IF Gflag$="Ghost Off" THEN GOSUB Ghoston : RETURN
```

Ghostoff:

```
Gflag=1
Gflag$="Ghost Off"
GOSUB Menu.Set
RETURN
```

Ghoston:

```
Gflag=0
Gflag$="Ghost On"
GOSUB Menu.Set
RETURN
```

```
Calculate:
  IF Cflag$="A" THEN GOSUB Add ELSE GOSUB Subtract
  RETURN

Add:
  CLS:
  PRINT "Adding Machine: Enter 0 to exit"
  N=-1 : Sum=0
  WHILE N
    INPUT "Number ";N
    Sum=Sum + N
    PRINT "Running total is"; Sum
  WEND
  Cflag$="S"
  GOSUB Menu.Set
  SubFlag=1
  RETURN

Subtract:
  CLS
  PRINT "Subtracting Machine: Enter 0 to exit."
  N=-1 : Total=0
  INPUT "Beginning value ";Total
  WHILE N
    INPUT "Number to subtract ";N
    Total= Total- N
    PRINT "Running balance is";Total
  WEND
  Cflag$="A"
  GOSUB Menu.Set
  SubFlag=1
  RETURN

Finish:
  MENU RESET
  CLS
  END
Menu.Set:

  MENU 1,0,1, "Screen"
  IF Vflag$="R" THEN Rflag=2 ELSE Rflag=1
```

```
MENU 1,1,Rflag, " Reverse"
IF Vflag$="N" THEN Nflag=2 ELSE Nflag=1
MENU 1,2,Nflag, " Normal"
MENU 1,3,1, Gflag$

MENU 2,0,1, "Calculate"
IF Cflag$="A" THEN M$="Add"
IF Cflag$="S" THEN M$="Subtract"
MENU 2,1,1, M$

MENU 3,0,1, "Exit"
MENU 3,1,1, "Quit this sucker"

MENU 4,0,Gflag, "Ghost"
MENU 4,1,Gflag, "Booo"

RETURN
```

This menu program shows the flexibility of menus in programs. The 'Writer' routine represents any main program you may be using. In this example it just loops endlessly, printing everything you type on the screen. The main feature is that while in the 'Writer' routine, the program will branch to the menu subroutines when the various menus and options are clicked. Let's look at each option separately.

**CHECKED OPTIONS.** The options that are checked in the 'Screen' menu simply change the third parameter in the MENU statement in the 'Menu.Set' subroutine. The 'Rflag' and 'Nflag' variables are changed between 1 and 2 depending on whether 'Vflag\$' has been set to 'R' or 'N'. The 'Vflag\$' variable is set in the 'Video' subroutine initiated by the click of the mouse in the first menu. Notice that the third option in the 'Screen' menu is not checked. That's because its third parameter is a constant value of 1. This shows that the check option in setting up your menus need not involve all of the options in a single main menu.

**STRING TOGGLE.** Changing the option name is a "string toggle" since all you do is to change the string in the fourth parameter of the MENU. This is done in the first and second main menus, 'Screen' and 'Calculate.' In the Screen menu, 'Ghost On' and 'Ghost Off' is toggled in the variable Gflag\$ and the option 'Add' and 'Subtract' in the 'Calculate' menu with Cflag\$.



---

**GHOST TOGGLE.** Besides having the third MENU variable being a 1 or 2, it can also be a 0 (zero), making it inoperative. The 'Ghost' toggle itself is in the Screen menu, but the 'Ghost' menu and 'Booo' option are what get "ghosted." When the third parameter value is zero, the ghost is "on" and dims the menu or option. Also, it turns off the option choice so the program will not branch to a "ghosted" option.

## REFRESHING THE MENU

---

Each time a variable value is changed, the program must branch to the 'Menu.Set' routine to update the menu options. This is important to remember since the only way for the new values to be reflected in the menu items is for the numeric and string variables to be updated with the new information in the MENU statement. It is not enough simply to change the variable value and not run it through MENU.

Another useful tip along the same lines is illustrated in the 'Writer' routine. The SubFlag variable is toggled between the values of 0 and 1. This variable serves to flag a jump to either of the 'Calculate' subroutines. A 'SubFlag' value of 1 makes the program drop out of the WHILE/WEND loop and then to the beginning of the 'Writer' routine thereby clearing the screen and resetting the value of SubFlag to 0. If there were no such flag, the screen would stay cluttered with the last calculations from the 'Add' or 'Subtract' option. By changing the value of 'SubFlag' in the subroutines, you have a way of letting the program know that the subroutine jump was made.

## SUMMARY

---

This was a short and focused chapter, designed to tie together a lot of programming skills. First, we saw how the mouse can be used as another input device for branching programs to different options. The powerful MENU statement is used to create user-controlled options while a program is running. Equally important, you should begin to see how crucial structure, especially module structure, is when creating programs to keep everything simple yet integrated. Both program examples were fairly long, yet they did very simple things. You can imagine the number of things you must keep track of with more complex tasks.

Finally, we saw how to use “flags” to report changing states within the program. Both string and numeric variables update and refresh key parts of your program. Think of them as messengers running around the different modules making reports. The flags did everything from reporting a jump to a subroutine, to changing the state of a main menu or menu option. As we progress, whether or not programs use menu options, we will be using flag variables more and more. That’s because we’re going to be creating larger and more powerful programs, and the flags will report what the different elements are doing.

# Screen Control

---

## SCROLL MANAGEMENT

---

A major problem in output is keeping things on the screen when you need them and getting them off when you do not need them. For instance, suppose you need the first several values generated in this next program:

```
CLS
FOR NUM = 1 TO 50
  VALUE=RND(100) * 100
  PRINT VALUE
NEXT NUM
```

Instead of numbers, suppose you have a list of names you sorted or some other output you want to see before they zip off the top of the screen. Depending on the desired output, screen format and so forth, there are several different ways to control the scroll. We'll start with a simple method.

```
Scroll.Hold:
CLS
FOR num = 1 TO 80
  IF Count=19 THEN GOSUB Hold.It
```

```

    Count=Count+1
    PRINT num
NEXT num
GOSUB Hold.It
END

```

```

Hold.It:
    COLOR 0,1
    PRINT "Press any key ";
    Hold$=INPUT$(1)
    COLOR 1,0
    CLS : REM Optional to clear old screen
    Count=0 : REM Don't forget to reset counter!
    RETURN

```

By keeping an eye on where the cursor will be next, you can stop output until you have a chance to see what's on the screen.

## **FINDING THE CURSOR WITH POS(0) AND CSRLIN**

---

While we are on the topic of locating the cursor position, let's take a look at POS(0) and CSRLIN. The POS(0) function locates the horizontal cursor position and CSRLIN the vertical. POS(0) allows you to control side to side scrolling. For example, the following program enters a PRINT statement to give a line feed when a certain horizontal position is exceeded:

```

CLS
FOR X=1 TO 42
    Horizontal=POS(0)
    IF Horizontal>30 THEN PRINT
PRINT "Block";
NEXT X

```

Run the program, and all the "Blocks" are arranged in a block. Delete the line defining Horizontal as POS(0) in the program and try it again. The second time, the arrangement is not in blocks. In larger programs, you do not want to have to determine where to locate the cursor every time there is a new screen output. You can store the values of POS(0) in variables,

and then use those variables to move the cursor back to the desired position. The key to understanding how to use these two variables is to experiment with them in formatting output.

You can find the vertical position of the cursor just as easily with CSRLIN. Run this program to see how CSRLIN keeps track of the Y (vertical) position of your cursor:

```
CLS
FOR X = 1 TO 8
  PRINT : REM Add extra line.
  PRINT X;
  PRINT "Cursor at";CSRLIN
NEXT X
```

Note that the value of the X and the position are *not* the same. That's because we added an extra PRINT statement to show that no matter what kind of spacing you use in your program, the vertical cursor position will be tracked by CSRLIN.

If you use CSRLIN and POS(0) together, you can track the X,Y position of your cursor anywhere on the screen. In programs where you wish to pause, you will find it helpful to issue a line feed, tab spaces or other cursor related routines. The following little "word processor" keeps track of where the cursor is while you write text:

```
Cheap.WP:
PRINT : PRINT
WHILE Writo$ <> "|" : REM Vertical bar to exit.
  Writo$=INPUT$(1)
  PRINT Writo$;
  Y=CSRLIN
  X=POS(0)
  LOCATE 1,1
  PRINT Y;X
  LOCATE Y,X
WEND
```

Each time you press the RETURN key, the Y (vertical value) increases and each time a new character is printed to the right, the X (horizontal) value increases. The running total is in the upper left hand corner, and the cursor knows where to print the next letter since the program uses the X and Y values in the second LOCATE statement.

In addition to controlling output by holding the scroll, you can also control it by using sequential columns. To use more of the screen, you could have the output tabbed to another column after the vertical screen is filled. For example:

```
Two.Columns:
  CLS
  FOR X=1 TO 36
    IF X>18 THEN GOSUB Column.Jump
    PRINT X
  NEXT X
  END
REM *****
REM New Column
REM *****
Column.Jump:
  IF X=19 THEN LOCATE 1,1
  PRINT TAB(20);
  RETURN
```

Another trick is to make “calculated columns” that come up simultaneously. For example, the following program lines up output in three equal columns. If the number is not equally divisible by three, then it tags on the extra values in the last column. Notice how we used the MOD (modulo = division remainder) function to determine whether or not the number of columns would be even. See if you can change the program to line up the numbers evenly with the extra values being placed in the first two columns.

```
Tailer:
  CLS
  INPUT "Enter any number";N
  Y=INT(N/3)
  Jump= N MOD 3
  FOR X=1 TO Y
    PRINT X,X+Y,X+(2*Y)
  NEXT
  IF Jump THEN GOSUB Tail
  END
```

```
Tail:
  FOR K=X TO X+(Jump-1)
    PRINT ,,K+(2*Y)
  NEXT K
  RETURN
```

You get the idea. Format your output in a manner that best uses your size screen and your needs and get that scroll under control!

## YES—THE AMIGA DOES DO WINDOWS

---

Another powerful Amiga formatting statement is WINDOW. This statement allows you to partition your screen into separate little screens or 'windows.' You may want your input in one window and output in another, or you may want different colors for different windows. The window statement can do a lot to make your programs look professional. To get started, we'll partition your screen into two little screens. For each window include:

1. The window number from 1 (default window) to the total number of windows you use
2. Title of window
3. The window size in terms of a rectangle specified by the opposite corners (in parentheses) as the X (horizontal)/Y (vertical) positions
4. Type of window based on *adding* the following values:

Value	Characteristic
1	Can <i>change</i> window size
2	Can <i>move</i> window across screen
4	Can be moved <i>front</i> and <i>back</i>
8	Can <i>close</i> window
16	Reappears after being covered by another window

```
Set.Windows:
WINDOW 1, "alpha",(1,1)-(600,80),15
WINDOW 2, "beta",(1,90)-(100,160),16
```

```
Present.Windows:
WIDTH 62
WINDOW OUTPUT 1
PRINT "This is Window #1"

WINDOW OUTPUT 2
WIDTH 9 : REM Remove this and see what happens
PRINT "And this is #2"

WIDTH 62 : REM RESET TO WINDOW #1 size
```

The two windows, labeled 'alpha' and 'beta' have different characteristics and sizes. The alpha window fills most of the horizontal screen since it was drawn from pixel position 1 to position 600. A pixel is a dot of light approximately 1/10 the horizontal size of a character. That is, for every screen character, there's room for 10 pixels. You can get a few more pixels in a window if you do not choose to have the screen change in size or be removed. We'll discuss pixels further on in this chapter when we discuss the SCREEN statement.

The alpha window is type 15. The value 15 was generated by adding  $1 + 2 + 4 + 8$ , giving the window all but the last (value 16) characteristic. The beta window is *only* type 16. It cannot be moved, change its size or move behind another window. It stays put and visible until you quit the program and drop the LIST window over it. If you want all of the characteristics available in a single window, just use 31 as the type. (In case you didn't notice, 31 is the sum of *all* of the values.)

To see some window characteristics, run the program and place the pointer in the alpha window and click the mouse. You'll see the vertical cursor appear in the alpha window. Now, in the immediate mode, type in:

```
PRINT 1 + 2 + 4 + 8 + 16
```

When you press RETURN, your result does not appear in the alpha window. Instead, it appears in the beta window. To see why, list your program and look at the *last* WINDOW OUTPUT statement. It is to window 2, not the alpha window, window 1. Therefore, until you tell your Amiga that the output is to go to window 1, the alpha window, it will continue to send all results to the beta window. To fix that, from the immediate mode, type in:

```
WINDOW OUTPUT 1
```



Now, go ahead and add up some numbers in the alpha window. This time the sum will appear in the alpha window. Of course you would normally make the window output changes from the program itself.

The only window you can use with your cursor is window 1. Go ahead and place the pointer on the beta window and click the mouse button. The title bar will darken, indicating the window is selected, but there will be no cursor; if you type in characters from the keyboard, they will not appear on the screen. On the other hand, since the alpha window is window 1, when it is selected, it darkens and the cursor appears. However, you can input into your current window even if it is other than window 1 when input is required. For example, the next program places characters on the screen until you press the vertical character bar. Then it goes into a "calculator window" for input and output:

Set.Windows:

```
WINDOW 1, "Writer", (1,1)-(600,180),15
WINDOW 2, "Calculator", (400,1)-(600,100),15
WINDOW OUTPUT 1
WIDTH 60
```

Writer:

```
WHILE Typo$ <> "@"
  TYPE$=INKEY$
  PRINT Typo$;
  IF Typo$="|" THEN GOSUB Calculate
WEND
WINDOW CLOSE 2
END
```

Calculate:

```
WINDOW OUTPUT 2
CLS : WIDTH 20
Num = -1
WHILE Num
  PRINT "Enter value"
  INPUT Num
  Total=Total+Num
  PRINT "Total=";Total
WEND
```

```
WINDOW OUTPUT 1
WIDTH 60
RETURN
```

Notice when you run this program how the typing goes behind the calculator window. That can be a problem, so let's see how to make the calculator window appear only when it is needed. Change the program to look like this:

```
Set.Windows:
WINDOW 1, "Writer", (1,1)-(600,180),15
WINDOW OUTPUT 1
WIDTH 60

Writer:
WHILE Typo$ <> "@"
  Typo$=INKEY$
  PRINT Typo$;
  IF Typo$="|" THEN GOSUB Calculate
WEND
END

Calculate:
GOSUB Calc.Window
WINDOW OUTPUT 2
CLS : WIDTH 20
Num = -1
WHILE Num
  PRINT "Enter value"
  INPUT Num
  Total=Total+Num
  PRINT "Total=";Total
WEND
WINDOW CLOSE 2
WINDOW OUTPUT 1
WIDTH 60
RETURN

Calc.Window:
WINDOW 2, "Calculator", (400,1)-(600,100),15
RETURN
```

---

By using the WINDOW CLOSE statement in the 'Calculate' routine, you can turn the 'Calculator' window off when it's not being used. When it is needed, the 'Calc.Window' routine makes it reappear. When using multiple windows, it's wise to have each window as a separate subroutine so that it can be turned on and off as needed.

Add the following window to the end of the last program to provide instructions as to what keys to press for calculations and exit. After

```
WINDOW 1, "Writer . . .
```

type:

```
GOSUB Inst.Window
```

At very end of program:

```
Inst. Window:
```

```
WINDOW 3, "Instructions", (1,1)-(310,10),1  
WINDOW OUTPUT 3  
COLOR 0,1  
PRINT "|" to Calculate - '@' to Exit.";  
RETURN
```

It's also a good idea to add a couple of PRINT statements at the beginning of the 'Writer' routine so that the output won't be covered by the new window.

## SCREEN WORK

---

Besides having a WINDOW statement, there is also a SCREEN statement. In the next chapter on graphics, we will see how SCREEN is used with color and drawings, but it has some important implications for use with screen formatting, so we will introduce it here.

To get started, look at this program using the SCREEN statement:

```
Fat.Characters:
```

```
SCREEN 2,320,200,5,1  
REM Last value in the following WINDOW statement  
REM indicates the screen number.
```

```

WINDOW 1, "Characters", (1,1)-(280,180),15,2
Alphabet$="abcdefghijklmnopqrstuvwxy"
Alpha$=UCASE$(Alphabet$)
PRINT Alpha$
PRINT Alphabet$

```

The first line sets the five screen parameters. They include:

1. Screen number or id
2. Horizontal pixels for screen, 1-640
3. Vertical pixels for screen, 1-400
4. Color number value:

<b>Value</b>	<b>Colors</b>
1	2
2	4
3	8
4	16
5	32

5. Screen mode: Resolution and interlacing

<b>Mode</b>	<b>Resolution</b>	<b>Interlaced</b>
1	Low	No
2	High	No
3	Low	Yes
4	High	Yes

In our example, we used an identification number 2, a width of 320 and height of 200 with 32 colors and non-interlaced low resolution screen. As a result, the output showed "fat" letters and a screen we cannot use for immediate mode input.

To link our window to a screen, we had to add a parameter to the WINDOW statement. The value 2 after the 15 indicates that output in Window 1 will be with Screen 2. As you can see, there is no need for windows and screens to have the same identification number. Experiment with SCREEN by changing its values. We'll be doing more with it in the next chapter on graphics and colors.

---

## SUMMARY

---

This chapter concentrated on controlling screen output. The more use you can make of your Amiga screen, the more output it can handle. Like everything else that becomes larger, your screen output must be managed in order to keep it under control. If it gets out of hand, it will scroll off the screen, be confused with mixed output and just look messy. Using the `POS(0)` and `CSRLIN` functions, you can keep track of the cursor's horizontal and vertical positions. When it reaches a point near scrolling, you can hold the screen until the user has had a chance to see everything needed. Similarly, using different algorithms, it's possible to use more of the screen to place output anyway you wish.

If a single window is not enough, you can place output in as many windows as you can create on the screen. This gives you the option of separating different kinds of output in different windows, or bringing on required output only when needed in a separate window. The `WINDOW` statement in Amiga's BASIC makes that very easy. Combined with the `SCREEN` statement, it is even simpler.



## Drawing With Graphics

Your Amiga can produce spectacular graphics. There are several aspects of Amiga graphics, but we will divide them into two major categories: drawing and animation. This chapter will focus on creating images, and the next chapter will examine animating images. Much of what we use in the next chapter will be introduced in this chapter, so before you have a go at animating graphics, you should finish this chapter.

### PIXELS

---

Graphics are produced with little “dots” on the screen. They are referred to as “pixels,” and each is made of a beam of light showing up on the screen. Your screen is made up of different numbers of rows and columns of pixels, depending on your screen mode. In the last chapter, when we introduced SCREEN we noted there were four modes, depending on which, there are different numbers of pixels:

1. Horizontal=320, Vertical=200
2. Horizontal=640, Vertical=200
3. Horizontal=320, Vertical=400
4. Horizontal=640, Vertical=400

The default screen mode on your Amiga is 2, which means you have a 640 by 200 pixel matrix in which to create graphics without changing any screens.

To light up a pixel on your screen, use the statement:

```
PSET (X,Y),C
```

where X is the horizontal position, Y the vertical and C the color. PSET works just like LOCATE in the text mode. (Think of PSET as Pixel-SET.) The value C sets the color, or if no color is indicated, it defaults to the current foreground color. Try the following:

```
CLS  
PSET (320,100)
```

That will place a cursor in the center of your screen. Now try this next to see the horizontal and vertical parameters:

```
FOR X= 1 TO 640  
  PSET (X,100)  
NEXT X  
FOR Y=1 TO 200  
  PSET (320,Y)  
NEXT Y
```

This time you got a cross intersecting in the middle of the screen. You should now be able to place a pixel anywhere you want on the screen. For practice, see if you can place one in each of the four corners of the screen.

## **THE AMIGA'S COLORFUL PALETTE**

---

The main advantage of Amiga graphics is the wide variety of colors available. (If you do not have a color TV or monitor, the colors will appear as different shades of black and white. However, we will assume the use of a color monitor as standard. If you have a color television, use the low resolution mode for best results. If the colors are not what you expect, adjust the color on your monitor or TV.)



---

The PALETTE statement has four parameters:

1. Number
2. Red value (0.0–1.0)
3. Green value (0.0–1.0)
4. Blue value (0.0–1.0)

To get started, we'll keep it simple. If any of the color values is 1 and the others are 0, then the color with a value of 1 will appear predominantly. For example, the following program will produce red letters and screen borders:

```
PALETTE 1,1,0,0  
PRINT "Red"
```

Change that to:

```
PALETTE 1,0,1,0  
PRINT "Green"
```

and you'll get a solid green result. The real power, though, comes by combining the various colors in values between 0 and 1. For example, by having the Red parameter at .5, you get a terra cotta red:

```
PALETTE 1,.5,0,0  
PRINT "Terra Cotta"
```

Combine other color combinations, and you can get all kinds of colors. If all of the values are 1, you create white, and if they are all zero, you get black. The thing to do is to write a program that will let you enter different values to see for yourself what colors you can create.

The next program lets you change the colors to any combination you can dream up. As soon as you enter values of 1 for each color, the program will exit and restore everything to normal. With each prompt, just enter three values, separated by a comma and press RETURN. For example, try:

```
1,.6,.67
```

to get a cherry flavored Amiga screen. You might want to make a note of any color values you'd like to use in a program.

```
WHILE Z$ <> "Q"
  COLOR 0,1
  CLS
  Get.Color:
  INPUT "Values for red,green,blue";Red,Green,Blue
  IF Red > 1 OR Green > 1 OR Blue > 1 THEN Get.Color
  PALETTE 1,Red,Green,Blue
  IF Red + Green + Blue = 3 THEN Z$="Q"
WEND
COLOR 1,0
CLS
```

## MULTIPLE COLORS

---

You can only get the number of colors on the screen at the time that is reserved by the SCREEN *depth* parameter. We'll use the high resolution screen (non-interlaced) to examine how to get several different colors on the screen at the same time. The maximum is 32, requiring a depth value of 5. We'll use 16 colors to get started; thus, we'll use the depth value of 4:

```
Color 16:
SCREEN 1, 600,380,4,2
WINDOW 1,"Color 16",(1,1)-(550,350),15,1
RANDOMIZE TIMER
FOR X= 1 TO 15
  PALETTE X,RND,RND,RND
  COLOR X,0
  PRINT "This color now"
NEXT
```

Notice how long it took to get everything on the screen. The more colors you use in a program, the longer it takes for it to run. Also note that we used only 15 of the available 16 colors. If you change the loop, from 1 TO 15 to 0 TO 15, you'll get all 16 colors, except that the first one will be invisible. The RND function gave us random colors, and since we

used the RANDOMIZE seed with the TIMER, we never know what the colors will be. Run it several times to see what different combinations you will get.

If you use a depth of 5 for 32 separate palettes, you'll need low resolution graphics. To get an idea of how to go about that, run this revision of the 16 color demonstration:

Color.Gen:

```
SCREEN 1, 300,180,5,1
WINDOW 1,"Color32",(1,1)-(275,160),15,1
RANDOMIZE TIMER
FOR X= 1 TO 31
  PALETTE X,RND,RND,RND
  COLOR X,0
  IF X> 16 THEN LOCATE X-16,10
  PRINT "Lo Color"
NEXT
PALETTE 1,1,1,1
COLOR 1,0
```

By combining your PSET and PALETTE statements, the following program makes some colorful spots in low resolution and shows you some tricks for defining color:

Sparkler:

```
SCREEN 1,320,200,3,1
WINDOW 1,"Sparkles",(10,10)-(270,170),15,1
GOSUB Color.Palette
FOR Hue= 1 TO 7
  RANDOMIZE TIMER
  FOR More = 1 TO 9
    COLOR J,0
    X%=INT (RND * 270)
    Y%=INT (RND * 170)
    PSET (X%,Y%),Hue
  NEXT More
NEXT Hue
COLOR 1,0
END
```

Color.Palette:

```
Yellow: PALETTE 1,1,1,.13
```

```

Cherry: PALETTE 2,1,.6,.67
Fire.Engine: PALETTE 3,.93,.2,0
Lime: PALETTE 4,.73,1,0
Brown: PALETTE 5,.8,.6,.53
Aqua: PALETTE 6,0,.93,.87
Gray: PALETTE 7,.73,.73,.73
RETURN

```

## GETTING IN SHAPE

---

OK, now that we have seen how to draw lines the hard way with PSET, let's look at doing some things the easy way. Instead of having loops to draw lines, we can use the LINE statement to draw lines, boxes and fill in the boxes! The general format is:

```
LINE (X,Y) - (X1,Y1), COLOR
```

There's more to the format that we will get to in a second, but to get started, we will draw a diagonal line we could not easily do with PSET:

```

Red.Line:
CLS
PALETTE 1,1,0,0
LINE (0,0) - (640,400),1

```

Now let's do something more with LINE. Instead of a line we will make a rectangle. To do that we enter B at the end of our LINE. The B (for Box) will use the X,Y coordinates for the opposite corners of the rectangle:

```

Black.Box:
PALETTE 1,0,0,0
LINE (10,10)-(200,100),1,B

```

This time, instead of getting a diagonal line, you got a Box. Add an F directly after the B. The F is for "Fill." See if you can guess what happens when the F is added:

```

Pink.Box:
PALETTE 1,1,.6,.67
LINE (10,10)-(200,100),1,BF

```

The box is filled with the palette color!

All of this may seem interesting, but what can you do with colored boxes? Well, for one thing, they are great for making graphs. We'll make a bar graph with elongated vertical rectangles using the LINE statement. Our first graph will be a simple one that makes three vertical bars, each in a different color. We will INPUT the values for each graph limited only by the vertical "window" we will use. Instead of crowding everything into our 400 vertical pixel screen, we will only use 380 vertical pixels. In this way we will have room for labels and other enhancements at the bottom of our chart:

```
Scrn.Win:
  SCREEN 1,300,190,2,1
  WINDOW 2,"Graph 1", (1,1)-(275,170),15,1
  WINDOW OUTPUT 2
EZ.Graph:
  CLS
  FOR X = 1 TO 3
  Get.Plot:
    PRINT "Value for bar #";X;
    INPUT Value(X)
  Trap:
    IF Value(X) > 150 THEN Get.Plot
  NEXT X

Make.Graph:
  GOSUB Graph.Hues
  COLOR 0,0
  CLS
  FOR X=1 TO 3
    Start=50 * X
    Finish = Start + 40
    Plot = 150 - Value(X)
    COLOR X,0
    LINE(Start,Plot) -(Finish,150),,BF
  NEXT X

  LINE (0,155) - (300,155)
  WINDOW OUTPUT 1
  END
```

```
Graph.Hues:  
  Red: PALETTE 1,1,0,0  
  Green: PALETTE 2,0,1,0  
  Blue: PALETTE 3,0,0,1  
  RETURN
```

Let's see how the graph was made:

1. Values for the graph were entered in the array Value(X).
2. The variable 'Start' used 50 times X to have the bars start at positions 50, 100 and 150.
3. The variable 'Finish' set the width of each bar to be 40 pixels since it uses the value of Start as an offset. That leaves 10 pixels between each bar.
4. The variable 'Plot' specified the starting location of the vertical position by subtracting the value of Value(X) from 150, the maximum value of the plot. This was done because the vertical pixels' values start at the top of the screen, and we wanted to have the higher values placed higher on the screen. That is, since 5 is "higher" on the vertical screen than, say 20, by subtracting Value(X) from 150, the value 5 is now 145 and 20 is 130. Therefore, when plotting the bars, the 20 (130) bar will appear "higher" than the 5 (145) bar.
5. The 'Make.Graph' routine draws the bars by using the variables we defined. The value 150 is a constant representing the pixel position where all the bars begin. The whole thing is run through a FOR/NEXT loop between the X variable generating 3 different palette colors, and 3 bars. Finally, a bottom line is drawn, and in Line 190 the output window is returned to 1 so that we can see output from the default window. (Things get very strange if you forget to do that.)

We took a very simple example, limiting the number of bars and the maximum value. This is fine for bar charts that have a maximum value of 150 and where you need only three parameters. But what about having more graphs, higher values and perhaps labels? This is a little trickier, but it can be done. To do this, we have to set up a ratio for a maximum value relative to any value we enter, adjust the width of our bars depending on the number of entries we make and figure out how to place the labels where we want them. By using good algorithms, we can let the computer figure all this out for us. Let's give it a try:

```
Smart.Plot:
  CLS
Get.Max:
  INPUT "Maximum value";MV
  RATIO = 179.99/MV
Plot.Num:
  PRINT
  INPUT "Number of plots(MAX=144) ";N%
  IF N% > 144 THEN Plot.Num
  DIM VALUE(N%)
  CLS

Plot.Info:
  FOR X=1 TO N%
  Get.Plot:
    PRINT "Value for plot";X; "(Maximum value=";MV;)"
    INPUT "->";VALUE(X)
    IF VALUE(X) > MV THEN Get.Plot
    VALUE(X)= INT(VALUE(X) * RATIO)
  NEXT X

Make.Graph:
  CLS
  FOR X = 1 TO N%
    HORWIDTH = INT (640/N%): Start = HORWIDTH * (X-1)
    Finish = Start + INT (HORWIDTH/2)
    Plot = 180 - VALUE(X)
    LINE(Start,Plot) - (Finish,180),,BF
  NEXT X
LINE (0,181) - (640,181)
```

In comparing this second graph program with the first, you can see the similarities. However, there are some important differences.

First, in the "Get.Max" routine it was necessary to enter the maximum values for the plots and the number of plots. This gives the program a good deal more flexibility than our first one with only three plots and a maximum value of 180.

Second, the variable RATIO determined the number by which we would have to multiply our plot values to make full use of the 180 vertical pixels (179.99 was used for precision).

Third, in the graph making block, we had to determine the horizontal width of the bars with the variable HORWIDTH. We used the INT function in the process. This function turns values into integers (whole numbers), rounding downwards. We did this since we wanted our LINE statement to have integers for plotting the bars. (Your Amiga would not know what to do with a plot at 103.45; it expects whole numbers for plots.) Thus, both Value(X) and HORWIDTH were made into INTEGER values. Depending on the number of plots, the bar size varies. We did this to make the chart use the maximum amount of screen space without going over boundaries.

The only thing we left out of our graph is some kind of label. There are a lot of different labels we could put in, but for simplicity, let's just number our bars. This will show us how to mix text and graphics. The important part will be to use LOCATE to place our text relative to where our bars are going to be printed on the screen. Since we are using a 640 by 200 matrix with our graphics and a 62 by 20 (assuming a WIDTH value of 62) matrix with our text, we are going to have to figure out an algorithm to translate our graphic locations into text locations. To get started we have to determine the relationship between our "plotting" points for text relative to graphics by dividing the vertical and horizontal graphic maximums by the text maximums. We will use the variable LV and LH to represent LOCATE VERTICAL and LOCATE HORIZONTAL respectively. Thus, we would have:

```
LV = INT (200/20)
LH = INT (640/62)
```

Next, we will have to find where each bar is being placed. Since we used the variable START to set the beginning of our horizontal bars, we can determine where our text will go. However, since we placed our first bar at location zero, and we do not want to divide zeros, we will "pad" the START variable. We will call the horizontal text position LSTART.

```
LSTART = 1 + INT ((1+START)/LH)
```

Our vertical position is a little easier since we know the bottom of the bars are at graphic position 180. However, we want our labels below the bars, so we will use position 182:

```
VSTART = 182/LV
```



Now to label the bars on your graph, insert the following lines:

```
Label.Find:
  LV = INT (200/20) : LH = INT (640/62)
  LSTART = 1 + INT ((1 + START)/LH)
  VSTART = 182/LV
  LOCATE VSTART,LSTART
  PRINT X
```

We'll also change the bottom line to 150 (from 180) so there will be more room for the labels.

```
Label.Plot:
  WIDTH 62
  CLS
  Get.Max:
    INPUT "Maximum value";MV
    RATIO = 149.99/MV
  Plot.Num:
    PRINT
    INPUT "Number of plots(MAX=144) ";N%
    IF N% > 144 THEN Plot.Num
  DIM VALUE(N%)
  CLS
  Plot.Info:
    FOR X=1 TO N%
      Get.Plot:
        PRINT "Value for plot";X; "(Maximum value=";MV;")"
        INPUT "->";VALUE(X)
        IF VALUE(X) > MV THEN Get.Plot
        VALUE(X)= INT(VALUE(X) * RATIO)
    NEXT X
  Make.Graph:
    CLS
    FOR X = 1 TO N%
      HORWIDTH = INT (640/N%): START = HORWIDTH * (X-1)
      Finish = START + INT (HORWIDTH/2)
      Plot = 150 - VALUE(X)
      LINE(START,Plot) - (Finish,150),,BF
```

```
Label.Find:
  LV= INT(200/20) : LH= INT(640/62)
  LSTART= 1 + INT((1 + START)/LH)
  VSTART= 1+ INT (199/LV).
  LOCATE VSTART,LSTART
  PRINT X;
NEXT X
LINE (0,151) - (640,151)
A$=INPUT$(1)
```

The program is fine up to a point, but as you increase the number of plots beyond 15, the numbers begin to get off center and pretty soon there is a real mess. That's because the graphics are using finer plotting points and the multidigit numbers begin bumping into the next bar. However, the enhancements show you how to find where a label goes relative to a graphic plotting point. See what other labels you can LOCATE on your graph. (HINT: For a neat chart using the 12 months of the year labeled with the first letter of each month, create a big string called "JFMAMJJASOND" and, using MID\$, label each bar by month.) Also, you might want to add some color to the bars. See if you can add three colors in high resolution.

## **PLOTS FROM LAST PLOT: RELATIVE PLOTS**

---

So far, we have only examined PSET and LINE in terms of absolute plotting points. However, we can also make lines and points relative to the last plot. When using PSET, the STEP statement is read as an offset to the last plotted point. For example, enter the following:

```
CLS
PSET (20,20)
PSET STEP (20,20)
```

When you RUN the program, you can see the different locations of the white dots on your screen. One dot is at position 20 vertical and 20 horizontal. The other dot is at 20 + 20 vertical and 20 + 20 horizontal. Thus, even though the parentheses contained (20,20), the STEP statement made a difference in where the pixel was set. To make a diagonal line with PSET STEP, you could do the following:

```
CLS
PSET (1,1) : REM SET THE BEGINNING
FOR X = 0 TO 199
  PSET STEP (1,1)
NEXT X
```

You can also STEP backwards, just like with FOR/NEXT loops:

```
CLS
PSET (160,200)
FOR X = 1 TO 200
  PSET STEP (0,-1)
NEXT X
```

In the same way, you can use relative lines. The last plotted point is the starting point for your next line. This saves programming and makes it simpler to conceive graphic shapes. For example, the following draws an arrow, beginning with a regular line and then using offsets from the last plotted point:

```
Arrow:
CLS
LINE (0,100)-(319,100)
LINE - STEP (-50,-50)
LINE - STEP (0,100)
LINE - STEP (50,-50)
```

For an interesting effect, try the following using relative lines:

```
SCREEN 1,600,200,2,2
WINDOW 1, "Surprise", (1,1)-(550,180),15,1
GOSUB Get.Colors:
```

```
CLS
FOR x=0 TO 100
  PSET(x+100,x+50)
  LINE - STEP(-50,-50),1
  LINE -STEP (0,100),2
  LINE -STEP (50,-50),3
NEXT x
END
```

```
Get.Colors:
  PALETTE 1,1,0,0
  PALETTE 2,1,1,0
  PALETTE 3,1,0,1
  RETURN
```

You might note that this program is very similar to the first one except we changed the beginning point from a line to PSET. Experiment and see what you can create!

## AMIGA ART

---

We will now do some art work with the Amiga. So far all we have done is to make shapes using pixels, lines and boxes. We were able to fill in our boxes with the F function. Now we will see how to PAINT different parts of our screen.

### Paint

The PAINT statement allows you to indicate a point on your screen and then fill in everything from that point to any line. If you have an enclosed area, that area will be "filled" with color, and the rest of the screen will remain in the original background color. The general format for PAINT is:

```
PAINT (X,Y)
```

The following program draws a triangle and then fills it in:

```
CLS
LINE (160,100)-(160,50)
LINE - STEP (-100,50)
LINE -STEP (100,0)
PAINT (159,99)
```

The PAINT statement must be inside the lines of the figure to be filled. Thus, we placed our X,Y coordinates to the left and above the point

where we started to draw our triangle. Now change the line with PAINT to read:

```
PAINT (161,99)
```

The starting point will put the PAINT statement just outside of the triangle. Run the program again to see what happens. Instead of painting the triangle, it painted everything but the triangle. Now, to add color, we simply include a color in the same format as we did with PSET using the desired palette number for your color. Go back to the original program and add a PALETTE, and we'll add another triangle:

```
SCREEN 1,600,200,2,2  
WINDOW 1,"Color Paint",(1,1)-(550,180),15,1
```

```
CLS  
PALETTE 1,0,1,0  
PALETTE 2,1,0,0  
LINE (160,100)-(160,50),2  
LINE -STEP (-100,50),2  
LINE -STEP (100,0),2  
PAINT (159,99),2
```

```
LINE (260,100)-(260,50),1  
LINE -STEP (-100,50),1  
LINE -STEP (100,0),1  
PAINT (259,99),1
```

## Getting Around: Circle

If you tried to draw a circle with pixels, you would need either a good algorithm or a lot of patience and some graph paper. However, with the CIRCLE statement, making circles is easy. The general format is:

```
CIRCLE (X,Y) Radius, C
```

The X,Y coordinates specify the center of the circle and RADIUS, the radius. C is the color parameter. For example, enter the following:

```
CLS
CIRCLE (300,100), 50
```

There's your circle, and now add the following to fill it in:

```
PAINT (300,99)
```

Now, there's more to circles than just drawing and filling them in. You can also specify starting and ending points and aspect ratio:

```
CIRCLE (X,Y), RADIUS, C,S,E,AR
```

To get the points to begin and end, you have to use radians. These are parts of the circle in terms of pi (3.141593). Think of the circle as a clock using 12, 3, 6 and 9 o'clock positions. The starting position or zero is 3 o'clock and the circle is drawn counterclockwise from there:

```
12 (PI/2)
9 (PI)      3 (2*PI)
6 (3 * (PI/2))
```

Take a look at the following program to see how this works. We will use the variables TWELVE, THREE, SIX and NINE to represent the various positions on the circle:

```
CLS
PI=3.14159
TWELVE=PI/2
THREE = 2*PI
SIX = 3*(PI/2)
NINE=PI
CIRCLE (100,100),50,,NINE,THREE
```

This program makes your Amiga smile at you! By defining the radians in terms of positions on the clock, it's a lot easier to envision where your curved lines will begin and end.

Another feature of CIRCLE can be seen if the starting or ending position is negative. If so, a line will be drawn from center of the circle to the edge. Let's see what happens with the following program. (While we're at it, why not PAINT it too?)

```

CLS
PI=3.14159
TWELVE=PI/2
THREE = 2*PI
SIX = 3*(PI/2)
NINE=PI
CIRCLE (300,100),50,,,-THREE,-SIX
PAINT (300,99)

```

Notice that all we had to do to get a negative value was to place a minus (-) sign before our start and end variables, THREE and SIX. If you like pie charts, this part of the CIRCLE statement can come in handy.

**ASPECT.** Finally, we can change the aspect ratio of a circle to make ovals and oblong shapes. If this parameter is not specified, the default value is 2.25-1 in high resolution or roughly 0.44. As long as the aspect ratio is less than 1, the radius is measured horizontally. If the value is more than 1, then it is measured on the Y or vertical axis. For example, the following program will create three different ellipses with three different aspect ratios affecting the Y axis:

```

CLS
FOR X = 1 TO 3
CIRCLE (300,100),50,X,,,X : REM 4 Commas between X's
NEXT X

```

Now, to see the different horizontal aspect ratios change the CIRCLE statement to read:

```

CIRCLE (300,100),20,X,,, (1/(X+1))

```

By experimenting, you can make all kinds of different ellipses. With high resolution graphics, your ellipses are not as jagged, and even though you are limited to black and white, your results will be more precise.

## Filling an Area with Pattern

Another way to make shapes on your Amiga is with AREA, AREAFILL and PATTERN. Basically, using AREA as a relative STEPPing statement, you can plot several points of a polygon, and then fill the area defined by

the polygon with a PATTERN. The default pattern is a solid fill just as we saw with circles and rectangles. For example, the following program fills the area defined by the four area points with the current pattern:

```
AREA (1,1)
AREA STEP (300,100)
AREA STEP (-100,0)
AREA STEP (-140,-50)
AREA STEP (-60,-50)
AREAFILL
LOCATE 10,1
```

**PATTERN.** There are two ways to go in this section. Hang in there and try to understand how bit patterns work, or just look at the examples and wait until later to try and assimilate everything. In the last chapter, we tackled more of the advanced technical materials, but if you think in *simplified* terms as follows, you probably will understand how your Amiga makes graphics.

First, everything your Amiga and all other computers do can be boiled down to 1's and 0's. There's a counting system called 'binary' that is made up entirely of 1's and 0's used by computer people to give instructions to computers. The pixels on your screen are lit or unlit depending on whether a 1 or a 0 is signaled for a given position of your screen. The counting system in binary works like the decimal system you're used to, but it only has two digits instead of ten. Another counting system, hexadecimal, works on a base of 16, which we will see, is very useful. We will use hexadecimal (or hex as it's commonly called) to create graphic patterns. Let's count from zero to 16 and compare the three different numbering systems:

Decimal	Binary	Hexadecimal	Decimal	Binary	Hexadecimal
0	0	0	9	1001	9
1	1	1	10	1010	A
2	10	2	11	1011	B
3	11	3	12	1100	C
4	100	4	13	1101	D
5	101	5	14	1110	E
6	110	6	15	1111	F
7	111	7	16	10000	10
8	1000	8			



The hexadecimal system can count further than the other two systems before having to add a second digit. At 16, an interesting thing happens; both the hex and binary systems add a digit. Thus, numbering from 0–15, the hexadecimal system can handle all values with a single digit, and the binary with no more than four digits. If we can keep everything between 0 and 15, it will be possible to simplify our counting of binary numbers and control the 1's and 0's we will need for making patterns.

Imagine a line as a series of pixels, turned on or off, and a pattern as a stack of lines. To draw a 16 pixel line with 1's and 0's, we would have something like the following:

0101010101010101

That 16 digit binary number would be difficult to convert to either decimal or hexadecimal, but we could break it up into groups of four and then convert the four digit value into hex values like so:

Binary 0101 0101 0101 0101

Hex 5 5 5 5

Hex value = \$5555 (a dollar sign indicates a hex value)

That particular configuration broke down into four equal value groupings. The value 101 in binary equals 5 in hex. (We read 0101 the same as 101.) If we made a line with every other pixel lit, it would look "light" or "screened" compared with a solid line. That's exactly what the above configuration would do. Suppose, we wanted every other group of four pixels lit, we would have a pattern like the following:

Binary	1111	0000	1111	0000
Hex	<F>	<0>	<F>	<0>

Thus, the value \$F0F0 would result in a somewhat different line pattern than \$5555. By stacking up these lines, we can begin to have patterns made up of different line combinations. For example, let's make a checkerboard pattern. We would need lines alternating in solids and fills. The following would look a lot like a checkerboard:

```

1111 0000 1111 0000 (Hex=$F0F0)
1111 0000 1111 0000 (Hex=$F0F0)
0000 1111 0000 1111 (Hex=$0F0F)
0000 1111 0000 1111 (Hex=$0F0F)

```

If you follow so far, the rest is easy. The PATTERN statement establishes an array made up of the lines. The lines are defined as hexadecimal numbers. If you remember, we get hexadecimal values in Amiga BASIC using the '&H' symbol. Thus, &HF0F0 would give us \$F0F0. Forget about decimal, and just make the translations between binary and hexadecimal using the number conversion table above. Let's look at a program that will make a checkered pattern and then see how it was done:

```

Check.Patt:
  DIM Pattern1%(1)
  Pattern1%(0) =&HFFFF
  Pattern1%(1) =&H0
  PATTERN&HFFF,Pattern1%
Box:
  Point1: AREA(1,1)
  Point2: AREA STEP (300,0)
  Point3: AREA STEP (0,100)
  Point4: AREA STEP (-300,0)
  AREAFILL
  LOCATE 15,1

```

Each array must be a power of 2 (e.g., 2, 4, 8, 16, etc.), so the dimension of the array is a power of two minus one. (That's because there's the zero element in each array that counts toward the total.) In our example, we dimensioned an array called 'Pattern1%' with four elements numbered from zero to three. Using the PATTERN statement and the address \$FFF (&HFFF), we placed the array called 'Pattern1%' into memory. This time when the AREAFILL command was given, it filled with the checkered pattern. Any other kind of fill or paint will give the defined pattern. For example, enter the following at the end of the program:

```

CIRCLE (400,150),50
PAINT (401,151)

```

You'll get a checkered circle. Even your cursor looks blotchy since the pattern affects it too. Until you restore the pattern to the default solid one, it will continue to be blotchy. The following little routine restores everything for you:

```
Solid.Pattern:  
  DIM Solid%(1)  
  Solid%(0)=&HFFFF  
  Solid%(1)=&HFFFF  
  PATTERN &HFFF,Solid%
```

If you place that little routine at the end of your programs where you change the pattern, the default program will be restored when you're finished.

## SUMMARY

---

This chapter was just the first part of using your powerful Amiga graphics. By thinking of your screen as a pixel canvas, you can draw just about anything you want. Beginning with the individual pixels and the PSET statement, we could place pixels anywhere we wanted on the screen. The screen itself was set by SCREEN to give us options in terms of resolution, mode, size and color depth. Using LINE and CIRCLE, we saw how to make not only lines and circles but also boxes and ellipses. All of these can be colored in with PAINT if desired.

Various colors can be controlled with PALETTE giving the user an almost infinite set of possible hues by changing the combinations of red, green and blue settings. Multiple colors can be interchanged simultaneously on the screen. Finally, the "brush" that paints and fills can be altered with PATTERN. It is a little tricky to work with pattern changes since it involves making translations between binary and hexadecimal values, but you do have complete control over the patterns. With practice, it will be relatively simple and you'll learn a lot. Like everything else with your Amiga the key to getting the most out of it is practice and experimentation.



# Animation, Sprites and Bobs

---

## MOVING GRAPHICS

---

This chapter is going to show you how to move graphics. Not only will we be moving graphics with animation techniques, we'll also see how you can use the mouse to interact with your graphics programs. (While we're at it, we might as well look at some general control moves with the mouse too.) In part we'll be using the techniques we learned in Chapter 12, but we'll also introduce new graphic concepts using "sprites" and "bobs" that are particularly well suited for animation.

Animation can be used in games and special effects. However, we will only touch upon some elementary examples to provide you with the concepts of how animation works. Basically, you plot a position and then plot a new position and cover up the old position. This gives the appearance of a "pixel" moving since it is plotted from one adjacent position to another and its previous position "erased" with a blue plot. We can use the PRESET statement to erase our pixel. PRESET works like PSET, but if there is no color parameter, it defaults to blue. For example, the following program will "move" a pixel from side to side by plotting it in white with PSET and erasing it with PRESET:

```
CLS
X=320 : Y=100
FOR K = 1 TO 20
  PSET (X,Y),2
  FOR HOLD = 1 TO 1000 : NEXT HOLD
  PRESET (X,Y)
  PSET (X+9,Y),2
  FOR HOLD = 1 TO 1000 : NEXT HOLD
  PRESET (X+9,Y)
NEXT K
```

If you watched carefully, you saw what appeared to be a moving pixel. However, we can see from what we entered, that it is really a matter of plotting a pixel in one position, erasing it and drawing it in another position. For a more dramatic example, the following program will start in the upper left hand corner of your screen and “bounce” a white pixel around your screen.

```
CLS
FOR XY = 0 TO 199
  PSET (XY,XY),1
  FOR HOLD=1 TO 10 : NEXT HOLD
  PRESET (XY,XY)
NEXT XY
REM *****
REM UP AND RIGHT
REM *****
Y=199
FOR X=200 TO 300
  Y=Y-1
  PSET (X,Y),3
  FOR HOLD = 1 TO 10: NEXT HOLD
  PRESET (X,Y)
NEXT X
REM *****
REM UP AND LEFT
REM *****
J=Y
FOR Y=J TO 1 STEP-1
```

```
X=X-1
PSET (X,Y),1
FOR HOLD = 1 TO 10 : NEXT HOLD
PRESET (X,Y)
NEXT Y
REM *****
REM DOWN AND LEFT
REM *****
J = Y
FOR Y= J TO 199
  X=X-1
  PSET(X,Y),1
  FOR HOLD = 1 TO 10 : NEXT HOLD
  PRESET(X,Y)
Next Y
```

By experimenting with different algorithms, you can do anything from making letters and numbers to animated games.

## MOVING OUT WITH PUT AND GET

---

At this point we are going to have to slow down and carefully examine the graphics statements PUT and GET. These two statements have a special use and meaning with graphics that are entirely different from the PUT and GET used in file handling in the chapter on random access files.

The best way to envision how GET and PUT work is to think of a specified area of your screen with graphics. The GET statement “scans” that specified area of your screen, and sticks the graphics into a special array. Then, using that array name, PUT places that graphic anywhere you specify on the screen. In other words, you could draw anything on the screen and then have it moved wherever you want without having to redraw it. All you would have to do is to PUT it somewhere else. For animation, GET and PUT allow you to draw whatever you want and move them smoothly across the screen.

For the most part PUT and GET are just like all the other graphic statements we have been using in terms of locating coordinates and

placing graphics on those coordinates. However, before you get going, it is necessary to figure out the dimension of the special integer array for your graphics. To do that we use the following formula:

$$\text{BYTE\$} = 6 + (\text{YLen}) * 2 * \text{INT}((\text{XLen} + 16) / 16) * \text{D}$$

(Ylen = Vertical length; YLen = Y2-Y1+1. XLen = Horizontal width. XLen = X2-X1. D = Depth. Default is 2, but depends on SCREEN parameter.)

The dimension of the array is

<DIM C%(BYTE%/BE)>

(BE represents the Bytes per Element. Integer = 2 bytes; Single precision = 4 bytes; Double precision = 8 bytes. C% = Array name.)

We'll stick with the integer arrays to keep it simple (and fast).

To use GET and PUT, we take the following steps:

**STEP 1.** Determine the size of the graphic area you will use. Think of the area in terms of the number of horizontal and vertical coordinates you will need for your graphic display. For example, you might want a 10 by 15 "screen" for your graphics made up of 10 rows and 15 columns. Define the variables XLen and YLen.

**STEP 2.** Decide what screen depth and type of variable you want to use. Define the variables BE and D.

**STEP 3.** Determine the value of BYTE% by placing the variables in the formula, and then the array size by dividing BYTE% by BE. Dimension your array (BYTE%/BE) using any legal variable name you want. Integer arrays should be followed by a percent sign.

Once the first three steps are completed, the rest is easy. The secret to using GET and PUT in graphics is carefully organizing your program up to this point. From now on, things are much simpler.

**STEP 4.** Draw your graphics within the limits of the graphic area you defined. If your "graphic screen" is 10 by 15, for example, all of your graphics have to be within 10 rows and 15 columns.

**STEP 5.** Place your graphics in your special graphic array with GET. The format for GET is:

```
GET (X1,Y1) - (X2,Y2), C%
```





```

REM %%%%%%%%%%%
REM Get Graphics
REM %%%%%%%%%%%
GET (1,1)-(8,8),C%
REM %%%%%%%%%%%
REM Move Graphic
REM %%%%%%%%%%%
FOR X=1 TO 500
  PUT (X,100),C%
  PUT (X,100),C%
NEXT X
PUT (X, 100),C%
Move cursor out of the way
LOCATE 19,1

```

When you RUN the above program, you will see a little ball appear in the upper left hand corner. That's to show you the original graphic you created and its location. You will then see the same ball move from the lower left towards the right of the screen. It doesn't matter what you draw. As long as it is placed in the graphic array with GET, it can be easily moved around with PUT. Let's create a space fighter, some stars and planets and then zoom around with our Amiga Space Fighter!

```

Array.Fighter:
  SCREEN 1,620,200,2,2
  WINDOW 1,"Zoom",(1,1)-(590,180),15,1
Le.Palette:
  PALETTE 1,1,0,0 'Red
  PALETTE 2,0,.93,.87 'Aqua
  PALETTE 3,.73,1,0 'Lime
  CLS
Milky.Way:
  REM =====
  REM Stars in the Galaxy
  REM =====
  RANDOMIZE TIMER
  FOR X= 1 TO 50
    StarH = 1 + INT(RND *(62+1))
    StarV=1 +INT(RND * (20+1))
    LOCATE StarV,StarH

```

```
StarColor=StarColor + 1
IF StarColor = 4 THEN StarColor=0
COLOR StarColor
PRINT "*";
NEXT X
REM =====
REM Add Planets
REM =====
CIRCLE (50,50),30,1
PAINT(50,50),1
CIRCLE (400,150),20,1
PAINT(400,150),1
CIRCLE (220,30),50,2
PAINT(220,50),2
Set.Array:
REM =====
REM Array Setup
REM =====
D=2
BE=2
XLen=17
Ylen=17
Byte%= 6 +(Ylen) *2* INT((XLen + 16)/16)*D
DIM SF%(Byte%/BE)
REM =====
REM Draw Fighter
REM =====
CIRCLE (10,100),5,3
PAINT (10,100),3
LINE (2,100)-(18,100),3
LINE (2,92)-(2,108),3
LINE (18,92)-(18,108),3
REM =====
REM Get Fighter in Array
REM =====
GET (2,92)-(18,108),SF%
Up.And.Away:
REM =====
REM Take Off!
REM =====
```

```
PUT (2,92),SF%
FOR X=2 TO 500
  PUT (X,100),SF%
  PUT (X,100),SF%
NEXT X
J=X : Y=100
FOR X=J TO 10 STEP -1
  PUT (X,Y),SF%
  PUT (X,Y),SF%
  Y=Y-1 : IF Y < 12 THEN Y=12
NEXT X
J=X : K=Y
FOR Y=K TO 160
  PUT (J,Y),SF%
  PUT (J,Y),SF%
NEXT Y
PUT (J,Y),SF%
```

As your little space fighter flew around the screen, you may have noticed that it did not disturb any of the text created “stars” or the graphic planets. The fighter flickered some and the color seemed to fade as it moved, but overall, it looked pretty neat. As it passed across the planets’ faces, it did not leave a “scar.”

You can speed up your fighter by including a STEP value and putting it at 2 or more. You can put a pause loop between the identical PUT statements and the moving figure will not be as pale, but it will slow down considerably. By combining STEP and pauses, you will be able to adjust the fighter’s movement and its appearance so that it looks good to you.

A final aspect of PUT is the use of AND and OR. The default mode of XOR is ideal for animation where you do not want to disturb the existing images on the screen. However, you may want to cover over them or wipe them out or get some other desired effect by using AND or OR. For example, you may wish to create a “paint brush” that will shade in parts of the screen with various colors using a combination of AND or OR. Add/change the following lines to your Amiga Fighter program to see one effect:

```
Warp.Speed:
W=Y
X=J
```

```
OR.Effect:
FOR Y=W TO 0 STEP -4
  PUT (X,Y),SF%,OR
  PUT (X,Y),SF%,OR
  X=X+5
NEXT Y
X=J-2
```

```
AND.Effect:
FOR Y=W TO 0 STEP -4
  PUT (X,Y),SF%,AND
  PUT (X,Y),SF%,AND
  X=X+5
NEXT Y
```

In the next chapter we're going to discuss sound. You can combine sound with your animated graphics to create arcade style games. Now, we're going to see how to control graphics with the mouse.

## MOUSE CONTROL

---

We saw how to use the mouse with menus in programs, and here we're going to see how to use it with graphics. Now that you're used to finding the X and Y pixel coordinates on your screen in graphics, controlling them with the mouse will be relatively simple since it too is located with X and Y coordinates.

The MOUSE function returns the following information:

### **Mouse(0).**

0. Left mouse button is not currently pressed.
1. Left mouse button is not currently pressed, but it was pressed since the last time there was a call to the MOUSE(0) function. Mouse functions 3-6 return the start and end points in this situation.
2. Left mouse button is not currently pressed, but it was clicked twice since the last time there was a call to the MOUSE(0) function. Mouse functions 3-6 return the start and end points in this situation.

3. Left mouse button is not currently pressed, but it was clicked *three times* since the last time there was a call to the MOUSE(0) function. Mouse functions 3-6 return the start and end points in this situation.
- 1. Left mouse button is currently pressed *once*. Mouse functions 1-6 return the current, start and end points in this situation. (-2 and -3 return similar information for the button having been pressed two and three times respectively.)

For the following functions, we will use X and Y to be:

X=Horizontal coordinate

Y=Vertical coordinate

**MOUSE(1).** Current X position on screen since last time left button was pressed.

**MOUSE(2).** Current Y position on screen since last time left button was pressed.

**MOUSE(3).** Starting X position on screen since last time left button was pressed *prior* to a call to MOUSE(0).

**MOUSE(4).** Starting Y position on screen since last time left button was pressed *prior* to a call to MOUSE(0).

**MOUSE(5).** The X position of mouse cursor if button is pressed when MOUSE(0) was called. If button was not pressed when call to MOUSE(0) was made, then it is the X position when button was released.

**MOUSE(6).** The Y position of mouse cursor if button is pressed when MOUSE(0) was called. If button was not pressed when call to MOUSE(0) was made, then it is the Y position when button was released.

The best way to see how this works is to write a program that will show you what's going on with these different functions. In other words, we're going to put the Amiga to work as a tutor for itself. (To stop execution on this program, press the right button and select 'Stop' from the Menu Bar.) Be sure to experiment with all the different positions with your mouse and quickly press the button different numbers of times to see how it affects the value of MOUSE(0).

```
MouseTell:
  COLOR 0,1
  PRINT "Press left mouse button to start"
  COLOR 1,0
  Flag=1
```

```
MouseClick:
  IF MOUSE(0)=0 THEN MouseClick
ShowInfo:
  IF Flag=1 THEN Flag=0 : CLS
  LOCATE 1,1
  PRINT "Left button: MOUSE(0)=";MOUSE(0)
  LOCATE 3,1
  PRINT "Current X: MOUSE(1)=";MOUSE(1)
  LOCATE 5,1
  PRINT "Current Y: MOUSE(2)=";MOUSE(2)
  LOCATE 7,1
  PRINT "Starting X: MOUSE(3)=";MOUSE(3)
  LOCATE 9,1
  PRINT "Starting Y: MOUSE(4)=";MOUSE(4)
  LOCATE 11,1
  PRINT "Ending X: MOUSE(5)=";MOUSE(5)
  LOCATE 13,1
  PRINT "Ending Y: MOUSE(6)=";MOUSE(6)
  GOTO MouseClick
```

After experimenting with the 'MouseTell' program for a while you should have some idea of the different mouse functions. Now, we're ready to make a program using graphics and the mouse. By incorporating PSET into a mouse routine, and using the X and Y positions of the mouse pointer, we can create a simple drawing program:

```
MouseDraw:
MouseClick:
  IF MOUSE(0)=2 THEN END
  IF MOUSE(0)=0 THEN MouseClick
  X=MOUSE(1)
  Y=MOUSE(2)
  PSET(X,Y)
  GOTO MouseClick
```

That was simple, and it works fine for making sketches. All you have to use are the MOUSE(1) and MOUSE(2) functions. Now, let's see about using the MOUSE(3-6) functions in a graphics program. Also, let's see how to exit the program by using a "triple click" instead of having to rely on the pull down menu "Stop" method. Instead of using PSET, this

program will use LINE and make graphic box designs created by holding down the mouse button and “dragging” different designs:

```

MouseBox:
MouseClicked:
  IF MOUSE(0)=3 THEN END 'Three clicks to exit
  IF MOUSE(0)=0 THEN MouseClick

Make.Box:
  'Hold left button down and drag
  'across screen
  X1=MOUSE(3)
  Y1=MOUSE(4)
  X2=MOUSE(5)
  Y2=MOUSE(6)
Box.Here:
  LINE (X1,Y1)-(X2,Y2),,b
  GOTO MouseClick

```

You can do the same thing with CIRCLE by replacing the 'Box.Here' routine with the following:

```

Circle.Here:
  IF X2 > X1 THEN Radius = X2-X1
  CIRCLE (X1,Y1),Radius
  GOTO MouseClick

```

Now, if we use those X and Y values returned by the mouse to PUT our graphics, we can move pictures with the mouse. Let's make our own arrow pointer for the mouse and move it around with PUT:

```

MousePut:
  DIM Arrow%(116)
  LINE(1,5)-(20,11 ),,BF
  LINE(30,8)-STEP(-10,-8)
  LINE-STEP(0,16)
  LINE-STEP(10,-8)
  GET (1,1)-(30,20),Arrow%
ClickMouse:
  IF MOUSE(0)=0 THEN ClickMouse
  IF ABS(X-MOUSE(1)) > 3 THEN MoveIt
  IF ABS(Y-MOUSE(1)) < 3 THEN ClickMouse

```



```
MoveIt:
  PUT(X,Y),Arrow%
  IF Flag=0 THEN Flag=1 : CLS
  X=MOUSE(1)
  Y=MOUSE(2)
  PUT (X,Y),Arrow%
  GOTO ClickMouse
```

When you run the program, notice where the regular pointer connects with the fat arrow we made. See if you can change the program so that the points of both arrows connect whenever it moves. To make an “arrow brush” change the second PUT statement to:

```
PUT(X-2,Y-2),Arrow%
```

The main point is to try different things. By combining several techniques from this and the previous chapter, you will soon be able to create a drawing program that will do anything you want on your Amiga.

## SPRITES AND BOBS

---

In a file called “Basic Demos” on your “Extra” disk (the one with AmigaBasic), there’s a program called ObjEdit. You’ll need it and two support files, in the same general file, called ‘Library’ and ‘graphics.bmap’ for this next section. They are used to edit and create sprites and bobs. First, we’ll make a sprite and then see how to use the OBJECT and COLLISION statements to use them. Then we’ll do the same with a bob and explain the differences between the two.

## USING OBJEDIT

---

The ObjEdit program makes it very simple to create sprites and bobs. Just run the program, and you will be given a choice of making sprites (1) or bobs (0). The first time through choose sprites.

**SPRITES.** In the upper left hand corner of your screen you’ll see a little vertical rectangle that is your “canvas” for drawing sprites. Press the right mouse button and you’ll see menus for ‘File,’ ‘Tools’ and ‘Enlarge.’

At the bottom of your screen, you'll see a color bar that you use to select your drawing colors. Just point and click to choose a color.

To get started, select 'Oval' from your Tools menu. Using the orange color, draw an oval at the top of your canvas by placing the pointer in your little canvas, holding down the left button and dragging the pointer. It will look like a box, but it will turn into an oval when you release the left button. Next, choose 'Paint' from your tools, and color the oval by clicking the pointer inside the oval. Next, click the white and the rectangle in the Tools menu and draw a rectangle about halfway down the oval and then fill it with white "paint." It should look like a bullet. Finally, choose black, and draw some fins to make a sprite rocket.



To touch up your rocket, choose  $4 \times 4$  from the enlarge menu. When you're in the enlarged state, the only tool you can use is the pen. That's why we first did the basic drawing in the little canvas before using the enlarged one.

When you're finished touching up your rocket, choose 'Save as ...' from the File menu and save the rocket under the name "Rocket.Sprite." Be sure to use the name "Rocket.Sprite" since we'll be using it in our example program. Choose 'Quit' from the File menu, and then NEW to get the ObjEdit program out of memory.

## **DISPLAYING SPRITES AND BOBS WITH OBJECT**

---

You use the same statements to manipulate sprites and bobs, a variation on the OBJECT statement. There are 16 different OBJECT statements you can use for manipulating your sprites and bobs. First, you need to get your sprite or bob in memory. To do that, use OBJECT.SHAPE # where the number(#) can be anything from 1 upwards depending on how much memory is left. The procedure for placing a sprite or bob in memory that has been created with the ObjEdit program is as follows:

```
OPEN "Sprite/Bob Name" FOR INPUT AS 1
OBJECT.SHAPE 1,INPUT$(LOF(1), 1)
CLOSE 1
```

The number (file number) in the first line can be from 1 to 255, and the values after LOF must correspond to that number. The OBJECT.SHAPE number has no bearing on the file number. If we changed the OBJECT.SHAPE number from 1 to 3, everything would work fine.

To position a sprite/bob on your screen, you use OBJECT.X # and OBJECT.Y #, positioning the X and Y coordinates respectively. For example:

```
OBJECT.X 1,320
OBJECT.Y 1,100
```

would place a sprite/bob in the middle of the non-interlaced high resolution screen.

Once positioned, you then turn on your object or objects with OBJECT.ON #. The object number is optional. If no number is used, all sprites/bobs will be turned on at once. Let's write a program to display our rocket in the middle of the screen:

```
Get.Sprite:
  OPEN "Rocket.Sprite" FOR INPUT AS 1
  OBJECT.SHAPE 1,INPUT$(LOF(1), 1)
  CLOSE 1
Show.Sprite:
  OBJECT.X 1,320
  OBJECT.Y 1,100
Turn.On:
  OBJECT.ON
```

If you wanted several sprites on at the same time, you would have to open a "file" for each sprite. For example, if we want several rockets at once, we would use the following:

```
Get.Lots:
  OPEN "Rocket.Sprite" FOR INPUT AS 3
  OPEN "Rocket.Sprite" FOR INPUT AS 4
  OPEN "Rocket.Sprite" FOR INPUT AS 5
```

```
OBJECT.SHAPE 1,INPUT$(LOF(3),3)
OBJECT.SHAPE 2,INPUT$(LOF(4),4)
OBJECT.SHAPE 3,INPUT$(LOF(5),5)
```

Close.Em.All:

```
CLOSE 3
CLOSE 4
CLOSE 5
```

Place.Sprites:

```
OBJECT.X 1,570
OBJECT.Y 1,15
OBJECT.X 2,220
OBJECT.Y 2,100
OBJECT.X 3,340
OBJECT.Y 3,160
```

Turn.On:

```
OBJECT.ON
```

Notice that the sprites stay on your screen, even when you list your program. To turn them off, just place the following line at the end of the program. As soon as you hit any key, they'll all disappear:

Off.Em:

```
OBJECT.OFF
```

To turn off selected sprites, place their number after the OBJECT.OFF statement.

## **MOVING SPRITES AND BOBS**

---

Sprites and bobs are not moved in the same way as we have been moving other graphics. Two sets of statements control the direction, velocity and acceleration variables of sprites and bobs.

**OBJECT.VX/VY.** Velocity refers to the number of pixels a sprite or bob moves in a given amount of time. A positive X value moves to the

right and a negative X value moves to the left. Positive Y moves down, and negative Y moves up. The statements:

```
OBJECT.VX 1,0 *No horizontal movement
OBJECT.VY 1,-100 *Negative Y moves up
```

would move an object straight up. This would be good for our rocket; so let's use it in a program:

Go.Sprite:

```
OPEN "Rocket.Sprite" FOR INPUT AS 1
OBJECT.SHAPE 1,INPUT$(LOF(1),1)
CLOSE 1
```

Place.Sprite:

```
OBJECT.X 1,320
OBJECT.Y 1,100
```

Sprite.Speed:

```
OBJECT.VX 1,0
OBJECT.VY 1,-100
```

Turn.On:

```
OBJECT.ON
OBJECT.START 1
```

```
WHILE 1
```

```
  *Forever loop - click stop to exit
```

```
  SLEEP
```

```
WEND
```

The line OBJECT.START 1 begins the sprite movement. OBJECT STOP 1 has the opposite effect.

**OBJECT.AX/AY.** *Acceleration* refers to the number of pixels a sprite or bob moves at an increasing velocity. A positive X value accelerates to the right and a negative X value to the left. Positive Y accelerates down, and negative Y moves up. The following statements can be used:

```
OBJECT.AX 1,10 'Slight horizontal movement
OBJECT.AY 1,-10 'Negative Y moves up
```

By slightly changing our last program, we can test these statements:

```
Speedup.Sprite:
  OPEN "Rocket.Sprite" FOR INPUT AS 1
  OBJECT.SHAPE 1,INPUT$(LOF(1),1)
  CLOSE 1
```

```
Place.Sprite:
  OBJECT.X 1,320
  OBJECT.Y 1,100
```

```
Sprite.Acceleration:
  OBJECT.AX 1,0
  OBJECT.AY 1,-10
```

```
Turn.On:
  OBJECT.ON
  OBJECT.START 1
```

```
WHILE 1
  SLEEP
WEND
```

Experiment *a lot* with the velocity, acceleration and direction in the above two programs. Once you can move your sprites in any direction you want, then go on to the next sections.

## MANAGING CRASHES WITH COLLISION

---

When your sprite smacks into the border, it would be nice if you knew so that you could reverse it or do something other than have it just protruding from the side. The COLLISION statement is a special event-handling routine that does just that. Like MENU, you can set up a subroutine handler using ON COLLISION GOSUB and have it initiated anywhere in the program by using a COLLISION ON statement. In the

following program, if you collide with the border, the subroutine stops the program and pronounces the condition (Crash!) at the top of the screen:

```
Collide.Sprite:
  ON COLLISION GOSUB Crash
  COLLISION ON
  OPEN "Rocket.Sprite" FOR INPUT AS 1
  OBJECT.SHAPE 1,INPUT$(LOF(1),1)
  CLOSE 1
Place.Sprite:
  OBJECT.X 1,320
  OBJECT.Y 1,100
Sprite.Acceleration:
  OBJECT.AX 1,0
  OBJECT.AY 1,-10
Turn.On:
  OBJECT.ON
  OBJECT.START 1

  WHILE Bang <> 1
    SLEEP
  WEND
END

Crash:
  PRINT "Crash!"
  Bang=1
  RETURN
```

You can also have the collision detection reverse the direction of your sprite or bob by “bouncing” it off the sides.

Besides hitting the sides, your sprites and bobs can also hit other objects. The default condition for colliding objects is for them to stop. With a collision routine, they can be made to bounce off one another, or they can be made to pass over one another. By using OBJECT.HIT, each object can be made to hit or pass through other objects and/or the border. For example, the following program has one sprite pass through another, but collide with the border, while the other sprite doesn't collide with anything:

Hit.Sprite:

```
ON COLLISION GOSUB Smack
COLLISION ON
OPEN "Rocket.Sprite" FOR INPUT AS 1
OPEN "Rocket.Sprite" FOR INPUT AS 2
OBJECT.SHAPE 1,INPUT$(LOF(1),1)
OBJECT.SHAPE 2,INPUT$(LOF(2),2)
CLOSE 1
CLOSE 2
```

Hit.Fix:

```
'Object 1 collides with nothing
OBJECT.HIT 1,0,0

'Object 2 collides with border
'but not Object 1
OBJECT.HIT 2,2,2
```

Place.Sprite:

```
OBJECT.X 1,320
OBJECT.Y 1,100
OBJECT.X 2,220
OBJECT.Y 2,100
```

Move.Ob2:

```
OBJECT.VX 2,40
OBJECT.VY 2,0
```

Turn.On:

```
OBJECT.ON
OBJECT.START 2
```

```
WHILE 1
SLEEP
WEND
```

Smack:

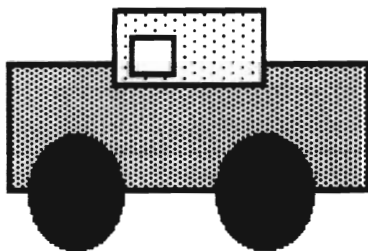
```
OBJECT.AX 1,0
OBJECT.AY 1,-20
OBJECT.START 1
RETURN
```



Using OBJECT.HIT is tricky since you have to use masks. Basically, the first parameter of OBJECT.HIT is the object number and the next two values define the masks. The second parameter is the "MeMask" and the third is the "HitMask." The setting of bits in these two masks determines whether the border and other objects will collide or pass through one another. Experiment with the values of the two masks to see what happens. For a really detailed discussion of this process, see pages 2-159 through 2-163 of the ROM KERNAL MANUAL VOL. 1 (Published by Commodore, Inc.).

## MAKING BOBS

Since we've covered most of the statements for dealing with sprites and bobs, making and moving bobs will be simple, but there are some differences. First, run the "ObjEdit" program, but this time choose "Bobs" instead of "Sprites." When you get your canvas, place the pointer in the lower left hand corner and pull it to the left and downward. Using this larger canvas, you can easily create large graphic objects to be animated. For example, the following shows a car bob created on the enlarged canvas (by a nonartist).



When you're finished, save the bob under the name, "Car.Bob" for use in the following program:

```
Get.Bob:
ON COLLISION GOSUB Stop.It
COLLISION ON
OPEN "Car.Bob" FOR INPUT AS 1
OBJECT.SHAPE 1, INPUT$(LOF(1),1)
CLOSE 1
```

```
Show.Bob:
  OBJECT.X 1,100
  OBJECT.Y 1,15

Look.No.Y:
  OBJECT.VX 1,30
  'There is no OBJECT.VY since it's going to be
  'zero anyway.

Turn.On:
  OBJECT.ON
  OBJECT.START

WHILE Halt <> 1
SLEEP
WEND
END

Stop.It:
  BEEP
  Halt=1
  RETURN
```

With a more talented artist than this book's author (which includes most people), you can do a lot with bobs. Bobs are slower than sprites and not as smooth in animation. However, they are easy to create, and you can use a full set of colors with them. The screen depth of sprites is limited to 2, while any screen depth can be used with bobs.

## SUMMARY

---

We've spent a lot of time with graphics in these last two chapters, but we've only begun to explore what you can accomplish graphically on your Amiga. There are a lot of graphic statements to keep straight, but in the long run, they make it a lot easier to work with and get the most out of your Amiga's graphic capabilities.

Moving graphics with the mouse or through automatic animation routines involves simple, fundamental algorithms. All movement is on an X,Y axis, and by increasing or decreasing those values the graphic figure

appears to move, whether it is because of the position of the mouse or a value generated in a loop. As we saw in the last chapter, though, the basic structures of sequence, branch and loop are used to create graphics just as they are used to accomplish any other programming task. In the next chapter, we will continue with those structures, but instead of examining graphics, we will be looking at sounds, music and voice synthesizing in programming your Amiga.



# Sound, Music and Voice Synthesizing

---

## AMIGA SOUNDS

---

Your Amiga has incredible sound generating capabilities. Most new microcomputers do, but Amiga BASIC capitalizes on the wide variety of sounds that can be generated by having special sound statements.

We will first examine how to make simple and complex sounds on your computer. Next, we'll see how to incorporate these sounds into routines and how to control them. Then, we'll look at how to use these words to make music and musical instrument sounds. Finally, we'll show you how you can incorporate the built-in voice synthesizer into your programs with the special words available in Amiga BASIC.

### The Sound Statement

The sound statement on your Amiga has four parameters, namely:

1. Frequency: 20–15000 (measured in hertz). The higher the frequency, the higher the pitch of the sound.
2. Duration: 0–77. This is how long the sound is held.

3. Volume: 0-255. From a default of 127, the sound can be turned up or down.
4. Voice: 0,3 and 1,2. Without speakers, voices 0 and 3 work off the speakers in the monitor. 0,3 = left speaker and 1, 2 = right speaker.

To see the range of these parameters and experience what they sound like, turn up your sound and enter the following program:

```
Sound.Tester:
WHILE F <> 5 'A value of 5 for frequency exits.
  INPUT "Frequency 20-15000 ";F
  INPUT "Duration 0-77";D
  INPUT "Volume 0-255";V
  INPUT "Voice 0-3 ";A
  SOUND F,D,V,A
WEND
```

You created a lot of racket with that. Now let's use another word that will give you even more control over the sound.

The WAVE statement works with a special 256 element array. (The array is something like the one used with PUT/GET graphics in the last chapters.) The form of the wave can be virtually any pattern that can fit into the array, with element values from -128 to 127. The statement has only two parameters, voice (0,3,1,2) and the array that defines the wave. It is possible to use the SIN (sine) function to define the wave pattern instead of creating your own wave. To get started, let's make a simple program that will provide an example of how to build a wave array:

```
DIM Timbre%(255)
FOR X= -128 to 127
  Timbre%(C)=X
  C=C+1
NEXT X

WAVE 0, Timbre%
SOUND 200,30,127,0
```

The wave form built by the array is a straight single line from -128 to +127. Now, change the WAVE statement in the above program to the following:

```
WAVE 0,SIN
```

This time when you run the program, there is a different sound even though the SOUND parameters were not changed. You have a lot of control over the way your sounds come out, and with all of the possible combinations, you can make just about any sound you want. You can even have your Amiga generate multiple voices simultaneously. We'll just use the 0 and 3 voice in case you only have the monitor set up, but for those of you who are real music and sound buffs, there's a lot more you can do by expanding on what we will discuss in this chapter.

To get sounds to come out at the same time, use two different voices in two SOUND statements and the SOUND WAIT and SOUND RESUME statements. These two statements wait until all of the sound information is ready to play, and then release all the voices at once in any SOUND statements between SOUND WAIT and SOUND RESUME. For example, using two different WAVE definitions and two different voices, the following program lets you further experiment with the different sound parameters:

```
Sound.Waiter:
GOSUB Timber
WHILE F <> 5
  INPUT "Frequency 20-15000 ";F
  INPUT "Duration 0-77";D
  INPUT "Volume 0-255";V
  WAVE 0,Timbre%
  WAVE 3,SIN
  SOUND WAIT
  SOUND F,D,V,0 : REM Voice Zero
  SOUND F,D,V,3 : REM Voice Three
  SOUND RESUME
WEND
END
```

```
Timber:
DIM Timbre%(255)
FOR Y=1 TO 2
  FOR X=0 TO 126
    C=C+1
    Timbre%(C)=X
  NEXT X
NEXT Y
RETURN
```

Thus, besides having control over the sound and wave parameters, you can mix multiple sounds and waves on different voices for even more sounds. The best thing to do is to experiment and see what comes up.

## AMIGA MUSIC

---

Since Amiga BASIC gives you so much control over the sounds, making music should not be too difficult. Simple music is easy to create with some organization and programming care. We'll make up a set of basic musical notes and a few song programs to see how it is done. More advanced and sophisticated programs are left to your own resources and imagination.

First, let's make up a set of notes. We'll start with a low octave, and progressively multiply the note values by powers of two to make a set of four octaves. In other words after defining seven basic notes in an array and by using the note name for the array name, we'll let your Amiga figure out the rest and then play it for you. (Notice we do not dimension the array since no array goes over four elements.)

Notes:

```
C(1)=130.81
D(1)=146.83
E(1)=164.81
F(1)=174.61
G(1)=196
A(1)=220
B(1)=246.94
```

Three.More.Octaves:

```
FOR X=1 TO 3
  C(X+1)=C(1) * (2 ^ X)
  D(X+1)=D(1) * (2 ^ X)
  E(X+1)=E(1) * (2 ^ X)
  F(X+1)=F(1) * (2 ^ X)
  G(X+1)=G(1) * (2 ^ X)
  A(X+1)=A(1) * (2 ^ X)
```



```
B(X+1)=B(1) * (2 ^ X)
NEXT X

Play.Notes:
FOR Octave=1 TO 4
  SOUND C(Octave),10,127,0
  SOUND D(Octave),10,127,0
  SOUND E(Octave),10,127,0
  SOUND F(Octave),10,127,0
  SOUND G(Octave),10,127,0
  SOUND A(Octave),10,127,0
  SOUND B(Octave),10,127,0
NEXT
```

Now that all of the notes are in an array, it shouldn't be too difficult to create a program that allows you to enter notes from sheet music. For this basic program, we'll forego sharps, flats and dotted notes. Use the following plan:

1. Put the notes of four octaves in an array.
2. Input the notes/octave in a single variable using the format: Note-octave (e.g., C4, A2).
3. Input the note durations as (W)hole, (H)alf, (Q)uarter, (E)ighth and (S)ixteenth notes.
4. After all values are in an array, play the song.

```
SongWriter:
Notes:
C(1)=130.81
D(1)=146.83
E(1)=164.81
F(1)=174.61
G(1)=196
A(1)=220
B(1)=246.94

FOR X=1 TO 3
  C(X+1)=C(1) * (2 ^ X)
  D(X+1)=D(1) * (2 ^ X)
```

```

E(X+1)=E(1) * (2 ^ X)
F(X+1)=F(1) * (2 ^ X)
G(X+1)=G(1) * (2 ^ X)
A(X+1)=A(1) * (2 ^ X)
B(X+1)=B(1) * (2 ^ X)
NEXT X

```

Enter.Notes:

```

INPUT "How many notes ";N%
DIM Note(N%)
DIM Dur(N%)
Wnote=20: 'Set tempo with whole note
FOR X= 1 TO N%
  PRINT "Note #";X;
  INPUT Note$
  Note$=UCASE$(Note$)
  Octave$=RIGHT$(Note$,1)
  Octave=VAL(Octave$)
  Snote$=LEFT$(Note$,1)
  IF Snote$="C" THEN Note(X)=C(Octave)
  IF Snote$="D" THEN Note(X)=D(Octave)
  IF Snote$="E" THEN Note(X)=E(Octave)
  IF Snote$="F" THEN Note(X)=F(Octave)
  IF Snote$="G" THEN Note(X)=G(Octave)
  IF Snote$="A" THEN Note(X)=A(Octave)
  IF Snote$="B" THEN Note(X)=B(Octave)

```

Note.Duration:

```

PRINT "Duration (W,H,Q,E,S) of note #";X;
INPUT Dur$
Dur$=UCASE$(Dur$)
IF Dur$="W" THEN Dur(X)=Wnote: 'Whole
IF Dur$="H" THEN Dur(X)=Wnote/2: 'Half
IF Dur$="Q" THEN Dur(X)=Wnote/4: 'Quarter
IF Dur$="E" THEN Dur(X)=Wnote/8: 'Eighth
IF Dur$="S" THEN Dur(X)=Wnote/16: 'Sixteenth
NEXT X

```

Play.It.Amiga:

```

FOR X=1 TO N%
  SOUND Note(X),Dur(X),127,0
NEXT X

```

Because we're using a single volume and voice, they are constant. If you wanted, they could be varied as well. Likewise, you could set up a routine to find sharps, flats and dotted notes.

## Song Reader

If you spend time creating a song, you might as well have a way to save it so you can play it whenever you want rather than having to enter all the notes and durations each time you run the program. By making some changes in the above program, we can have a program that will store music in DATA statements. Look at the following program with some lines from Cole Porter's *Anything Goes*. (Don't rewrite the whole program! Use your editor to make just the necessary changes from the last program.)

Song.Reader:

Notes:

C(1)=130.81

D(1)=146.83

E(1)=164.81

F(1)=174.61

G(1)=196

A(1)=220

B(1)=246.94

FOR X=1 TO 3

C(X+1)=C(1) \* (2 ^ X)

D(X+1)=D(1) \* (2 ^ X)

E(X+1)=E(1) \* (2 ^ X)

F(X+1)=F(1) \* (2 ^ X)

G(X+1)=G(1) \* (2 ^ X)

A(X+1)=A(1) \* (2 ^ X)

B(X+1)=B(1) \* (2 ^ X)

NEXT X

Read.Notes:

N%=33

DIM Dur(N%)

DIM Note(N%)

Wnote=20: 'Tempo

FOR X= 1 TO N%

```

READ Note$
Note$=UCASE$(Note$)
Octave$=RIGHT$(Note$,1)
Octave=VAL(Octave$)
Snote$=LEFT$(Note$,1)
IF Snote$="C" THEN Note(X)=C(Octave)
IF Snote$="D" THEN Note(X)=D(Octave)
IF Snote$="E" THEN Note(X)=E(Octave)
IF Snote$="F" THEN Note(X)=F(Octave)
IF Snote$="G" THEN Note(X)=G(Octave)
IF Snote$="A" THEN Note(X)=A(Octave)
IF Snote$="B" THEN Note(X)=B(Octave)

```

Note.Duration:

```

READ Dur$
Dur$=UCASE$(Dur$)
IF Dur$="W" THEN Dur(X)=Wnote
IF Dur$="H" THEN Dur(X)=Wnote/2
IF Dur$="Q" THEN Dur(X)=Wnote/4
IF Dur$="E" THEN Dur(X)=Wnote/8
IF Dur$="S" THEN Dur(X)=Wnote/16

```

```

NEXT X
FOR X=1 TO N%
SOUND Note(X),Dur(X),127,0
NEXT

```

Anything.Goes:

```

'ANYTHING GOES
'©1934 (Renewed) WARNER BROS. INC.
'All Rights Reserved
'Used by Permission

```

```

DATA G3,Q,G3,Q,A3,Q,E3,Q,G3,Q,A3,Q,G3,E,A3,E
DATA A3,E,G3,E,E3,Q,G3,E,A3,Q,E3,E,G3,Q,A3,E
DATA C4,E,C4,E,A3,E,C4,Q,D4,Q,E4,Q,D4,W,D4,Q
DATA 00,Q,C4,E,00,S,C4,E,00,S,C4,Q,00,S,C4,W,C4,Q

```

All of the DATA statements are alternating notes and duration values. The 00 “notes” are actually pauses, either at rests or between identical notes for separation.

---

## MORE WAVE WORK

---

Let's do a little more work with WAVE. The following example illustrates a sawtooth-form wave: Look at the routine for creating the "Timbre%" array:

Sawtooth.Wave:

```
CLEAR
GOSUB Timber
F=500
D=10
V=127
A=0
WAVE 0,Timbre%
SOUND F/4,D,V,A
SOUND F/2,D,V,A
SOUND F,D,V,A
SOUND 2*F,D,V,A
END
```

Timber:

```
DIM Timbre%(255)
'Sawtooth
FOR X=0 TO 42
  Timbre%(C)=X
  C=C+1
NEXT X
FOR X=-42 TO 42
  Timbre%(C)=X
  C=C+1
NEXT X
FOR X=-42 TO 42
  Timbre%(C)=X
  C=C+1
NEXT X
FOR X= -42 TO 0
  Timbre%(C)=X
  C=C+1
NEXT X
PRINT C : REM Check Array Size
RETURN
```

The same fundamental routine is used to create the pulse wave illustrated below, but it was set up with the intention of changing the pulse width for generating different wave forms and sounds:

Pulse.Wave:

```
CLEAR
GOSUB Timber
F=500
D=10
V=127
A=0
WAVE 0, Timbre%
SOUND F/4,D,V,A
SOUND F/2,D,V,A
SOUND F,D,V,A
SOUND 2*F,D,V,A
END
```

Timber:

```
DIM Timbre%(255)
'Pulse
PH=90:'Pulse height
PW=70:'Pulse width

FOR X=1 TO 3
  FOR Y=1 TO PW
    Timbre%(C)=PH
    C=C+1
  NEXT Y
  FOR Z=1 TO 10
    Timbre%(C)=-PH
    C=C+1
  NEXT Z
NEXT X
K=255-C
FOR R=C TO C+K
  Timbre%(R)=PH
NEXT R
PRINT R : REM Check array size. Be careful setting
  ' the PH and PW variables or the array will overflow
RETURN
```

You can create different instrumental sounds, sound effects, noises and just about any other audible output by working with the various sound statements. It's important to keep notes on various sounds you discover and to systematically work with information you get from your programs.

## SPEECH SYNTHESIS

---

There are two basic ways to do voice synthesizing on your Amiga: the easy way and the hard way. We'll look at the easy way first, using TRANSLATE\$, so that you can get off to a running start with voice synthesizing. Then we'll see how to make a bridge between the built-in translating capabilities and give you options using an array you build yourself. Finally, we'll examine "the hard way" to do voice synthesizing. This lets you do all sorts of things with your Amiga's voice so that it will say what you want, the way you want it said.

### Translating Text to Talk

The first thing to try is using the statement SAY with TRANSLATE\$. SAY works something like the PRINT statement, except instead of the output being shown on your screen, it is sent to the built-in voice synthesizer. When using regular text (and regularly spelled text), the TRANSLATE\$ function is required. The general format is:

```
SAY TRANSLATE$ ("MESSAGE")
```

or

```
SAY TRANSLATE$ (MESSAGE$)
```

To get started, try the following little programs (substitute your name for "Sam"):

```
First.Talk:
```

```
One.Way:
```

```
  SAY TRANSLATE$ ("Feed me silicone")
```

```
Another:
```

```
  Talk$= "And I will talk to you"
```

```
  SAY TRANSLATE$ (Talk$)
```

```

Still.Another:
  Talk2$ = TRANSLATE$ ("You can start now" )
  SAY TALK2$
END

Talk.To.Me:
  Message$= "Hello. My name is Amiga and your name"
  Message2$ = "is Sam. I know that because you told"
  Message3$ = "me so."
  BigMessage$= Message$ + Message2$ + Message3$
  SAY TRANSLATE$ (BigMessage$)
END

```

As you can see from the examples, there are different ways you can format the SAY TRANSLATE\$ sequence. There is no inherent advantage of one way over another, so use whatever method is most appropriate to your programming needs.

## The SAY Array

To control various parts of the speech presentation, you can use the optional integer "mode-array." This array has nine parameters to change the sound of the speech. We'll glance at what each one does and then look at examples of what you can do with them:

Number	Control	Range and Function
0	Pitch	65–320 Voice pitch in hertz (110 = default)
1	Inflection	0–1 Modulation—Inflection (0), Monotone (1)
2	Rate	40–400 Speech rate
3	Gender	0–1—Male (0), Female (1)
4	Tune	5000–28000 from Low to High
5	Volume	0–64 (64 = default)
6	Channel	See below (10 = default)
	<i>Value</i>	<i>Channel(s)</i>
	0	0
	1	1
	2	2
	3	3
	4	0 + 1
	5	0 + 2



Number	Control	Range and Function
	<i>Value</i>	<i>Channel(s)</i>
	6	1 + 3
	7	2 + 3
	8	Either available left channel
	9	Either available right channel
	10	Either available left/right pair of channels
	11	Any available single channel
7	Mode	0-1—Synchronous (0), Asynchronous (1)
8	Control	0-2 (with Asynchronous mode only)—Normal (0), Stop speech (1), Override (2)

To work with the parameters of the array, each value is read into a nine-element integer array. The array is then placed at the end of a voice command sequence, as:

```
SAY TRANSLATE$ (TEXT$), ARRAY%
```

The following program illustrates using an array with the SAY statement:

```
Wake.Up:
FOR X=0 TO 8
  READ V
  TALK%(X)=V
NEXT
```

```
DATA 200      :'Pitch - higher than default
DATA 0       :'Inflection
DATA 145     :'Rate - slower than default
DATA 1       :'Gender - female
DATA 24000   :'Higher voice
DATA 64      :'Default volume
DATA 10     :'Either available pair of speakers
DATA 0      :'Default mode
DATA 0      :'Default control
```

```
Message1:
TEXT$=TRANSLATE$ ("GOOD MORNING BILL")
SAY (TEXT$),TALK%
Message2:
TEXT2$="How are you today"
SAY TRANSLATE$ (TEXT2$), TALK%
```

Normally, you'd want to put all of the DATA in a single line, but it was arranged vertically in this case to more clearly show all of the element values that make up the array TALK%. Change the different parameters in the preceding program to get your Amiga to sound just the way you want. Change the rate of speech to about 150 (the default), and then experiment with the pitch, gender and tuning to see what you can create.

## Writing Like It Sounds: Phonetic Transcription

Now we'll start doing speech synthesis the hard way. Since all of the work we'll be doing requires that phonetic spellings be in caps, put your CAPS LOCK key on. Read *Appendix H* in your Amiga BASIC Manual carefully, and forget everything you ever learned about spelling. Your built-in speech synthesizer (called Narrator) will only recognize words spelled phonetically. If you want it to work correctly, you have to spell your messages with the phonetic spellings we'll examine in this section.

Here's a list of things to keep in mind when you work with phonetic transcription:

1. Keep CAPS LOCK on.
2. All vowels (a, e, i, o, u) have two letters.
3. Use only phonetic spellings.
4. Do not use TRANSLATE\$.

The general format is:

```
SAY "PHONETIC SPELLING"
```

You can place the text in a string, concatenate and do everything else that can be done with strings. However, all spellings must be in the special Narrator alphabet. For example, to say "hello," do not enter this:

```
SAY "HELLO"
```

That would result in an ‘illegal function call’ error. Instead, write:

```
SAY "/HEHLOH"
```

To see the difference between using TRANSLATE\$ and the phonetic spellings, run the following little program:

```
SAY TRANSLATE$ ("Hello")
SAY "/HEHLOH"
```

The first hello sounded more like “halo,” while the second “hello” sounded more like what we expect to hear when the word is spoken. That is the advantage of using phonetic spellings. While it is a lot more difficult than using TRANSLATE\$ and the normal spelling of words, the phonetic method lets you adjust the Narrator’s speech to a more natural sound.

## Narrator’s Phonetic Alphabet

The following alphabet is to be used with the SAY statement when not using TRANSLATE\$:

Phoneme	Sound	Phoneme	Sound	Phoneme	Sound
<i>Vowels</i>					
IY	feet	ER	fir	AH	thunder
EH	set	AX	around	UH	hook
AA	lot	IH	sit	OH	order
AO	walk	AE	fan	IX	slid
<i>Consonants</i>					
R	row	/H	hunt	TH	thick
W	way	B	bun	ZH	assure
M	money	D	dig	DH	there
NX	ding	K	come	J	jade
S	soul	L	hello	/C	loch
F	fool	Y	yet	P	pull
Z	was	N	in	T	top
V	vest	SH	mush	G	gale
CH	chow				

		Diphthongs			
EY	shade	OW	blow	AW	power
OY	soil	AY	side	UW	shrew
		Special			
DX (tongue flap)	city	Q	mitt_en	RX	star
QX (silent vowel)	made	LX	fall		
UL	=AXL	UM	=AXM	UN	=AXN
IL	=IXL	IM	=IXM	IN	=IXN
Characters		Function			
1-9		Stress range on syllable			
.		Last character of sentence			
?		Last character of sentence			
-		Phrase delimiter			
,		Clause delimiter (comma)			
()		Noun phrase delimiters			

The above phonemes and characters (all in CAPS) make up the entries that Narrator can understand after a SAY statement (without a TRANSLATE\$ statement). Like everything else we've discussed, experimentation is the key to getting things the way you want them, especially when it comes to the subjective sounds your Amiga can create. However, some examples will help, and we do need to say a few things about stress and intonation before you take off on your own.

**STRESS.** By placing a number between one and nine (1-9) after the vowel in any syllable you want stressed, Narrator will emphasize that syllable. For example, in the sentence "I want you," stress can be placed on any one of the three words. A lover would stress the *want*, a choice would stress *you* and a desire would stress the *I*. Try the following three examples to see the difference in sound depending on where the stress is placed:

```
SAY "AYS WAONT YUHUW" : 'Stress on I
SAY "AY WAOSNT YUHUW" : 'Stress on want
SAY "AY WAONT YUH5UW" : 'Stress on you
```

Besides placing emphasis on syllables, stress values also change the sound of words. If a word doesn't sound right after you've experimented with different phoneme combinations, try changing the stress placement to see if that helps.

**STRESS MARK VALUES.** The following is a list of suggested stress mark values:

<u>Speech Part</u>	<u>Value</u>
Nouns	5
Pronouns	3
Verbs	4
Adjectives	5
Adverbs	7
Quantifiers	7
Exclamations	9
Articles	0
Prepositions	0
Conjunctions	0
Secondary stress	1 or 2

Another form of stress is intonation. Placing a value after the stress value gives a word its intonation value. Depending on how you want the statement to sound, more or less should be placed after the stress values. Take a look at the following example to see how this works:

```
SAY "IH4Z AENIY5WAAN /HAAM?"
'Is anyone home
```

The verb 'is' has a stress of 4 following the first phoneme vowel, while the noun 'anyone' is stressed on the second syllable with a 5. In reading this question, it would look like:

Is anyone home?

To add intonation, you would simply place a second number next to the first stress values. For example:

```
SAY "IH4Z AENIY57WAAN /HAAM?"
```

places an intonation value of 7 after the stress value of 5 in "anyone." Change the value and listen carefully to how it sounds with different intonation values.

In addition to using all of the phoneme alphabet with SAY, you can also include the array to change the mode parameters. By using the phoneme method combined with an array, you can have your "AH-MIY5GAH" talk up a storm. See if you can develop accents and other styles of speech to give your computer a vocal personality.

## SUMMARY

---

Like the previous two chapters on graphics, we spent a good deal of time just to get the fundamentals of sound across. Your Amiga is so powerful and complex that only by experimenting with your BASIC statements can you really begin to appreciate the capabilities it has. By changing the various values in the sound parameters, you can create anything from sound effects to a symphony. The Narrator speech synthesizer gives your Amiga a voice as well as the ability to produce a multitude of sounds. (In fact, if you worked with the SOUND statement enough, you could probably make your own speech synthesizer!)

The key to using new statements, especially unique ones like SAY, is to begin incorporating them into your programs. Instead of having a text prompt appear on the screen, send the output to the speaker and create a *talking* prompt. In the rest of this book, we'll use our newer statements to liven up the programs and illustrate how you can use them to enhance your programs.

# The Disk System and Sequential Files

In this chapter we are going to learn more about working with the disk system and creating data files. You will find out how to link program files and do other work with program files. The functions in this chapter have many practical applications, and we will attempt to use generic examples that you can customize for your own use. We are going to make simple sequential text file programs. These files are very useful for storing information you have entered; rather than having to reenter the data all over again, you simply OPEN a file and INPUT # the data. To show how these files work, we will make a simple program for keeping and updating names, cities and states.

One of the most useful applications of BASIC programs saved in ASCII format is to save frequently used subroutines and then use the MERGE statement to create a program from existing subroutines. You have to make sure your subroutines all have unique sets of line labels, but otherwise, a set of subroutines saved as ASCII files can save a lot of unnecessary programming. For example, SAVE the following two programs as ASCII files:

```
CLS
INPUT "How many names"; N%
FOR X = 1 TO N%
```

```
    INPUT "Name ";ID$
    GOSUB Mid.Screen
NEXT
END
```

Now type in:

```
SAVE "Namer",A<RETURN>
```

After you have done that, enter NEW and do the next program:

```
Mid.Screen:
'Routine for centering text
L = 31 - LEN(ID$)/2
PRINT TAB(L)ID$
RETURN
```

Next enter:

```
SAVE "Mid.Screen",A <RETURN>
```

Now you have two programs saved as ASCII files. Neither program will work by itself. If you tried to use them as regular BASIC program files, as soon as you loaded the second one, the first one would be wiped out. However, with ASCII files and the MERGE command, you can load them separately. Once they are both loaded, you can RUN the combined program, and if you want, you can even SAVE it as a BASIC program or ASCII file. Key in the following sequence:

```
LOAD "Namer" <RETURN>
MERGE "Mid.Screen"<RETURN>
RUN <RETURN>
```

At this point everything should work just fine. Now enter:

```
SAVE "Mid.Name"<RETURN>
```

You now have a BASIC file that is made up of two ASCII files. As you collect useful subroutines, keep a record of their line number ranges. You can thus write a program simply by MERGEing several subroutines.



---

## CHAIN ROUTINES

---

Another way to save and use program chunks is with the CHAIN statement. Unlike MERGE, which brings two programs together, the CHAIN statement allows you to use only parts of a program at a time. In other words, one program could be used as an input buffer, for example, and then it would be erased from memory and a second part could be used as an output routine. The variables and arrays can be optionally passed from the first to the second part with either an ALL or COMMON statement. If ALL is used, then all of the variables and arrays are passed. If COMMON is used, then only those specified are used. Let's make two very standard routines to show how CHAIN works. First, we'll make a name and address input buffer that puts keyboard information into an array:

```
Input.Names.Addresses:
  INPUT "How many names";N%
  DIM Names$(N%),Address$(N%),City$(N%)
  DIM State$(N%),ZIP$(N%)
  REM *****
  REM Input Buffer
  REM *****
  FOR X=1 TO N%
    INPUT "Name"; Names$(X)
    INPUT "Address "; Address$(X)
    INPUT "City "; City$(X)
    INPUT "State "; State$(X)
    INPUT "Zip Code "; ZIP$(X)
  PRINT
  NEXT X

  CHAIN "NameAd.Output" ,, ALL
```

That can be used with just about any program we want to enter names and addresses. (Later we'll use it for our sequential file program.) Notice at the bottom on the program, the CHAIN statement specifies another file called "NameAd.Output" and has ALL preceded by two commas. (We'll see what option goes between the two commas further on.) All five arrays and the integer variable N%, along with their contents,

are passed to the program called "NameAd.Output." Let's see what it looks like:

```
Name.Address.Output:
  REM *****
  REM Output Buffer to Screen
  REM *****
  CLS
  FOR X=1 TO N%
    PRINT "Name: "; Names$(X)
    PRINT "Address: "; Address$(X)
    PRINT "City, State, Zip: "; City$(X);", ";
    PRINT State$(X);SPACE$(1);ZIP$(X)
    PRINT
  NEXT X
```

This is another frequently used routine for printing names and addresses in a format often used for addressing a letter. (We have left the headers, 'Name:', etc. for illustration purposes.) Notice that the output routine uses a different format to send information to the screen than that used for entering the data.

In our example, we needed to use all of the arrays and variables in the first program. However, there may be occasions where you will need only some of the values to be passed. In this case you would specify their "passage" with COMMON, and ALL would be omitted from the CHAIN statement sequence. Here are a couple of short routines that illustrate the use of COMMON with selected variables and arrays:

```
Get.Seal:
  FOR X=1 TO 10
    Ball$(X)=STR$(X)
  NEXT
  N=2
  FOR X=1 TO 10 STEP N
    Seal$(X)="Seal" + Ball$(X)
  NEXT
  COMMON Seal(),N
  CHAIN "See.Seal"
```

Save the above program as "Get.Seal."

```
See.Seal:
  FOR X=1 TO 10 STEP N
    PRINT Seal$(X)
  NEXT
```

Save above program as "See.Seal."

Notice that we generated a set of numbers with the array 'Ball\$', but we did not need it in the program we chained to the first one after 'Seal\$' was defined. However, we did need the variable 'N' and the array 'Seal\$' which were passed with COMMON. Had we used ALL in the first program, we would have passed the unnecessary array 'Ball\$.'

Also, take note of the format for passing arrays. There is no need to specify the dimension, only the array name followed by the double parentheses—'()'. Variables are simply passed using the variables' name as in the original program.

It is also possible to use CHAIN with an option to specify the first line to be executed in the chained on program. Unfortunately, line labels *cannot* be used. For example, these two routines show how the first program chains the second beginning at line 30:

```
First.One:
  PRINT "This is from the first program"
  Chain "Show", 30
```

Save as "One."

```
No.Show:
  PRINT "This is not to be seen"
30 PRINT "Show this"
More.Show:
  PRINT "This can also been seen"
```

Save as "Show."

When you run the first program, it will chain the second, but it will not execute the first PRINT statement since it is *before* line 30 in the program. This option may not be used right away, but when you have an application where you need to begin in the middle of a routine, it can be useful.

### Amiga Notes

#### Changing Drives and Drawers

When you start writing a lot of programs and creating several files, you will want to organize everything in different “drawers” on your disk. To open a file in a specific drawer, you have to enter the name of the drawer, a slash (/) and the name of the file. In some cases, you may even have to specify disk drive drawer and file name.

Creating drawers within drawers can really make it difficult for you to locate a specific file. If you want to save everything in a single drawer or even a subdrawer, and get everything out of the same subsection on your disk, it's a lot easier to use CHDIR to make the drawer the “current” or default file directory. To do that, just enter:

```
CHDIR "df#:drawer"
```

where '#' is the drive number (usually 0 or 1), and 'drawer' is the name of the subdirectory. To check if you're in the correct subdirectory, catalog your disk from BASIC with FILES; if all you see are files in your selected subdirectory, you know you did it correctly.

---

## SEQUENTIAL FILES

---

To get started working with sequential files, think of them as another place for *outputting* information instead of the screen, and another place for *inputting* information instead of the keyboard. Unless we save information in DATA statements, we really have no other way of preserving material we create with our programs.

### From Buffer to Disk and Disk to Buffer

The key to understanding files is to understand the concept of a “buffer.” A buffer is not a file; it is the “bucket” that you use to “pour” information into a disk file and into which you temporarily store information from a disk file. Basically, you will use the following sequence in file work:

```
Keyboard Input -> Buffer  
Buffer Output -> Disk File  
Disk Input -> Buffer  
Buffer Output -> Screen
```

Arrays and variables are buffers. It's as simple as that. Enter data, and it goes into a variable or array. That's the buffer. Write the information in the array or variable to the disk, and that's buffer output to the disk. It is important to keep the buffer concept that simple. Novice programmers tend to overcomplicate these things, making it unnecessarily difficult for themselves.

## MAKING A SEQUENTIAL FILE

---

Our first step will be to write a program that creates a sequential file. To create a file, use:

```
OPEN "Filename" FOR OUTPUT AS #1
```

That opens a new file identified as #1. Any number will do. Next, to write the buffer to the disk, enter:

```
PRINT #1, (or PRINT#1, USING or WRITE#1,)
```

The number must match the file number used to create the sequential file.

When all of the buffer has been written to the disk, close it with:

```
CLOSE #1
```

If no file number is specified, all files will be closed. Sometimes you will need more than a single file open at the same time. In those cases, you will only want to close the opened files and maybe not all of them.

The above simple statements cover everything we need in sequential files so we can begin using them. Let's write a creation program for a file. We'll make one for a nursery business that will hold the names of plants that customers have requested information about.

```

Plant.File:
  CLS
  LOCATE 10,1
  INPUT "How many requests";N%
  DIM Plant$(N%)
  CLS
  REM *****
  REM Buffer
  REM *****
  FOR X = 1 TO N%
    INPUT Plant$(X)
  NEXT X
  REM *****
  REM Put Buffer into Sequential File
  REM *****
  OPEN "Plants" FOR OUTPUT AS #1
  FOR X = 1 TO N%
    PRINT #1, Plant$(X)
  NEXT X
  CLOSE #1
  END

```

After you enter the program, run it; remember the number of plants that you entered. Save the program under the name "PlantEntry," as we will come back to it later. Enter FILES from BASIC to make sure there is a file called 'Plants' that was created by our "PlantEntry" program.

The next step is to read our files, using:

```
OPEN "FILENAME" FOR INPUT AS #1
```

The only difference is that the information is now going to be *input* instead of *output*. It is input from the disk into a buffer. Thus, instead of PRINT#, use:

```
INPUT#1, (or LINE INPUT#1)
```

The buffer used for input can be a variable or an array. It does not have to be the same array or variable name used for entering the information into the file. In fact, it is not even necessary to use the same type of buffer. If an array buffer placed the information into the file, a variable buffer can take it out.

Next, you will need something to see if all of the information is out of the file. To do that, use the EOF (end of file) function. The fundamental form is:

```
IF EOF(1) = 1 THEN end of file has been reached
```

The loop:

```
WHILE NOT EOF(1)
  Read file . . . . .,etc.
WEND
Close
```

is a fairly standard way of reading all of the contents of a file.

The following program will OPEN 'Plants,' INPUT the file, check EOF, CLOSE the file and then PRINT out the contents on the screen. Notice the similarities and differences between this program and the one used to write files:

```
Plant.Read:
CLS
OPEN "Plants" FOR INPUT AS #1
REM
REM Put Sequential File into Buffer
REM
WHILE NOT EOF(1)
  INPUT #1,Plant$
  PRINT Plant$
WEND
CLOSE #1
END
```

Save this under the file name "Plant.Read." Using the EOF function eliminates the need to keep track of the number of files you entered. If there are more files, EOF(1) (with "1" being the file number, e.g., INPUT #1), then EOF(1) = 0. If EOF(1) = -1 then the program has found the End Of File. Using the WHILE/WEND statement, we check for the case where EOF is NOT true. When this condition is met, the program exits the loop. Notice that as soon as we INPUT# the data from the 'Plants' file we PRINTed it to the screen using the normal PRINT statement. Also notice

that we used a string variable buffer, "Plant\$," instead of a string array buffer.

Thus far, we have a program that will OUTPUT a list of names in a data file and one that will INPUT those names back to us. What happens, though, if we want to add some names to the file? Well, we could make a new file under another name, but a better way is to APPEND our current 'Plants' file. Using the APPEND statement, we write our additional file name(s) at the bottom of the data list in our existing file. It is important to remember that when using APPEND, there must be an existing file to which we can APPEND our data. To do that, use the following format:

```
OPEN "FILENAME" FOR APPEND AS #1
PRINT#1 (or PRINT#1 USING or WRITE#1)
CLOSE#1
```

We will need only a slightly different program than 'PlantEntry.' Simply load 'PlantEntry,' make the appropriate changes and save the program under the name 'PlantApnd':

```
Plant.Apnd:
  CLS
  LOCATE 10,1
  INPUT "How many requests to add";N%
  DIM Plant$(N%)
  CLS
  REM *****
  REM Buffer
  REM *****
  FOR X=1 TO N%
    INPUT Plant$(X)
  NEXT X
  REM *****
  REM Put Buffer into Sequential File
  REM *****
  OPEN "Plants" FOR APPEND AS #1
  FOR X=1 TO N%
    PRINT #1,Plant$(X)
  NEXT X
  CLOSE #1
  END
```



That didn't take much work, did it? All you had to do was change OUTPUT to APPEND; once you SAVE the program as 'PlantApnd,' you have programs that will OUTPUT, INPUT and APPEND sequential text files.

We've seen how to OUTPUT, APPEND and INPUT elements of a single file. However, since file names are essentially just strings, we could use variables to do a lot of the work automatically. Remember, if we can write one program that will do most of the work, then we can save a lot of time by writing several little programs. The following program, 'Ad.Book,' will create, append and read any text file you want. It handles names and addresses, but that can be changed if so desired.

### Amiga Notes

#### Amiga Short Cut #256K

The following program is relatively long, but certain parts of it are very similar to earlier ones. Therefore, rather than re-typing everything that is similar, it is much easier to use the CHAIN, MERGE and EDIT functions available on your Amiga. Using your cut and paste functions in the editor, change OUTPUT to APPEND and change the block labels as well. We'll be using our "general" name, address, etc. buffer that we introduced at the beginning of the chapter.

Address.Book:

Menu.Service:

```
MENU 1,0,1, "File Work"  
  MENU 1,1,1, "Create New Address Book"  
  MENU 1,2,1, "Append to Existing Book"  
  MENU 1,3,1, "Read An Address Book"
```

```
MENU 2,0,1, "<->" : REM Cover Menu Bar 2
```

```
  MENU 2,1,1, "Not in use"
```

```
MENU 3,0,1, "Exit"
```

```
MENU 3,1,1, "Quit the program"
```

```
MENU 4,0,1, "<->" : REM Cover Menu Bar 4
```

```
  MENU 4,1,1, "Not in use"
```

```
Clear.Me:
  Flag=0
  CLS
  PRINT "Press the RIGHT mouse button and choose a menu."

Menu.Read:
  REM Loop to scan menu choice.
  IF Flag THEN Clear.Me
  CHOOZEIT=MENU(0)
  ON CHOOZEIT GOSUB Make.It,Empty1,Terminate,Empty2
GOTO Menu.Read

Make.It:
  Flag=MENU(1)
  ON MENU(1) GOSUB Get.Buffer,Get.Buffer,Read.File
  RETURN

Empty1:
  RETURN

Terminate:
  MENU RESET
  CLS
  END

Empty2:
  RETURN

Get.Buffer:
  GOSUB See.Files
  INPUT "How many names";N%
  PRINT
  DIM Names$(N%),Address$(N%),City$(N%)
  DIM State$(N%),Zip$(N%)

  REM *****
  REM Input Buffer
  REM *****
  FOR X=1 TO N%
    INPUT "Name "; Names$(X)
    INPUT "Address "; Address$(X)
```

```
    INPUT "City "; City$(X)
    INPUT "State "; State$(X)
    INPUT "Zip Code "; Zip$(X)
    PRINT
NEXT X
IF Flag=1 THEN Create.File ELSE Append.File
Create.File:
REM *****
REM Put Buffer into Sequential File
REM *****
OPEN NF$ FOR OUTPUT AS #1
FOR X=1 TO N%
    PRINT#1, Names$(X)
    PRINT#1, Address$(X)
    PRINT#1, City$(X)
    PRINT#1, State$(X)
    PRINT#1, Zip$(X)
NEXT X
CLOSE #1
RETURN

Append.File:
REM *****
REM Put Buffer into Sequential File
REM *****
OPEN NF$ FOR APPEND AS #1
FOR X=1 TO N%
    PRINT#1, Names$(X)
    PRINT#1, Address$(X)
    PRINT#1, City$(X)
    PRINT#1, State$(X)
    PRINT#1, Zip$(X)
NEXT X
CLOSE #1
RETURN

Read.File:
GOSUB See.Files
REM *****
REM Put Sequential File into Buffer
REM *****
OPEN NF$ FOR INPUT AS #1
```

```

WHILE NOT EOF(1)
  INPUT#1, Names$
  INPUT#1, Address$
  INPUT#1, City$
  INPUT#1, State$
  INPUT#1, Zip$

  REM *****
  REM Send From Buffer to Screen
  REM *****
  PRINT Names$
  PRINT Address$
  PRINT City$; ", ";State$; SPACE$(1);Zip$
  PRINT
  Count=Count+1
  IF Count=5 THEN GOSUB Hold.It
WEND
  CLOSE #1
  GOSUB Hold.It
RETURN

Hold.It:
COLOR 0,1
PRINT "<Hit any key to continue>";
Hit$=INPUT$(1)
COLOR 1,0
RETURN

See.Files:
INPUT "Would you like to see the files";F$
F$=UCASE$(F$)
IF F$="Y" THEN FILES
PRINT
COLOR 0,1
INPUT "Name of File ";NF$
COLOR 1,0
PRINT
RETURN

```

That was a long program, but if you used your editing tricks, you saved a lot of duplicate efforts. Take special note of menus 2 and 4 that are not in use. The "Not in use" notice and the empty subroutines for

those two menus are to prevent a program crash. If you have some blanks in the MENU statements, you're likely to bomb your program if they are accidentally chosen. To prevent such mistakes, it is better to put something in the menus.

## FORMATTING TEXT AND NUMBERS IN FILES WITH PRINT # USING

Another way of storing information on disks is with PRINT # USING. Like the PRINT USING statement we examined in formatting output to the screen, PRINT # USING does the same thing to the disk. The format is slightly different, but the statement works essentially the same. If you use programs where the formatting of numeric data is important, such as expense accounts, PRINT # USING is very handy. The following program shows how to use PRINT # USING to keep track of expenses on a business trip:

```
Trip.Expenses:
REM *****
REM Put information into buffer
REM *****
INPUT "Trip to: ";NF$
PRINT
INPUT "Airline tickets ";AirTic
INPUT "Cab fare ";CabFare
INPUT "Hotel ";Hotel
INPUT "Meals ";Meals
TripSum = AirTic + CabFare + Hotel + Meals

REM *****
REM Send from buffer to disk with $ format
REM *****
Buck$="$$###.##"
OPEN NF$ FOR OUTPUT AS #1
PRINT #1, USING Buck$;AirTic
PRINT #1, USING Buck$;CabFare
PRINT #1, USING Buck$;Hotel
PRINT #1, USING Buck$;Meals
PRINT #1, USING Buck$;TripSum
CLOSE #1
```

```

See.File:
CLS
REM *****
REM Send file to buffer
REM *****
INPUT "Name of file to read ";NF$
OPEN NF$ FOR INPUT AS #1
WHILE NOT EOF(1)
  INPUT #1,Num$
  PRINT Num$
WEND

```

Notice how *numeric* variables are used to write the file to the disk, but we use a *string* variable to read the data from the disk. The reason for this is, when PRINT #, USING wrote the file to disk, it included string elements in the dollar sign. If you try reading that data with a numeric variable, you'll get a "Type mismatch" error. Since there may be times when you need numeric data for calculations (adding up all your trip expenses for a month or year for instance), let's see how we can change it back to numbers. Exchange the 'See.File' routine for the following to see how to make the transformation:

```

Change.File:
CLS
REM *****
REM Send file to buffer and change to numbers
REM *****
INPUT "Name of file to read ";NF$
OPEN NF$ FOR INPUT AS #1
WHILE NOT EOF(1)
  INPUT #1,Num$
  PRINT Num$,
  NL=LEN(Num$)-1
  Num$=RIGHT$(Num$,NL)
  Num=VAL(Num$)
  PRINT Num
WEND

```

```

LINE INPUT#

```

Let's take a look at what else we can do with an input statement. Suppose you want to enter a name, address and phone number into an

array, store it on disk and read it back later. Using LINE INPUT, it is possible to use a single string or string array variable to put all the information in at once. Likewise, when retrieving information from the disk, you can get a whole line using LINE INPUT #. This is especially useful when you are reading a file with an unknown format. For example, let's say that you want to read the contents of a disk but you do not know whether it is composed of strings or numeric variables or their order. By using LINE INPUT # and a string variable, you can read the file line by line rather than variable by variable.

To see how LINE INPUT works, enter the following program. When you RUN it, be sure to include commas between the store, plant and availability. Unlike the INPUT statement, commas will not result in an error message when LINE INPUT is used.

```
Line.Put:
REM *****
REM Load Buffer using LINE INPUT
REM *****
CLS
INPUT "Enter number of nurseries"; N%
DIM Store$(N%)
CLS
FOR X=1 TO N%
  LINE INPUT "Store,Plant,Available(Y/N)";Store$(X)
NEXT X
Buffer.Out:
REM *****
REM Send from Buffer to Screen
REM *****
CLS
FOR X=1 TO N%
  PRINT Store$(X)
NEXT X
```

If you want, you can change the program to send the buffer out to the disk instead of to the screen. Just change the Buffer.Out routine to be inside an open file.

The next program will divide the three items we entered in the last program into three separate arrays. The buffer will be written to the disk, using three arrays, but by using LINE INPUT# we'll load everything back into a single variable buffer. Also, take note of the method used to enter 'Avail\$(X).' A single keystroke is all that's needed, so it saves time. (Of

course, it is a little risky since the user cannot correct a typing error before pressing RETURN as with INPUT.)

```

Array.Put:
REM *****
REM Load Buffer using 3 Arrays
REM *****
CLS
INPUT "Enter number of nurseries"; N%
DIM Store$(N%),Plant$(N%),Avail$(N%)
CLS
FOR X=1 TO N%
  INPUT "Store";Store$(X)
  INPUT "Plant";Plant$(X)
  PRINT "Is it in stock (Y/N)"
  Avail$(X)=INPUT$(1)
  PRINT
NEXT X
Buffer.Out:
REM *****
REM Send from Buffer to Disk
REM *****
OPEN "Plant.Here" FOR OUTPUT AS #1
CLS
FOR X=1 TO N%
  PRINT #1,Store$(X),
  PRINT #1,Plant$(X),
  PRINT #1,Avail$(X)
NEXT X
CLOSE #1

Hit.Key:
PRINT "Hit any key to load back into memory."
Hit$=INPUT$(1)
CLS
Buffer.Back:
OPEN "Plant.Here" FOR INPUT AS #1
WHILE NOT EOF(1)
  LINE INPUT #1,Buffer$
  PRINT Buffer$
WEND
CLOSE #1

```



---

The output showed the information lined up in columns. The commas act as column tabs when the information is written to disk. To see a different effect, change the three PRINT # statements to a single line:

```
WRITE #1, Store$(X),Plant$(X),Avail$(X)
```

and change the OPEN statement to APPEND. Now when you run the program and enter data, you can see the two different ways WRITE# and PRINT# store data on the disk. The commas do not work as tab setters, and the quote marks are around the strings.

## SUMMARY

---

The disk system on the Amiga provides for several applications to make your programming tasks easier and more practical. The MERGE and CHAIN statements allow you to use small routines in different programs without having to rewrite program segments or routines. This is a further extension of the module programming style we've discussed and used throughout this book. Modular programming does more than make the creation of large, complex programs feasible; using modules with MERGE and CHAIN makes the overall task much easier.

Sequential files are very handy for a lot of applications, but they can be tricky. Work and experiment with them until you are comfortable using the various parts. The main thing to remember is to use buffers to organize data *before* it is written to the disk and *after* it has been read from the disk. However, the disk file organization can be changed once the data is in the buffers. File statements such as LINE INPUT # and PRINT #, USING help organize data between the buffer and the disk, so even disorganized data can be reformatted if the need arises.



# Random Access Files

---

### RANDOM AND SEQUENTIAL FILES: DIFFERENCES AND SIMILARITIES

---

If you can imagine sequential files to be like oil tankers that are filled up with oil by pumping the oil into big empty holds, random access files are like container ships with containers of equal size. Large or small size cargoes can go into each container, but the containers are all the same. The big advantage of random access files over sequential files is that they store each set of information as a single *record*; instead of having to open the entire file, you just open the container with your record and examine it.

To create random files, you first decide how big a container you will need, based on the maximum size of the material you will be putting in the container. Since all we can put into a random access file is numbers or strings, the problem is greatly simplified. Each character in a string takes 1 byte. (As you remember, a “byte” is a unit of measurement in the Amiga’s memory.) Therefore, if your maximum length for a given string is 10, it will be necessary to allocate a total of 10 bytes: one for each of the ten characters. With numbers, storage is different. In summary, you must plan for following allocations:

Variable	Bytes
String	1 byte per character
Short Integer	2 bytes per number
Long Integer	4 bytes per number
Single precision number	4 bytes per number
Double precision number	8 bytes per number

Data placed into a random access file must be in a string format. You will learn the special words needed to make that change. To get started, we will examine how to place simple strings into random files.

Like sequential files, you first place the information into a buffer and then send it to the file on the disk. However, there are very important differences in formatting random access files. When you OPEN a random access file, you must include the length of the file. First, as we did with sequential files, we OPEN the file and place the name of the file in quotes. However, instead of writing the mode, we indicate the file number and the length of our file. The following example shows the format for OPENing a random access file:

```
OPEN "R", #1, "RanFile",40
```

(Default is 128 if file length is not specified.)

With this statement we can either write to or read from the disk, depending on what else we place in the program. Unlike sequential files, we do not indicate whether the mode is OUTPUT or INPUT when we OPEN a random access file.

Random access files are divided into fields, each with a maximum length. The FIELD statement expects a file number, width and string variable.

```
FIELD #1, 20 AS X$, 10 AS Y$, 10 AS Z$
```

The above statement sets the width of X\$ at 20, Y\$ at 10 and Z\$ at 10. When the file is opened, the length value (the last value entered) must be equal to the total of the sum of the FIELD values. In the above example the length must be  $20 + 10 + 10 = 40$ . When you open a random access file, indicated by the "R" after the OPEN command, the value 40 must be placed at the end of the statement sequence:

```
OPEN "R",#1,"RanFile",40
```

Remember, the same format is used to open a random access file for input or output.

## RANDOM FILE BUFFERS

---

To illustrate how to use random access files, we will use a variation of the buffer we built for names and addresses in the last chapter. (Just load the routine from your disk, and we can patch this together easily.) Before we can enter the data into a random access file, we have to use the LSET statement to store our records in their respective fields. Moreover, the variable names we LSET cannot be the same ones we have in the input buffer. Therefore, we have two sets of variables: one for the input buffer and one for LSET. The nice thing about LSET is that it automatically “pads” the strings with sufficient spaces to fit the field exactly or truncate the string if it is too long. However, it is *absolutely necessary* to use different string variable names for INPUT and LSET.

Names\$ for name—LSET = N\$

Address\$ for address—LSET = A\$

City\$ for a city’s name—LSET = C\$

State\$ for a state’s two character abbreviation—LSET = S\$

Zip\$ for a zip code—LSET = Z\$

Since names of people and cities are of different length, we have to decide on a maximum size name; longer names will simply be truncated to our desired size. This process is extremely important in working with random access files since we are limited to the number of bytes specified when we open a random access file. If our entries go over that length, they will spill over into the next record. Therefore, we will limit the length of a name to 20, the address to 30, a city to 10, states to the 2-character abbreviations employed by the post office and zip codes to 5. (If you want, you can use the new longer form zip code. Just remember to add the extra amount to your calculation.) If a string is longer or shorter than the specified length, the LSET statement will take care of padding and truncating. Thus, if one of your entries has too long a name, it will be stored only at the length set in the FIELD statement. To recap:

```

N$ = 20
A$ = 30
C$ = 10
S$ = 2
Z$ = 5
TOTAL = 67

```

To get going, this next program will write a single record:

```
All.Purpose.Buffer:
```

```
N%=1
```

```
DIM Names$(N%),Address$(N%),City$(N%)
```

```
DIM State$(N%),Zip$(N%)
```

```
REM *****
```

```
REM Input Buffer
```

```
REM *****
```

```
FOR X = 1 TO N%
```

```
  INPUT "Name ";Names$(X)
```

```
  INPUT "Address ";Address$(X)
```

```
  INPUT "City "; City$(X)
```

```
  INPUT "State "; State$(X)
```

```
  INPUT "Zip Code "; Zip$(X)
```

```
  PRINT
```

```
NEXT X
```

```
REM *****
```

```
REM Write A Record
```

```
REM *****
```

```
Record%=1
```

```
OPEN "R", #1, "RanAd",67
```

```
FIELD #1,20 AS N$,30 AS A$, 10 AS C$,2 AS S$,5 AS Z$
```

```
FOR X=1 TO N%
```

```
  LSET N$=Names$(X)
```

```
  LSET A$=Address$(X)
```

```
  LSET C$=City$(X)
```

```
  LSET S$=State$(X)
```

```
  LSET Z$=Zip$(X)
```

```
  PUT #1,Record%
```

```
NEXT X
```

```
CLOSE #1
```

That was a lot of work to enter a single record, but be patient and we will do more. Now, we will GET# a record from a random access file. Like writing random access files, we must OPEN them with a specified length and read them in terms of a specified Record. The following program will read Record #1 in the name and address file:

```
REM *****
REM Read A Record
REM *****
N%=1
Record%=1
OPEN "R", #1, "RanAd",67
FIELD #1,20 AS N$,30 AS A$, 10 AS C$,2 AS S$,5 AS Z$
FOR X=1 TO N%
GET #1,Record%
PRINT N$
PRINT A$
PRINT C$; ", ";S$;SPACE$(1);Z$
NEXT X
CLOSE #1
```

All that's needed to write multiple records is a counter that increments the record number in the variable 'Record%.' However, since we are using random access files now, there is no APPEND statement. Therefore, we need a way to keep track of what the last record was so that if we want to add a record, we will not accidentally write over an existing record. To do that, we'll build on some routines to the exiting program.

First we need a little random file to store our pointer that tells us how many records there are in the file. This will be a one record file that uses a number. We'll use an integer number since it takes the least number of bytes, and we do not need fractions. Since random access files can only handle strings, we need to transform numbers into strings and then back to numbers. There are special file words for changing numbers into strings:

MKI\$—Short integer  
MKL\$—Long integer  
MK\$—Single precision number  
MKD\$—Double precision number

The following is an example of how to change a single precision variable into a string for use in a random access file:

```
LSET N$=MK$(Num)
```

Now, whatever value is in the variable 'Num' is written to the file as 'N\$.'

We also need a conversion function to change numbers stored in random access files back into integers or real numbers.

```
CVI—Short integer
CVL—Long integer
CVS—Single precision number
CVD—Double precision number
```

For example, the following would change a long integer stored as a four byte string in Record #3 back into a long integer for calculations:

```
GET #1,3
LongI&=CVL(N$)
```

Now we'll put all of this together in a program that will let us write as many random access files as we want and add new records to the file:

```
REM *****
REM File Maker
REM *****
Get.File.Name:
  INPUT "Name of file ";NF%
Point.File:
  NP$=NF$+ ".P"
Get.State:
  INPUT "Is this a (N)ew or (E)xisting file";AN$
  AN$=UCASE$(AN$)
  IF AN$="N" THEN Pointer$=1 : Flag=1
  IF AN$="E" THEN GOSUB Get.Pointer : Flag=2
  IF Flag < 1 THEN Get.State

All.Purpose.Buffer:
  INPUT "How many items to enter ";N%
  DIM Names$(N%),Address$(N%),City$(N%)
  DIM State$(N%),Zip$(N%)
```



```

REM *****
REM Input Buffer
REM *****
  FOR X = 1 TO N%
    INPUT "Name"; Names$(X)
    INPUT "Address "; Address$(X)
    INPUT "City "; City$(X)
    INPUT "State "; State$(X)
    INPUT "Zip Code "; Zip$(X)
    PRINT
  NEXT X

REM *****
REM Write A Record
REM *****
IF Flag=1 THEN Record%=Pointer% ELSE Record%=Pointer% + 1
OPEN "R", #1, NF$,67
FIELD #1,20 AS N$,30 AS A$, 10 AS C$,2 AS S$,5 AS Z$
FOR X=1 TO N%
  LSET N$=Names$(X)
  LSET A$=Address$(X)
  LSET C$=City$(X)
  LSET S$=State$(X)
  LSET Z$=Zip$(X)
  PUT #1,Record%
  Record%=Record%+1
NEXT X
CLOSE #1

Set.Pointer:
  Pointer%=Record%-1
  OPEN "R", #1, NP$,2
  FIELD #1,2 AS P$
  LSET P$=MKI$(Pointer%)
  PUT #1, 1
  Pointer%=CVI(P$)
  CLOSE #1
END

Get.Pointer:
  OPEN "R", #1, NP$,2: 'The pointer file

```

```
FIELD #1,2 AS P$
GET #1,1
Pointer%=CVI(P$)
PRINT "Begin with record #"; Pointer%
PRINT
CLOSE #1
RETURN
```

Notice how we made an accompanying “pointer file” name simply by appending a “.P” to whatever name we used for the main file. In effect, each time the program is run, it works with two separate but related random access files.

## FINDING AND CHANGING RECORDS

---

The big advantage of random access files is in locating and changing records without having to load the whole file into memory. If you need to change or locate information, the record number lets you go directly to it.

**SEARCH FOR RECORD.** Finding a single record is relatively simple, but there is a trick that you will need to know. When the information in a buffer is saved to disk, it is automatically “padded” with LSET. That means there will be a lot of extra spaces added to the string when it is read from the random file. Unless you get rid of the excess spaces, you’ll never match the search string with the one in your file. The little routine ‘Strip.Pad’ contained in the following program does that by just taking the LEFT\$ portion of the string from the record that matches the length of the search string. Since all the padding is done with spaces added to the right end of the string, it is easy to solve this problem. You just have to realize what you are looking for. For example, since our program has all strings in the name field defined as 20 bytes long (i.e., 20 AS N\$), a name like ‘Tom Jones,’ which has nine characters, will have 11 spaces padded on with LSET. If the search string is ‘Tom Jones,’ then the length of it will be nine. (The space between ‘Tom’ and ‘Jones’ is counted.) By truncating all but the first nine characters of all names compared with the search string, if ‘Tom Jones’ is in our file, it will be found.

```
REM *****
REM Find Record
```

```
REM *****
WIDTH 62
Get.File.Name:
  INPUT "Name of file ";NF$
  Point.File:
  NP$=NF$+ ".P"
Get.State:
  GOSUB Get.Pointer

Name.Find:
  INPUT "Name to find";Who$
REM *****
REM Read A Record
REM *****
INPUT "Begin with Record #";Record%
OPEN "R", #1, NF$,67
FIELD #1,20 AS N$,30 AS A$, 10 AS C$,2 AS S$,5 AS Z$
FOR X=Record% TO Pointer%
GET #1,Record%
Names$=N$
Address$=A$
City$=C$
States$=S$
Zip$=Z$
PRINT
Strip.Pad:
  Lfind=LEN(Who$)
  Match$=LEFT$(Names$,Lfind)
  IF Match$=Who$ THEN GOSUB Show.Find
Next.Record:
  Record%=Record%+1
NEXT X
CLOSE #1
IF Flag <> 7 THEN PRINT "Name not found." : BEEP
END

Get.Pointer:
OPEN "R", #1, NP$,2
FIELD #1,2 AS P$
GET #1, 1
Pointer%=CVI(P$)
```

```

PRINT "There are"; Pointer% ;" records in this file"
PRINT
CLOSE #1
RETURN
Show.Find:
  PRINT Names$
  PRINT Address$
  PRINT City$," ", ";State$;SPACE$(1);Zip$
  PRINT
  COLOR 0,1
  PRINT "Press any key to continue";
  COLOR 1,0
  Hit$=INPUT$(1)
  X=Pointer%
  Flag=7
  RETURN

```

**CHANGING A RECORD.** Since each record is written almost like a “mini-file,” it is possible to go into an existing random access file and change a single record without having to change the entire file. This is done by rewriting the record by record number. Thus, all you need is to find the record number of the record you wish to change. To make this easy, the program should first show the records in the file by record number; you don’t want to have to memorize all the record numbers. The process is then identical to creating or appending a random file except you can put the information anywhere in the file you want, not just at the end of the file. In outline, the blocks of the program include:

1. Open the file
2. Show the records and record numbers
3. Choose the record by number
4. Put new information into buffer
5. Write single file to disk

Notice how we reused some slightly changed parts of other programs. Remember, it is smarter to use existing blocks of programs than reinventing the wheel every time you sit down and program.

```

REM *****
REM Change Records
REM *****

```

```
Get.File.Name:
  INPUT "Name of file ";NF$
  NP$=NF$ + ".P"
Get.State:
  GOSUB Get.Pointer
  REM *****
  REM Show Records
  REM *****
  INPUT "Begin with Record #";Record%
  CLS
  OPEN "R", #1, NF$,67
  FIELD #1,20 AS N$,30 AS A$,10 AS C$,2 AS S$,5 AS Z$
  FOR X=Record% TO Pointer%
    GET #1,Record%
    Names$=N$
    Address$=A$
    City$=C$
    State$=S$
    Zip$=Z$
    COLOR 0,1
    PRINT "Record #";Record%
    COLOR 1,0
    PRINT Names$
    PRINT Address$
    PRINT City$;" ", ";State$;SPACE$(1);Zip$
    Record%=Record%+1
    Count=Count+1
    IF Count >3 THEN Count=0 : GOSUB Hold.It
  NEXT X
  CLOSE #1
  PRINT

Choose.Record:
  INPUT "Which record to change";Record%
  GOSUB Record.Changer
  PRINT
  PRINT "Change another record (Y/N)"
  An$=INPUT$(1)
  An$=UCASE$(An$)
  IF An$="Y" THEN Choose.Record
```

```
END
```

Get.Pointer:

```
OPEN "R", #1, NP$,2
FIELD #1,2 AS P$
GET #1, 1
Pointer%=CVI(P$)
PRINT "There are"; Pointer % ;" records in this file"
PRINT
CLOSE
RETURN
```

Hold.It:

```
PRINT
COLOR 0,1
PRINT "Press any key to continue";
COLOR 1,0
Hit$=INPUT$(1)
RETURN
```

Record.Changer:

```
REM *****
REM Input Buffer
REM *****
CLS
INPUT "Name"; Names$
INPUT "Address "; Address$
INPUT "City "; City$
INPUT "State "; State$
INPUT "Zip Code "; Zip$

REM *****
REM Write Single Record
REM *****
OPEN "R", #1, NF$,67
FIELD #1,20 AS N$,30 AS A$,10 AS C$,2 AS S$,5 AS Z$
LSET N$=Names$
LSET A$=Address$
LSET C$=City$
LSET S$=State$
LSET Z$=Zip$
PUT #1,Record%
CLOSE #1
Return
```

Now that you have seen how to create, append, read, find and change records in random access files, see if you can write a program that does all these things. Using parts of the programs you already have, merge the various parts together to make one large 'master' random access file program.

## **SUMMARY**

---

The secret to random access files, like all other programming, is careful planning broken down into small modules. This is especially true with random access files since it is necessary to decide at the outset how many bytes are to be allocated to different fields within each record. However, once the plan is laid and the bytes are allocated, the rest is relatively simple. For practical purposes of keeping records, they are a lot easier than sequential files since each record can be treated almost like a unique file in itself. This makes it easy to change the information in individual records.

The automatic padding and parsing in LSET help in keeping everything the correct size in the file, but when getting data from the file, it is important to remember that spaces have been added to the file information. Therefore, once the strings are in a buffer, it is necessary to strip spaces from the strings. Similarly, when using number translation words, remember to change the numbers to strings for storage and real numbers or to integers for calculations as needed.





# Printer Control

Your Amiga either gets information from a source or puts it somewhere. Your printer is another device where you can put information: the statement `PRINT` "This message" puts information on your screen; `SAVE` or `PRINT#` puts information on your disk. Just like you put things on your screen or disk, you can put it on your printer. However, the procedures for sending material to your printer and using your printer's special capabilities require certain routines we have not yet discussed. While much of what we will examine in this chapter will not be new in terms of the language of commands, it will be new in terms of how to arrange those commands.

In addition, we'll see how the printer can be used for things other output devices do not handle very well. For example, no matter how long a program listing is, it can be printed out to the printer, while long listings on the screen scroll right off the top into silicon oblivion. Likewise, mailing labels and letters need to be sent to the printer. In short, while not everything needs to be printed, a number of things do, and it is very simple with Amiga BASIC.

There are a vast array of printers available for use with your Amiga. However, to keep things simple and to show the maximum use of your Amiga with a printer, most examples will be done with a typical dot matrix printer. This type of printer will provide all graphic and text features you will need, and it is easily interfaced with the Amiga system. It is also very inexpensive. If you have another printer and an interface for the Amiga, then you will have to rely heavily on your printer's

manual. Unfortunately, many printer manuals are not very good for beginners since they tend to use highly technical descriptions of how to interface and operate their printers. Therefore, pay special attention to the codes used to turn on or off special features of your printer. This is usually done with a CHR\$ function from BASIC that we will learn in this chapter.

### Amiga Notes

#### Getting The Right Printer

Before you run out and buy a printer, get some advice from a fellow Amiga owner. You can hook up either a parallel or serial printer to your Amiga, but most printers run off the parallel port. (The printer icon is used to indicate the parallel port.) Depending on what you intend the *primary* function of your printer to be, you will want one type or another. Here is a list of printers with suggested uses:

1. **Dot Matrix.** This is the most versatile printer for text and graphics. There is a wide variety of types and quality; for an all around general use printer, this one is hard to beat. The higher quality ones can produce near "letter quality" text and excellent graphics as well.
2. **Ink Jet.** These printers have all of the qualities of a dot matrix printer, but are much quieter. The replacement cartridges have to be replaced more often than ribbons. They are not recommended for use where carbon duplicates are required.
3. **Daisy Wheel.** This printer is something like a computer driven typewriter and produces letter quality type. For heavy text work, such as business letters and multiple copy "carbon" printing, these are excellent. However, they make a lot of racket and they are much slower than dot matrix printers. Also, they are not very good for graphics.
4. **Laser Printer.** These printers are very expensive compared with the others we've discussed, but for desk top publishing (such as newsletter and small press publications), they are fast becoming a standard. They produce near typeset quality text and graphics, and they have been used to typeset books and magazines.
5. **Color Printers.** Dot matrix and ink jet printers are available in color as well as black ink. If you want to take advantage of your Amiga's color capabilities, this may be what you want.

(Continues)

Their qualities vary greatly, and be sure to get a demonstration on an Amiga of the printer's capabilities before you buy. The price of these printers has dropped in the last few years, and good ones are very affordable. Be sure the printer can handle black only ribbons and ink jets as well as color since there is often a very short life span on the color ribbons and ink cartridges. Also, remember that while the Amiga color looks great, reproducing color for newsletters or some similar project is expensive.

## TEXT OUTPUT TO THE PRINTER

Listing a program on your printer is a good way to de-bug it, so we'll start with a program listing. Load any program you would like listed to your printer, turn on your printer, make sure it is "on-line," and enter:

```
LLIST <Return>
```

Instead of listing to your screen, your listing was to your printer. The LLIST statement stems from (L)ineprinter LIST. Most printers used today are either dot matrix or daisy wheels, but think of your printer as a Lineprinter, and it will help remember why you have to stick an "L" before the statements that access your printer. Now that we can LLIST a program to the printer, let's see how we can LPRINT as well. Enter the following:

```
LPRINT "Amiga Computer" <Return>
```

Again, it is simple to have output go to the printer instead of the screen.

Let's try a little program that works like a typewriter to see how LPRINT works:

```
WIDTH 60
WHILE T$ <> "@"
  T$=INPUT$(1)
  LPRINT T$;
'Take the following line out to see what happens
  PRINT T$;
```

```
'The above line lets you see your text on the screen.  
X=X+1  
IF X=60 THEN X=0 : LPRINT  
WEND
```

The first thing we will learn about is the ASCII code. ASCII (pronounced ASS-KEY) stands for the American Standard Code for Information Interchange. Essentially, this is a set of numbers that has been standardized to reference certain characters. In Amiga BASIC the CHR\$ (character string) function ties into ASCII and can be used to directly output ASCII. As we will see, the CHR\$ command is very useful for outputting special characters. Certain characters used with Amiga BASIC are not standard ASCII, but most of the normal alphanumeric characters are.

## DECODING WITH CHR\$

---

Up to this point we have not used control characters. To take a look at your control characters, hold down the Control key and press several different keys. From the Program Mode, you can create those characters and more using the CHR\$ function. For example, to get a paragraph symbol, ¶:

```
PRINT CHR$(182) <Return>
```

Whenever we want to access a character, all we have to do is enter the CHR\$ and the decimal value of the character we want. For example the alphabet begins at 65, so to run through the alphabet from A-Z, you would write something like the following program:

```
FOR X= 65 TO (65 + 25)  
  PRINT CHR$(X);  
NEXT X
```

There are keys and characters that are inaccessible in a program unless you use the ASCII code or CHR\$ to get them. For example, the following currency conversion program shows how to use CHR\$ to get both the Japanese Yen and British Pound symbols:

```
'Foreign Currency converter
PRINT "How many ";CHR$(165);:INPUT " to the dollar";Yen
PRINT "How many dollars per ";CHR$(163);:INPUT Pound
CLS
PRINT "Choose conversion 1 or 2"
PRINT
FOR X=1 TO 2
  Currency=165 + Loop
  PRINT X;". ";CHR$(Currency) : PRINT
  Loop = Loop - 2
NEXT
Funds$=INPUT$(1) : Funds=VAL(Funds$)
ON Funds GOSUB Rising.Sun,God.Save.The.Queen
END
```

Rising.Sun:

```
CLS
INPUT "How many dollars";Bucks
NewDough=Bucks * Yen
PRINT
PRINT "You have "; CHR$(165);NewDough
RETURN
```

God.Save.The.Queen:

```
CLS
INPUT "How many dollars";Bucks
NewDough=Bucks / Pound
PRINT
PRINT "You have "; CHR$(163);NewDough
RETURN
```

Note how the program used numeric variables to generate the correct CHR\$ values for the Yen and Pound in the loop. That is another possible use of CHR\$ to incorporate in your programs.

You may also want to find certain CHR\$ values to use in your programs simply by pressing the various keys. For example, suppose you want to use one of the ten function keys along the top of your keyboard, the ESC key or some other key that does not show up on your screen when you press it. Using CHR\$, you would be able to use that key in your program.

The opposite of CHR\$ is ASC. It finds the ASCII values of the character you want. For example,

```
PRINT ASC("A")
```

would return a "65," the ASCII value of a capital "A." The following program shows you the value of any key you press that has an ASCII value. You might want to take note of the value of the ten function keys, the arrow keys, the HELP key and other keys you may want to use. The program uses the ESC key to exit using its value, CHR\$(27). Also, be sure to press the CTRL key in conjunction with the alphabetic keys to see what their values are as well. (If you press CTRL-C, the program will stop, since that is a break sequence.)

```
'CHR$ finder
Key.Me:
PRINT "Hit a key (ESC to quit)"
KEY$=INPUT$(1)
PRINT "The ASCII value of ";KEY$:" is";ASC(KEY$)
IF ASC(KEY$)=27 THEN END ELSE Key.Me
```

Now that you have a handle on going between ASCII and related characters, here's a program that will map the whole thing for you.

```
WIDTH 60
FOR x=32 TO 255
count = count + 1
IF count = 72 THEN count=0: GOSUB Holdit
PRINT x;"=";CHR$(x),
NEXT
END
```

```
Holdit:
PRINT
COLOR 0,1
PRINT "Hit a key";
COLOR 1,0
A$=INPUT$(1)
PRINT
RETURN
```

---

## CHRS AND PRINTER CONTROL

---

Most printer manuals are less than crystal clear on what is required to make the printer do all of its tricks. However, once you learn how to interpret the manuals, they can be very useful. First of all, there is a sequence of codes, usually control or ESC sequences, that are required to get your printer to do something. For example, if your printer manual says that

ESC \$A  
or ESC HA  
or ESC ctrl-J

(or some cryptic variation thereof) will make your printer perform a linefeed, here's what's going on:

1. First the printer wants the ESC (escape) code—CHR\$(27).
2. Next it wants Hex value A, which is 10 in decimal. (Both \$A and HA are ways to indicate hex values.) To convert from hex to decimal just PRINT &Hxx where 'xx' is the hex value. You will get the decimal equivalent. Use only the derived decimal value in the CHR\$ function. To find the values of things like ctrl-J, use the ASCII converter program above.
3. The combined sequence is CHR\$(27) + CHR\$(10).
4. LPRINT the sequence to your printer.

For example:

```
WIDE$=CHR$(14)  
PRINT WIDE$; "This is wide type"
```

will print double width on most dot matrix printers. It's a lot clearer to use descriptive string names, as we did in the above example, rather than just the CHR\$ values.

The following shows character string values and functions for the Epson printer:

CHR\$(8)—Back space  
CHR\$(10)—Line feed

CHR\$(12)—Form feed  
CHR\$(13)—Carriage return  
CHR\$(14)—Double width  
CHR\$(15)—Condensed  
CHR\$(18)—Turn off condensed  
CHR\$(20)—Turn off double width  
CHR\$(27) + "X"—Escape, used in conjunction with the following characters [e.g., CHR\$(27) + "E" would turn on emphasized printing]:  
"E"—Emphasized printing  
"F"—Turn off emphasized printing  
"G"—Double strike printing  
"H"—Turn off double strike printing  
"K"—Normal density printing  
"L"—Dual density printing  
"Q"—Set column width

Now that we know the various combinations for setting typefaces, we'll create a program that will execute them for us on our printer. We will be using the Epson printer codes, and if you are using a different type of printer, just substitute the ones that work on your printer for those used in the example. With the exception of the expanded typeface on your printer, CHR\$(14), once you send a change of typeface to your printer, it will stay there.

```
TypeFacer:
COLOR 0,1
PRINT "Be sure your printer is turned on and on line";
COLOR 1,0
PRINT : PRINT
PRINT "Hit any key to continue:"
Hit$=INPUT$(1)
```

```
GetFace:
CLS
PRINT "Which face?"
PRINT
PRINT "{W}ide" : PRINT
PRINT "{C}ondensed" : PRINT
PRINT "{D}ouble Strike" : PRINT
PRINT "{E}mphasized" : PRINT
PRINT "{N}ormal" : PRINT
```



```
Typeface$=INPUT$(1)
Typeface$=UCASE$(Typeface$)
```

JumpFace:

```
IF Typeface$="W" THEN Fatface
IF Typeface$="C" THEN Littleface
IF Typeface$="D" THEN TwoBang
IF Typeface$="E" THEN HardPoint
IF Typeface$="N" THEN Mundane
```

Fatface:

```
LPRINT
Fat$=CHR$(14)
LPRINT Fat$; "Fat Face";
END
```

Littleface:

```
LPRINT
Scrunched$=CHR$(15)
LPRINT Scrunched$; "You can get a lot in a little ";
LPRINT "space with this face."
END
```

TwoBang:

```
LPRINT
ESC$=CHR$(27)
BangBang$=ESC$ + "G"
LPRINT BangBang$; "This will look BOLD"
END
```

HardPoint:

```
LPRINT
ESC$=CHR$(27)
Stress$=ESC$+"E"
LPRINT Stress$; "This is to really make a point."
END
```

Mundane:

```
LPRINT
ESC$=CHR$(27)
LPRINT CHR$(18)
LPRINT ESC$ + "F"
```

```
LPRINT ESC$ + "H"
END
```

Now, let's take a look at a program that will do something useful with the printer. For example, you might want to have a printed copy of a list (called a 'hardcopy' in computer parlance) of your program library, so you could see at a glance what programs you have. The following program follows the same general format used with the file programs. First, load everything into a buffer. Then output the buffer to the printer instead of a file.

```
ProgramFiler.Printer:
  Input.Buffer:
  INPUT "How many programs to print"; N%
  DIM Program$(N%)
  FOR X=1 TO N%
    INPUT "Program name ";Program$(X)
  NEXT X

Printer.Output:
  FOR X=1 TO N%
    LPRINT Program$(X)
  NEXT X
```

That was nice and simple, but it was not a lot different than using a typewriter to do the same thing. If we could use the printer in conjunction with a file program, then it would be possible to take something out of a file and send it to the printer. Go back to Chapter 15 and get the big sequential file that wrote the names and addresses. Change the routine with the title:

```
REM *****
REM Send From Buffer to Screen
REM *****
etc . . . .
```

to the following:

```
REM *****
REM Send From Buffer to Printer
REM *****
```

```
LPRINT Names$
LPRINT Address$
LPRINT City$; ", ";State$; SPACE$(1);Zip$
LPRINT
WEND
CLOSE #1
RETURN
```

Now when you choose, "Read an address book" with the mouse, your file will be sent to the printer. By making a few more changes, it would be very simple to have the program send the output to both the screen and the printer. See if you can enhance the sequential file program to do both.

## USING THE LPOS(0) FUNCTION

On some applications, you will want to keep track of where the printhead on the printer is. With the LPOS function, this is simple. Using a dummy variable, LPOS (DV) returns the current head position. To see how this works, look at the following program:

```
PrintPos:
LPRINT
FOR X= 1 TO 200
  LPRINT "X";
  IF LPOS(0) > 19 THEN LPRINT
NEXT X

FOR Y= 1 TO 200
  LPRINT "Y";
  IF LPOS(0) >24 THEN LPRINT
NEXT Y
```

The LPOS function comes in very handy when you want to write a program that has special formats on your printer. For example, you may want to have wide and narrow columns in different places on your printer paper. As you saw in the above example, the X's and Y's had different "column" widths.

## OPENING LPT1:

In some applications, you may want to open the printer as a logical file, just as you would a disk file. For example, suppose you had a routine that could be used to output a format that you could use for either the screen or the printer.

A special file name, LPT1:, is used to indicate the "file" is the printer. The general format

```
OPEN "LPT1:" FOR OUTPUT AS #1
```

where "#1" can be any file number, as with disk files, will send output to that device. Thus, PRINT #1 would go to the printer if the file were opened as "LPT1:." The screen can also be used as a "file" output device with the file name "SCRN:." Instead of having two routines doing the same thing, you'd just open either the screen or the printer as the output device. The following program shows how this is done.

```
INPUT "How many names ";N%
PRINT
DIM Names$(N%),Address$(N%),City$(N%)
DIM State$(N%),Zip$(N%)

REM *****
REM All Purpose Input Buffer
REM *****
FOR X=1 TO N%
  INPUT "Name"; Names$(X)
  INPUT "Address "; Address$(X)
  INPUT "City "; City$(X)
  INPUT "State "; State$(X)
  INPUT "Zip Code "; Zip$(X)
  PRINT
NEXT X

Choose.One:
  CLS
  PRINT "Output to (S)creen or (P)rinter"
  SP$=INPUT$(1)
  SP$=UCASE$(SP$)
```

```
IF SP$ <> "P" AND SP$ <> "S" THEN Choose.One
IF SP$="S" THEN GOSUB OutScreen
IF SP$="P" THEN GOSUB OutPrinter

FOR X=1 TO N%
  PRINT #1,Names$(X)
  PRINT #1,Address$(X)
  PRINT #1,City$(X);", ";State$(X);SPACE$(1);Zip$(X)
  PRINT #1,Dummy$
NEXT
CLOSE #1
END

Outscreen:
  OPEN "SCRN:" FOR OUTPUT AS #1
  RETURN

OutPrinter:
  OPEN "LPT1:" FOR OUTPUT AS #1
  RETURN
```

See if you can make a change in the program so that the output device is a disk file. Just add an "OutDisk" subroutine.

## LPRINT USING

---

The LPRINT USING statement is just like PRINT USING except it outputs to the printer. For example, to line up a column of financial figures, you could use:

```
LPRINT USING "$$####.##"
```

The following program will print out a list of checks along with their total:

```
CheckLister:
  INPUT "How many checks to enter ";N%
  INPUT "Starting with Check number ";CN%
  DIM Checks(N%)
```

```

FOR X=1 TO N%
  PRINT "Amount for check number";CN%;
  INPUT Checks(X)
  Total=Total + Checks(X)
  CN%=CN%+1
NEXT X
PrintChecks:
Dollar$="$#####.##"
FOR X= 1 TO N%
  LPRINT USING Dollar$;Checks(X)
NEXT X
LPRINT
LPRINT "Total=";
LPRINT USING Dollar$;Total

```

You can change the 'Dollar\$' format to left justify all of the dollar signs by using a single dollar sign instead of the double one.

## CONTROL THAT FORMAT!

---

The output on the above program is all right, but the total is out of line with the column of checks. In many ways, controlling your printer format is just like controlling the format on your screen. Using the TAB( ) statement with LPRINT it is possible to line things up correctly. The statement:

```
LPRINT TAB(15)
```

for example, will set the next printed character to position 15. Run this next little program to see the spacing TAB( ) makes:

```

LPRINT
FOR K=1 TO 19
  LPRINT "X";
NEXT K
LPRINT "X"
LPRINT TAB(15)
LPRINT "X"

```

Notice that the single "X" in the second line is in the fifteenth position, not the sixteenth. Just as it works on your screen, TAB(N) positions the output on your printer in the position 'N' in parentheses. However, you must remember to have a lower tab value before a higher tab value on the same line.

Going back to our check lister program, let's rewrite it to see if we can format the output to look better. We'll also toss in a printout of the check numbers and a line under the column of check values:

```

CheckListerPlus:
  INPUT "How many checks to enter ";N%
  INPUT "Starting with Check number ";CN%
  SN%=CN%
  DIM Checks(N%)
  FOR X=1 TO N%
    PRINT "Amount for check number";CN%;
    INPUT Checks(X)
    Total=Total + Checks(X)
    CN%=CN%+1
  NEXT X

PrintChecks:
  LPRINT
  Dollar$="#####.##"
  FOR X= 1 TO N%
    LPRINT "#";SN%;
    LPRINT TAB(10)
    LPRINT USING Dollar$;Checks(X)
    SN%=SN%+1
  NEXT X
  LPRINT TAB(10)
  LPRINT "-----"
  LPRINT "Total=";
  LPRINT TAB (10)
  LPRINT USING Dollar$;Total

```

You can make calculated spacing and relative spacing with SPC( ) or SPACE\$ just as you can on the screen. However, LOCATE only applies to positioning on your screen and will not work with printer output.

## SUMMARY

---

This chapter covered two very important topics. We examined how to get output to the printer and how to use the CHR\$ function. The two topics are separate but related. The ASCII code can be used to create non-keyable and non-printed characters, or in the case of printers, issue direct commands. The CHR\$ function is a way of placing ASCII in a string format. Conversely, using the ASC function, strings can be decoded into ASCII values. Thus, you saw how to encode and decode ASCII for accessing features of your Amiga and printer that could not otherwise have been utilized.

Sending material to your printer is simply a matter of using LPRINT instead of PRINT. Many of the same formatting statements used with screen formatting can be used with your printer as well. Thus, the main difference is where the output will be placed. Likewise, your printer can be opened as an output device very much in the same way as your disk when LPRT1: is incorporated as a file name. This enables you to use a single routine to output data to different devices, including your printer, disk, screen and, as we will see in the next chapter, your communications port.



# Telecommunications

One of the most exciting things you can do with your Amiga is to talk with other computers. Not only can you talk with other Amigas, your Amiga can talk with any other computer that uses ASCII. You can even communicate with mainframes. The best part is that all this can be done with a modem and your Amiga BASIC. Just as you can send output to your screen, printer or disk, you can open a file to be sent to your built-in communications port.

Most of this chapter will consist of a large program that uses just about every trick you've learned plus some new ones. However, first, we will discuss what is involved in, and what you will need to know to perform, computer communications.

## MODEMS

---

A modem is very much like a telephone that uses computer "voices" instead of human ones. If you have ever picked up a telephone and heard a high pitched whine, you have an idea of what computers sound like when they talk. Your computer uses lines on your communications or 'RS232' port on the back of your Amiga as its "mouth" and "ears." The 25 pins on your RS232 port have the following characteristics:

Pin	Code	Characteristic
1	GND	Frame ground
2	TXT	Send data
3	RXD	Receive data
4	RTS	Request to send
5	CTS	Clear to send
6	DSR	Data set ready
7	GND	System ground
8	CD	Carrier detect
9		Unused
10		Unused
11		Unused
12		Unused
13		Unused
14	-5	-5 volt power
15	AUDO	Audio out
16	AUDI	Audio in
17	EB	Buffered port clock 716 kHz
18	INT2	Interrupt line to Amiga
19		Unused
20	DTR	Data terminal ready
21	+5	+5 volt power
22		Unused
23	+12	+12 volt power
24	C2	3.58 MHZ Clock
25	RESB	Buffered system reset

The first eight pins are of primary concern with most modems. As long as the pins are compatible with the pins on the Amiga port, everything will work fine with a standard RS232 cable. Therefore, if you do *not* have a modem, be sure the one you purchase uses the standard RS232 configuration for pins 1-8. For the most part, you won't have to worry about the pin configuration on your Amiga. Just plug one end of your modem cable into your modem and the other into your communications (RS232) port.

## TYPES OF MODEMS

There are a lot of different brands of modems on the market, but the single most important consideration is *baud rate*. The baud rate is a

---

measure of speed at which a modem can transfer data. If you subscribe to an online network, you pay by the number of minutes you use the system. The faster you can transmit data, the less time you have to spend to get information from the system. (When you get data from another system, it is called *downloading* and when you send data to another system, it is called *uploading*. Those terms are important to remember.) Thus, not only can you save time with having a higher baud rate, you can save money as well.

Unfortunately, the higher the baud rate of modems, the higher the cost. The least expensive modems are 300 baud, and they are very slow. Slightly more expensive, but significantly faster, are 1200 baud modems. These are recommended as the best deal at the time of this writing since they are four times faster than the 300 baud modems, and only slightly more expensive. (A good price for a 1200 baud modem is around \$150 compared with \$100 for 300 baud.) The 2400 baud modems are the next step, and while clearly better to have than a 1200 baud one, they tend to be expensive. The *inexpensive* ones cost around \$300–\$400. However, if your business does a lot of communications, especially over long distance lines, they may well pay for the difference in initial cost outlay as savings on long distance bills. (For instance, if your 1200 baud modem results in long distance bills of \$100 per month, you can figure a 2400 baud would cost \$50 to send the same amount of data. The \$600 annual savings of a 2400 baud modem over a 1200 baud modem would more than make up the difference and save a lot of time as well.)

## NULL MODEM

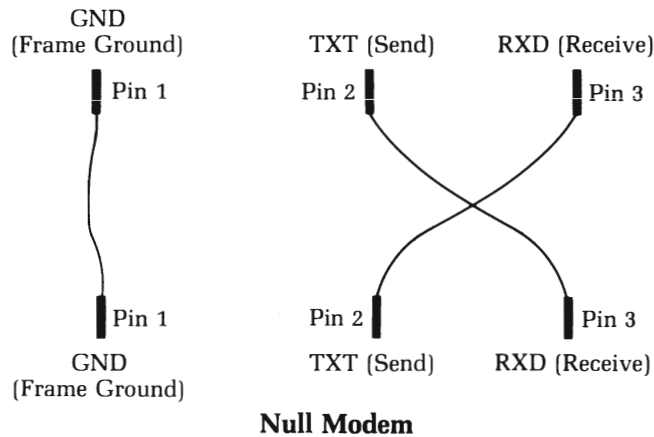
---

If you have data on a computer other than an Amiga, and you want to transfer data from your other computer to the Amiga over short distances (i.e., a few feet), there's a simple and fast way to do it. Since most ASCII formatted files, such as text files created from word processors, will directly transfer from any other computer to your Amiga, you can preserve all of your work done on another computer by transferring the files via a *null modem*. The best part about null modems, besides the fact that you can build them yourself for a few dollars, is that they run at 9600 baud.

Basically, all you have to do is to get an RS232 connector for your Amiga. (A male DB25 connector is required for the Amiga; depending on the other kind of computer, you'll need the appropriate connector for the

other end. Usually, the RS232 connector is a male or female DB25, but some computers, such as the Apple® Macintosh® require other kinds of connectors, such as a DB9.) Then, get three wires and solder them in the solder eyes of the connectors so that the grounds (GND) are connected directly, and the transfer (send) lines of one are connected to the receive lines of the other. That's all there is to it. (If you have no experience with soldering or electronics, get the help of someone who does. BY NO MEANS SHOULD ANY CONNECTIONS BE MADE BETWEEN TWO COMPUTERS WHILE EITHER HAS ITS POWER TURNED ON.)

The following diagram shows the connections and pins used with two *standard* DB25 RS232 connectors. The pin numbers may differ with different interfaces, but since the RS232 connector is a standard, you should not find too much variation.



## THE COM1: FILE

Like the printer and screen 'files', there is a special file for communications, 'COM1:.' As with disk, printer and screen files, when you open a file with 'COM1:' as the file name, all subsequent PRINT #N statements will send the information out the communications port to your modem or null modem. However, the 'COM1:' file has special parameters to set before communications will work correctly. The general format is:

```
OPEN "COM1: Baud,parity,data-bits, stop-bits" AS #N
```

When a COM1 file is opened, its parameters are treated as part of a large string defined by the characters surrounded by parentheses. Baud, data-bits and stop-bits are numbers and parity is one of three alphabetic characters. Let's take a quick look at each parameter.

**BAUD RATE.** As mentioned in the above discussion of modems, the speed of communications is measured in baud. In most cases only 300, 1200, 2400 and 9600 are relevant since most modems are set up for one of those rates. However, 110, 150, 600, 1800, 3600, 4800, 7200 and 19200 are also valid baud rates.

**PARITY.** Parity can be even (E), odd (O) or none (N). Most of the time, N is used, but you can also use E or O.

**DATA-BITS.** This can be a tricky parameter since you may be getting more than you bargained for in some transmissions. If straight ASCII is transmitted, 8 data-bits is what you want to use. However, if "overhead" bits are transmitted, then you only want the "real" information, not the overhead. This overhead is "stripped" by specifying less than 8 bits to be accepted as transmitted data. The extra parity bits and stop bits constitute the overhead. Usually there is only a 1 bit overhead so 7 bits would be used, but 5 or 6 bits may also be specified.

**STOP-BITS.** This is easy to remember. Unless transmitting at 110 baud, use 1 as the stop-bit parameter (110 baud uses 2).

For example, a fairly standard COM1: opening would be:

```
OPEN "COM1: 1200,N,8,1" AS #1
```

That would open the communications channel for 1200 baud communications with no parity, 8 bits and 1 stop-bit as file number one.

---

## READING FROM AND WRITING TO THE COM1 FILE

---

To understand how to send and receive data through the communications port, we'll have to learn what LOC(N) is and a new parameter for INPUT\$. When the COM1: file is opened, LOC(N) returns a '1' if there's something ready to be read. The 'N' is the opened file number. If there's

nothing, a "0" is returned. Thus, by scanning LOC(N) for a "1," it's easy to trap incoming data in a buffer by using a string variable or array.

When we looked at INPUT\$( ), we found that the first parameter was the number of characters to be read to make up the string. However, there's a second parameter: the file number. Thus, INPUT\$(N1,N2) specifies the number of characters to be read into a buffer (N1) from a given file (N2). Putting this together with LOC, we can take data coming into the communications port and put it in a buffer. For example, the following routine reads data coming into the communications port, puts it in a 1 byte buffer and prints it on the screen:

```
OPEN "COM1: 1200,N,8,1" AS #1
Check.Com:
WHILE LOC(1) <> 0
  RECEIVE$ = INPUT$(1,1)
  PRINT RECEIVE$;
WEND
GOTO Check.Com : REM Infinite loop
```

Now that we can see how to receive data, we will need a way to send data. That is really easy since it's just like writing data to a file. Using PRINT #N, you simply write the data to send to the COM1 file number. By scanning the keyboard for input using INKEY\$, we can wait until there's something to send and then shoot it out the communications port one character at a time. The following routine shows how to send data and use the F1 key, CHR\$(129), to exit the routine:

```
OPEN "COM1: 1200,N,8,1" AS #1
WHILE I$ <> CHR$(129)
  I$=INKEY$
  IF I$ <> "" THEN PRINT #1,I$;
WEND
```

That's all there is to sending straight ASCII code. It is possible to send other types of data, but we'll stick to ASCII since it is the easiest and least confusing. If we combine the send and receive routines, we can both read and write. The 'Dumb.Terminal' routine in the main program does just that.

---

## AMIGA TERMINAL PROGRAM

---

All right, we're set to jump into the big communications program. The heart of the program is the 'Dumb.Terminal' routine, and most of the rest is material that you already know. The program uses a sprite to indicate that your 'download' (receive data) is "on," and so the first thing to do is to create a sprite with the 'ObjEdit' program that looks like the following and save it under the name "Dnload." (The shaded areas represent different colors.)



It must be saved on the same disk as your terminal program. The reason for using a sprite is that it can superimpose itself on the screen over any text without erasing it. Everything that is received while the receive is turned on will be placed in a buffer and written to the disk. It's easy to forget that it's on, so we've included a sprite to remind you. You want to end the download once you've got the good stuff and write everything to your disk without a lot of excess junk.

Once you've created and saved the sprite, key in the following program and save it. It is set up for 60 column text, so if you have your text set to 80, change it with the "Preference" file to 60. It would be a very good idea to save the program after each module is typed in:

```
START.HERE:
  CLS : CLEAR
  GOSUB GraphicHead
  GOSUB Header
  Flag$="C" : WIDTH 62 : SW$="62"
  GOSUB Get.Sprite
  Cflag=2 : CR$=" Carriage Return On"
  FOR SET=1 TO 4 : CHECK(SET)=1 : NEXT
  Baud=1200 : CHECK(3)=2
  GOSUB Set.baud
```

```
GOSUB Set.menus
ON MENU GOSUB Get.Menu

Dumb.Terminal:
OPEN "com1:"+Baud$+",n,8,1" AS 1
WHILE I$ <> CHR$(129)
WHILE LOC(1)<>0
R$=INPUT$(1,1)
PRINT R$;
IF Flag$="R" THEN GOSUB Buffer
WEND
I$=INKEY$
IF Flag$="D" THEN I$="ATDT"+Dial$+CHR$(13)
IF Flag$="D" THEN Flag$="C" : PRINT I$
IF Flag$="X" THEN Flag$="C": CLOSE : GOTO Dumb.Terminal:
IF I$<>" " THEN PRINT #1,I$;
WEND

End.It.All:
MENU RESET:CLOSE:END

Get.Menu:
Menupick=MENU{0}
ON Menupick GOSUB Get.baud,Get.Term,Get.Ops
RETURN

Set.menus:
MENU ON
MENU 1,0,1, "Set Baud Rate"
MENU 1,1,CHECK(1), " 9600"
MENU 1,2,CHECK(2), " 2400"
MENU 1,3,CHECK(3), " 1200"
MENU 1,4,CHECK(4), " 300"

MENU 2,0,1, " Communicate"
MENU 2,1,1, "Start Receive ASCII"
MENU 2,2,1, "End Receive ASCII"
MENU 2,3,1, "Send ASCII"
MENU 2,4,1, "Terminal"

Men.Ops:
MENU 3,0,1, "Options"
```



```
MENU 3,1,1, "Clear screen"
MENU 3,2,Cflag, CR$
MENU 3,3,1, "Dial a Number"
MENU 3,4,1, "Screen Width " +SW$

MENU 4,0,1, ""
MENU 4,1,1, "Nothing"
RETURN

Get.baud:
FOR CK=1 TO 4: CHECK(CK)=1 : NEXT
ON MENU (1) GOSUB b9,b2,b1,b3
GOSUB Set.menus
CLOSE : Flag$="N" : GOSUB Dumb.Terminal
RETURN

b9:
Baud=9600 : GOSUB Set.baud:CHECK(1)=2 : RETURN
b2:
Baud=2400 :GOSUB Set.baud :CHECK(2)=2 : RETURN
b1:
Baud=1200 :GOSUB Set.baud :CHECK(3)=2: RETURN
b3:
Baud=300 : GOSUB Set.baud :CHECK(4)=2: RETURN

Set.baud:
Flag$="X"
Baud$=STR$(Baud)
C$="com1:"+Baud$+",n,8,1"
PRINT "Baud Set at";Baud$
RETURN

Get.Term:
ON MENU(1) GOSUB R1,R2,R3
RETURN

R1:
REM Start Receive
DIM Buffer$(255)
Flag$="R"
INPUT "Name of File to Save";RF$
SAY TRANSLATE$("Receiving data now")
```

```
GOSUB Show.Sprite
RETURN
```

```
R2:
  REM End Receive
  OPEN RF$ FOR OUTPUT AS 2
  Flag$="T" : REM 'T' is for 'Terminal'
  FOR X = 0 TO COUNT
    PRINT#2,Buffer$(X)
  NEXT
  CLOSE #2
  SAY TRANSLATE$ ("Buffer saved to disk")
  FOR Pause=1 TO 200: NEXT Pause
  SAY TRANSLATE$("Receive OFF")
  OBJECT.OFF
  ERASE Buffer$
RETURN
```

```
R3:
  REM Send File
  CLS : COLOR 0,1
  PRINT "Directory(Y/N)?";
  DIR$=INPUT$(1)
  COLOR 1,0 : PRINT
  DIR$=UCASE$(DIR$)
  IF DIR$="Y" THEN FILES
  INPUT "File to send";FS$
  ON ERROR GOTO Fix.Error
  OPEN FS$ FOR INPUT AS#2
  IF Flag$="Error" THEN Flag$="C" : RETURN
  WHILE NOT EOF(2)
    IF Cflag=1 THEN INPUT #2, SEND$
    IF Cflag=1 THEN PRINT #1, SEND$;
    IF Cflag=2 THEN LINE INPUT #2, SEND$
    IF Cflag=2 THEN PRINT #1, SEND$
    IF Echo$="On" THEN PRINT SEND$;
  WEND
  PRINT #1,CHR$(13)
  CLOSE #2
  SAY TRANSLATE$ ("Transfer complete")
```

---

```
GOSUB Header
RETURN

Get.Ops:
  ON MENU(1)GOSUB Header,Carriage,Dial,Wide
RETURN

Header:
  CLS
  COLOR 0,1
  PRINT "Press the 'F1' key to quit"
  COLOR 1,0
RETURN

Carriage:
  IF Cflag=1 THEN Cflag=2 : CR$=" Carriage Return On" :GOSUB Men.Ops:
RETURN
  IF Cflag=2 THEN Cflag=1 : CR$="Carriage Return Off"
GOSUB Men.Ops
RETURN

Dial:
  INPUT "Enter number to dial please";Dial$
  Flag$="D"
RETURN

Wide:
  INPUT "Screen width desired (1-255) ";SW
  SW$=STR$(SW)
  WIDTH SW
  GOSUB Men.Ops
RETURN

REM *****
REM Subroutine Suburbia
REM *****

Get.Sprite:
  OPEN "Dnload" FOR INPUT AS 3
  OBJECT.SHAPE 1,INPUT$(LOF(3),3)
```

```
CLOSE 3
```

```
RETURN
```

```
Show.Sprite:
```

```
OBJECT.X 1,570
```

```
OBJECT.Y 1,15
```

```
Turn.On:
```

```
OBJECT.ON
```

```
RETURN
```

```
Buffer:
```

```
Buffer$(COUNT)=Buffer$(COUNT) + R$
```

```
IF LEN(Buffer$(COUNT)) = 254 THEN COUNT=COUNT+1
```

```
RETURN
```

```
Fix.Error:
```

```
IF ERR = 53 THEN CLS : PRINT "File not on disk"
```

```
PRINT "Try again and use the Directory option."
```

```
Flag$="Error"
```

```
IF ERR <> 53 THEN CLS : PRINT "Unknown error" : LIST : END
```

```
RESUME NEXT
```

```
GraphicHead:
```

```
PALETTE 0,0,0,0
```

```
PALETTE 1,1,0,0
```

```
COLOR 0,1
```

```
CLS
```

```
WIDTH 255
```

```
Amiter$=" Bay Sic Amiga "
```

```
BA$=" BASIC Amiga "
```

```
Version$= " Version 1.0 "
```

```
Term$="Terminal Program "
```

```
Hit$="(Hit any key to continue)"
```

```
SAY TRANSLATE$(Amiter$)
```

```
FOR X= 1 to 62
```

```
  SOUND (500+(X*3)),2
```

```
  LOCATE 10,X
```

```
  PRINT BA$;
```

```
NEXT
```

```
FOR X=62 TO 23 STEP -1
```

```
  LOCATE 10,X
```

```
    PRINT Term$;
NEXT
SAY TRANSLATE$(Term$)
LOCATE 11,31-LEN(Version$)/2
PRINT Version$
LOCATE 18,31 - LEN (Hit$)/2
PRINT Hit$
BEEP
A$=INPUT$(1)
PALETTE 0,0,0,.6
PALETTE 1,1,1,1
COLOR 1,0
RETURN
    PRINT Term$;
NEXT
SAY TRANSLATE$(Term$)
LOCATE 11,31-LEN(Version$)/2
PRINT Version$
LOCATE 18,31 - LEN (Hit$)/2
PRINT Hit$
BEEP
A$=INPUT$(1)
PALETTE 0,0,0,.6
PALETTE 1,1,1,1
COLOR 1,0
RETURN
```

After all of that work, be sure to save a back up copy of the program on a separate disk.

## USING THE AMIGA TERMINAL PROGRAM

---

The program may have been hard to write, but it is easy to use. Everything is done with the mouse and menu bar. The three menus include the following:

1. *Set baud rate.* If your modem is other than 1200 baud, change the default value to whatever baud your modem is in the program. Alternatively you can click the baud rate from the menu bar. This

will be very useful if you have a 1200 or 2400 baud modem and wish to communicate with someone who has a slower rate. For example, if you have 1200 baud and you call a bulletin board with 300 baud, you can change the baud rate by clicking 300 on the baud rate menu.

2. *Communicate.* To upload (send) and download (receive) data, use this menu. You can only send or receive ASCII files with this program, so if you try any other kind of data, you're liable to get garbage. To begin a download, click the "Start Receive ASCII" option. Provide a unique file name (i.e., one not on your disk!) when prompted for a file name. Your sprite will pop up on the screen and your voice synthesizer will let you know that you have begun receiving data. When you get the data you want, click "End Receive ASCII" and your buffer will be written to the disk and the sprite will disappear. (Your Amiga will tell you that the data has been written to the disk.) The "Terminal" option just returns to the "Dumb Terminal" loop.
3. *Options.* You have four options. The "Clear screen" option just clears the screen. Once it gets filled with text, you can clear it if you want to send something from the keyboard. Second, you can toggle the "Carriage Return" on or off. Basically, this will determine whether or not a carriage return is placed at the end of a line or not. When sending program files in ASCII format, leave the carriage return option on. When sending text, turn it off.

The "Dial a number" option is a little different since it uses a special protocol not found on all modems. On some modems, a special "Hayes" protocol can be used employing the built-in autodial features of a modem. If your modem does not have autodial capabilities or does not use the "Hayes" version, this feature will not work. However, since the "Hayes" protocol is fairly standard on most modems using autodial, try this option if your modem has autodial features. (Autodial simply means you can dial a number from your computer rather than having to dial from your phone's buttons.) When prompted to enter a number, just type in the number you want to call and press RETURN. On your screen, "ATDT" and the phone number will appear. The "ATDT" is the "Hayes" modem code for dialing a touchtone number. The "AT" preface is used for all other built-in modem commands as well. You can use them directly from the default terminal mode. (The 'terminal mode' is the "Dumb.Terminal" routine that loops looking for keyboard input or input from the

---

communications port.) Just key in "AT" and the other code from the terminal mode, and your modem will perform its designated function. If your modem uses another protocol, read your modem manual, and type in its protocol from the terminal mode.

The final option is to change the screen width. If you want to communicate using a wider screen, such as 80 columns, change the screen width first from the 'Preferences' on the Workbench, and then change it from the 'Options' menu. You may also wish to change the default from 62 to something else in the program itself.

## SUMMARY

---

This chapter was designed to do two things. First, it showed you how to work with COM1: files. The communications port is simply treated as another place to receive or send data. However, by treating it as a file, it is a lot easier to use. The LOC function and INPUT\$ with two parameters allow easy access to the communications port.

A second and perhaps more important purpose of this chapter was to bring together the sum of the parts into a whole. We were able to use graphics, menus, sound, voice and different types of files all in a single program. In some cases, more than a single file was opened simultaneously to accommodate disk and communications files. All of the techniques that have been presented were integrated usefully. However, since each part was in a separate module, it was not that difficult to create. Moreover, if you want to add something to the program, you should be able to do so with little difficulty. By learning how to write your own programs, you can tailor the program to fit your needs more precisely, and that, after all, is the purpose of computers.





# **Algorithms and Advanced Techniques**

This final chapter illustrates what we have been attempting to do throughout this book: develop good programming techniques. Perhaps we've said it too many times, but the secret to good programming is breaking a big problem down into manageable sized modules. However, there are various techniques to make those modules more efficient by using the correct algorithm. In the context of programming, efficiency refers to two things: less programming code and less execution time. If one algorithm takes less code and executes faster than another, then it is better to use. Throughout this book, we've used various algorithms to get things done. Many have not been too efficient since we were attempting to keep things relatively simple so that you could understand what was happening and learn new statements, functions and commands.

The advanced techniques we will examine constitute extending our knowledge of existing programming techniques instead of learning new ones. Using what we know, we can do more things by incorporating extensions of current knowledge so that the new information is a matter of reorganizing the old. Using examples from artificial intelligence, we will see how your Amiga can be turned into an "intelligent" machine. The IF . . THEN statement will be used in a new, expanded way.

## A GOOD ALGORITHM IS WORTH A THOUSAND LINES OF CODE

---

The heart of good programs are good algorithms. Essentially algorithms are routines, formulas or sets of instructions that perform single tasks. Throughout this book, you've been introduced to all different kinds of algorithms even though we did not call them by that name. Some algorithms you will develop on your own, while others are fairly standard solutions to programming problems. Usually programmers just "look up" the standard algorithms and use them whenever needed.

Some of the most persistently revised and discussed algorithms are sort routines. Sorts are algorithms that put lists in alphabetical or numerical order. There are a lot of different sorts, but we will only deal with two in order to show how one algorithm can perform a task more efficiently than another. The sorts we will examine are relatively simple so that you can better understand what is going on. However, even if you do not fully see why one is more efficient than another (or even how they work), you can simply compare the speeds at which each sorts a list of strings.

### THE BUBBLE SORT

---

The bubble sort is so named since the strings near the top "bubble up" from the bottom of the list. Like all sorts, the bubble sort works with string arrays rather than non-array strings. There are two listings with the bubble sort below: one to illustrate graphically on your screen how the sort works, and the other to give a more practical demonstration.

The bubble sort works by comparing the two strings

```
A$(S) <= A$(S+1)
```

and either getting the next string or swapping the two compared strings. For example, suppose the original list had the following:

```
A$(7) = "Oranges"  
A$(8) = "Apples"
```

When the sort compares those two, it would be:

```
A$(S) = "Oranges"  
A$(S+1) = "Apples"
```

Since "Oranges" is greater than "Apples" (i.e., the ASCII value of 'O' is greater than the ASCII value of 'A') the two strings are swapped so that they are now:

```
A$(S) = "Apples"  
A$(S+1) = "Oranges"
```

The routine continues until all of the strings are in order.

```
Bubble.Sort:  
CLS  
REM *****  
REM Input Buffer  
REM *****  
INPUT "Number of strings to sort ";N%  
DIM A$(N%+1)  
FOR X=1 TO N%  
  PRINT "String #";X;  
  INPUT A$(X)  
NEXT X  
T=X-1  
CLS  
  
REM *****  
REM Unsorted Output  
REM *****  
FOR X=1 TO N%  
  PRINT A$(X)  
NEXT X  
  
Bubble.Algorithm:  
REM *****  
REM Sort Strings  
REM *****
```

```
Compare:
  Flag=0
  FOR S=1 TO T
    IF A$(S) <= A$(S+1) THEN Get.Next.String
    SWAP A$(S),A$(S+1)
    LOCATE S,1 : PRINT SPACE$(50)
    COLOR 0,1
    LOCATE S,1 : PRINT A$(S)
    COLOR 1,0
    LOCATE S+1,1 : PRINT SPACE$(50)
    LOCATE S+1,1 : PRINT A$(S+1)
    Flag=1
    T=S
  Get.Next.String:
  NEXT S
  IF Flag=1 THEN Compare

Clear.Top:
  LOCATE 1,1 : PRINT SPACE$(50)
  LOCATE 18,1
```

To see the “bubble” effect, enter the following list of 10 words:

```
apples
bananas
cranberry
grapes
lemons
oranges
peaches
kumquats
raisins
alfalfa
```

You'll see alfalfa “bubble” right to the top. Bubble sorts are most efficient with partially sorted lists. In fact, they are one of the most efficient sorts with partially sorted lists. While other sorts are much better algorithms for sorting wholly unsorted lists, the bubble sort is good for several applications. For example, if you have a data base to which you keep adding names, each time you add a name, you add it to a partially

sorted list. (This assumes you sort it each time you add a new name.) Therefore, the lowly bubble sort may be just the algorithm you need in some applications.

To give you a better example of the bubble sort's use, here's another listing that will sort much faster since it does not have to print to screen each time a swap is made:

```
Bubble.Sort:
CLS
REM *****
REM Input Buffer
REM *****
INPUT "Number of strings to sort ";N%
DIM A$(N%+1)
FOR X=1 TO N%
  PRINT "String #";X;
  INPUT A$(X)
NEXT X
Z=X
T=X-1
CLS
Bubble.Algorithm:
REM *****
REM Sort Strings
REM *****
Compare:
  Flag=0
  FOR S=1 TO T
    IF A$(S) <= A$(S+1) THEN Get.Next.String
    SWAP A$(S),A$(S+1)
    Flag=1
    T=S
  Get.Next.String:
  NEXT S
  IF Flag=1 THEN Compare

Out.Put.It:
REM *****
REM Out to Screen
REM *****
```

```
FOR X=1 TO Z
  PRINT A$(X)
  Count = Count + 1
  IF Count > 18 THEN Count = 0 : GOSUB Hold.Screen
NEXT X
END
Hold.Screen:
  COLOR 0,1
  PRINT "Hit any key";
  Hit$=INPUT$(1)
  COLOR 1,0
  CLS
  RETURN
```

Further on in this chapter, we will see how the bubble sort can be integrated into a database program.

## THE SHELL SORT

---

This next sort algorithm can sort lists about four times as fast as the bubble sort. Like the bubble sort, it compares string arrays (or numbers) and substitutes places in the array if one is out of order relative to the one with which it is compared. However, the sort does not continuously run through the loop comparing strings next to one another in order. Instead, it makes bigger jumps so that if the top of the order happens to be on the bottom, it will be placed in the correct position much faster than it would be by the bubble sort. Run the first example to see how the appearance of arrangement is different from the bubble sort. The list on the left will have gaps in it, but it shows how the resorting takes place. It happens so fast that it is hard to see what's going on. Use a long list of words to best see the sorting. The list on the right is simply a fully sorted list with the gaps filled in.

```
Shell.Sort:
CLS
REM *****
REM Input Buffer
REM *****
```

```

INPUT "Number of strings to sort ";N%
DIM A$(N%+1)
FOR X=1 TO N%
  PRINT "String #";X;
  INPUT A$(X)
NEXT X
T=X-1
CLS
Shell.Algorithm:
REM *****
REM Sort Strings
REM *****
N=N%
L=(2^INT(LOG(N)/LOG(2)))-1
Start:
L=INT(L/2)
IF L<1 THEN Sorted.Output
FOR J=1 TO L
  FOR K=J + L TO N STEP L
    I=K
    T$=A$(1)
    Compare:
    IF A$(I-L) <= T$ THEN Substitute
    A$(1)=A$(I-L)
    I=I-L
    IF I > L THEN Compare
    Substitute:
    A$(1)=T$
    LOCATE 1,1 : PRINT T$
  NEXT K
NEXT J
GOTO Start

Sorted.Output:
FOR X=1 TO T
  LOCATE X,20
  PRINT A$(X);
NEXT X

```

Now, let's put it in a program that will better show the speed of the sort and you can use:

```
Shell.Sort:
CLS
REM *****
REM Input Buffer
REM *****
INPUT "Number of strings to sort (at least 4) ";N%
DIM A$(N%+1)
FOR X=1 TO N%
  PRINT "String #";X;
  INPUT A$(X)
NEXT X
Z=X
T=X-1
CLS
Shell.Algorithm:
REM *****
REM Sort Strings
REM *****
N=N%
L=(2^INT(LOG(N)/LOG(2)))-1
Start:
L=INT(L/2)
IF L<1 THEN Sorted.Output
FOR J=1 TO L
  FOR K=J + L TO N STEP L
    I=K
    T$=A$(I)
    Compare:
    IF A$(I-L) <= T$ THEN Substitute
    A$(I)=A$(I-L)
    I=I-L
    IF I > L THEN Compare
    Substitute:
    A$(I)=T$
  NEXT K
NEXT J
GOTO Start

Sorted.Output:
FOR X=1 TO T
  PRINT A$(X)
```



```
Count=Count + 1
IF Count > 17 THEN Count=0 : GOSUB Hold.Screen
NEXT X
END
Hold.Screen:
Color 0,1
PRINT "Hit any key";
Hit$=INPUT$(1)
COLOR 1,0
CLS
RETURN
```

Having seen the relative advantages in speed, you can see the importance of using the appropriate algorithms. However, it is equally important to understand that *any* algorithm that gets the job done is the first priority of programming. As you become more adept at programming and working out your own algorithms, you will become better at making more efficient ones. Furthermore, you should remember to save a good algorithm on your disk as a routine to be incorporated into your other programs when needed.

## REARRANGEMENTS FOR SORTS

---

Besides sorting lists of strings by themselves, it is often important to sort information that accompanies strings. For example, if you have a list of names and addresses, how do you sort by name and get the addresses, cities and other information to accompany the sorted order of the names if they are in separate strings?

To see how to develop algorithms for rearranging data for “bundled” sorting, let’s start with something simple. Suppose you have a list of names and phone numbers in the arrays NA\$ and PH\$. You want to put the names in alphabetical order with the phone numbers, but you want to keep the strings separate. Here’s what to do:

1. First concatenate NA\$ and PH\$ with the phone number on the right.
2. Sort the list.
3. Separate the individual strings in the sorted list using RIGHT\$.

For example:

Before sort:

NA\$(5) = Wilson Betty

PH\$(5) = 555-4321

Concatenate:

A\$(5) = NA\$(5) + PH\$(5)

Once the sort is complete, we will assume that A\$(5) is moved to A\$(30).

A\$(30) = Wilson Betty 555-4321

L=8 [Number of characters in the phone number]

LS=LEN(A\$(30)) [Number of characters in entire string.]

NA\$(30) = LEFT\$(A\$(30),LS-L)

PH\$(30) = RIGHT\$(A\$(30),L)

Let's write a program that does all that and a couple of other things as well. Since we want to sort by the last name, but it feels more natural to enter the first and then the last name, we will have the computer reverse the order when it sorts the information and then change it back to the first-last order:

Input.Buffer:

```
INPUT "How many names to enter ";N%
```

```
DIM NF$(N%+1),NL$(N%+1),PH$(N%+1)
```

```
FOR X=1 TO N%
```

```
  INPUT "First name"; NF$(X)
```

```
  INPUT "Last name"; NL$(X)
```

```
  INPUT "Phone number (XXX-XXXX)";PH$(X)
```

```
  PRINT
```

```
NEXT X
```

Z=X

T=X-1

Rearrange:

```
FOR X= 1 TO N%
```

```
A$(X) = NL$(X)+"*"+NF$(X)+PH$(X)
```

```
NEXT X
```

```
Sort:
Compare:
  Flag=0
  FOR S=1 TO T
    IF A$(S) <= A$(S+1) THEN Get.Next.String
    SWAP A$(S),A$(S+1)
    Flag=1
    T=S
  Get.Next.String:
  NEXT S
  IF Flag=1 THEN Compare

Undo.Algorithm:
L=8
FOR X=2 TO Z
  LS=LEN(A$(X))
  PH$(X)=RIGHT$(A$(X),L)
  N$=LEFT$(A$(X),LS-L)
  GOSUB Find.Asterisk:
  NL$(X)=LEFT$(N$,P-1)
  LL=LEN(NL$(X))
  LN=LEN(N$)
  LF=LN-LL-1
  NF$(X)=RIGHT$(N$,LF)
NEXT X

Out.Put.It:
FOR X=2 TO Z
  PRINT NF$(X);SPACE$(1);NL$(X)
  PRINT "Phone: ";PH$(X)
  PRINT
  Count = Count + 1
  IF Count > 9 THEN Count = 0 : GOSUB Hold.Screen
NEXT X
END

Hold.Screen:
COLOR 0,1
PRINT "Hit any key";
Hit$=INPUT$(1)
COLOR 1,0
```

```

CLS
RETURN

Find.Asterisk:
FOR F=1 TO LEN(N$)
  IF MID$(N$,F,1)="*" THEN P=F
NEXT
RETURN

```

'Undo.Algorithm' in the above program is fairly involved and it takes up processing time. Another way to do the same thing is to insert "carriage returns" in the strings in the form of a CHR\$. The ASCII code for a carriage return is 13, so by concatenating CHR\$(13) in the combining string to be sorted, we can kill two birds with one stone. On the one hand, it is possible to format the output as desired, yet we can also sort on the name and not scramble the rest of the information. This gives us the best of both worlds and a much simpler algorithm.

While we're showing another algorithm for scrambling, sorting and unscrambling strings, we'll see how to put it in a sequential file. The sorting algorithm will "shuffle in" all new information added to the file so that whether you build a new file or add to an existing one, this program will sort it for you. It can be used as a handy file for keeping a list of sorted names and addresses. (Change PRINT to LPRINT and it can be used to send the information to your printer. You can use it to label all your party invitations!) Let's first take a look at the program and then see how it works:

```

GOSUB Define
Get.File.Name:
  INPUT "Name of file ";Fame$
  PRINT "(N)ew or (E)xisting File";
  Ne$=INPUT$(1)
  Ne$=UCASE$(Ne$)
  IF Ne$ <> "N" AND Ne$ <> "E" THEN Get.File.Name
  PRINT
  INPUT "How many names ";N%
  IF Ne$="E" THEN GOSUB Old.One
  IF Ne$="N" THEN L%=N%:DIM A$(L%)

Input.It.All:
  DIM NF$(L%),NL$(L%),AD$(L%),CT$(L%),SA$(L%),ZIP$(L%)
  CLS

```

```
FOR X=(1+Count) TO N% +Count
  INPUT "First Name ";NF$(X)
  NF$(X)=SPACE$(1)+NF$(X)
  INPUT "Last Name ";NL$(X)
  INPUT "Address";AD$(X)
  INPUT "City";CT$(X)
  INPUT "State";SA$(X)
  INPUT "Zip Code";ZIP$(X)
NEXT X
T=X-1
Z=X
PRINT
FOR X=(1+Count) TO N% +Count
  REM Insert Carriage returns and spaces
  A$(X)=NL$(X)+NF$(X)+CR$+AD$(X)+CR$+CT$(X)+", "
  A$(X)=A$(X)+SA$(X)+SP$+ZIP$(X)
NEXT X
```

Sort:

```
'This is the optimum use of a bubble sort
'since the file will be re-sorted each time
'the program is run. Therefore, it is doing
'a partial sort each time after the first
'rather than a full sort. The bubble sort
'is the best type of sort for this kind of
'sorting problem.
```

Compare:

```
Flag=0
FOR S=1 TO T
  IF A$(S) <= A$(S+1) THEN Get.Next.String
  SWAP A$(S),A$(S+1)
  Flag=1
  T=S
```

Get.Next.String:

```
NEXT S
IF Flag=1 THEN Compare
```

Write.File:

```
REM Sorted file is written to disk.
OPEN Fame$ FOR OUTPUT AS #1
FOR X=2 TO Z
```

```
PRINT #1, A$(X)
NEXT X
CLOSE #1
END
```

Old.One:

```
CLS
OPEN Fame$ FOR INPUT AS #1
WHILE NOT EOF(1)
  INPUT #1,A$,B$,C$,D$
  Count = Count + 1
WEND
CLOSE #1
L%=Count+N%+1
DIM A$(L%)
OPEN Fame$ FOR INPUT AS #1
FOR X=1 TO Count
  INPUT #1,A$,B$,C$,D$
  REM Insert Carriage returns and spaces
  A$(X)=A$+CR$+B$+CR$+C$+" "+SP$+D$
  PRINT A$(X) : REM Change to LPRINT for printer output.
  PRINT
  ScreenFill=ScreenFill+1
  IF ScreenFill=4 THEN ScreenFill=0
  IF ScreenFill=0 THEN GOSUB Hold.It
NEXT X
CLOSE #1
KILL Fame$
KILL Fame$+".info"
SumFiles=X
RETURN
```

Define:

```
CR$=CHR$(13) : REM Carriage Return
SP$=SPACE$(1)
RETURN
```

Hold.It:

```
COLOR 0,1
PRINT " Hit any key ";
```

```
COLOR 1,0
PRINT
A$=INPUT$(1)
RETURN
```

The algorithm for inserting the “carriage return” is in both the algorithm for sorting the original material and the material being read from the disk. Note how much simpler the routine is than the one used in the previous program. However, also note that we left the names in the order of last name first. Thus, while there are certain advantages in the simplicity of the second algorithm, it does not give the same amount of control as the first.

## ARTIFICIAL INTELLIGENCE AND IF . . . THEN . . .

---

Up to this point we’ve used only the most simple IF . . . THEN . . . ELSE structure. On your Amiga BASIC, you can also use “block” and “nested” structures with conditionals. As the use of artificial intelligence becomes increasingly important in the use of computers and robots driven by artificial intelligence software, these conditional structures are equally more important. (Artificial intelligence is called ‘AI’ and pronounced “aay eye” by the hip practitioners of the art. To make the right impression, just say “aay eye” and you’re in like Flint.)

To get started, take a look at the following program that will tell what kind of grade you’ll get by entering a number:

```
INPUT "Enter a test score "; N
Block 1:
  IF N >59 THEN
    IF N >69 THEN
      PRINT "You're Passing and above a D"
      PRINT "That's nice"
    ELSE
      PRINT "You're below a C"
    END IF
  ELSEIF N < 60 THEN
    PRINT "You're in trouble!"
  END IF
```

```
Block2:
  IF N >79 THEN
    PRINT "You're above average"
  IF N >89 THEN
    PRINT "You're hot!"
  ELSE
    PRINT "You have a B"
  END IF
END IF
```

In a block structure, each IF statement requires an END IF statement at the end of the block just like each FOR statement needs a NEXT statement. Using the block structure means that you can have several lines as the consequence of a true condition. For example, in Block1, there are two PRINT statements following the second IF statement. Without the block structure, you would have to put everything on the same line. Also, you can insert the ELSEIF statement that allows an additional condition with a single END IF.

In Block2 you can better see the “nested” character of the block structure being used. Both END IF statements are together at the end of the block satisfying the requirement of one END IF statement for each IF statement. With a nested block structure, if the outside (first) IF statement is not true, then the inner (second, etc.) IF statement will not be executed even if the condition of its test is true. For example, the second condition in the following program could never be executed because if it is true the first one would have to be false:

```
INPUT "Enter a number";N
IF N > 100 THEN
  PRINT "You can see this"
  IF N < 100 THEN
    PRINT "But you will never get this"
  END IF
END IF
```

Now that you have an idea of how the block conditional structure works, let's take a look at a simple example of artificial intelligence. We'll use the voice synthesizer to make a simulation of a program that appears to “think” or understand what a person wants. Even for this simple example, it takes a lot of programming since your Amiga will have to



make a lot of conditional branches. (It will also begin to give you an idea of how smart people really are!) We will use a “tree structure” that will lead to different directions depending on the response of the user. If we had an input that could read voice it would be even more exciting, but try this program out anyway. It simulates a computer running a very small book store trying to help a customer:

```
SAY TRANSLATE$("Hello, may I help you?")
GOSUB Voices
Instruct:
  SAY TRANSLATE$("Please respond with the keyboard")
  SAY TRANSLATE$("Type, Y, for Yes")
  SAY TRANSLATE$("N, for no.")
  SAY TRANSLATE$("Do you understand")
  GOSUB Respond
  IF A$="N" THEN Instruct

Category1:
  SAY TRANSLATE$("Good. I will ask more questions.")
  SAY TRANSLATE$("You will respond with a Y or N.")
  SAY TRANSLATE$("Would you like fiction?")
  GOSUB Respond
  IF A$="Y" THEN
    SAY TRANSLATE$("Good. I like it too.")
    GOTO Fiction
  ELSE
    GOTO Nonfiction
  END IF

Fiction:
  SAY TRANSLATE$("Do you like row manz?")
  GOSUB Respond
  IF A$="Y" THEN GOTO Romance
  SAY TRANSLATE$("How about science fiction?")
  GOSUB Respond
  IF A$="Y" THEN GOSUB Sci.Fi
  END

Romance:
  SAY TRANSLATE$("Hello. I am in charge of row manz"),Talkf%
  SAY TRANSLATE$("I luv, a good row manz"),Talkf%
```

```
SAY TRANSLATE$("Would you like a gothic row manz?"),Talkf%
GOSUB Respond
IF A$="Y" THEN
  SAY TRANSLATE$("I like Chips on Fire"),Talkf%
  SAY TRANSLATE$("It is about two programmers"),Talkf%
  SAY TRANSLATE$("who fall in luv."),Talkf%
ELSE
  SAY TRANSLATE$("How about Historical row manz?"),Talkf%
  GOSUB Respond
  IF A$="Y" THEN
    SAY TRANSLATE$("Try Love in Silicon Valley"),Talkf%
  END IF
END IF
SAY TRANSLATE$("That is all I have.")
END
```

**Sci.Fi:**

```
SAY TRANSLATE$("I. in joy. row bot. stor eez."),Talkr%
SAY TRANSLATE$("Do you like row bot stor eez?")
GOSUB Respond
IF A$="Y" THEN
  SAY TRANSLATE$("That is a good choice.")
  SAY TRANSLATE$("We will get a long well.")
  SAY TRANSLATE$("That is all for today.")
  SAY TRANSLATE$("My chips are tired.")
  SAY TRANSLATE$("Good bye.")
ELSE
  SAY TRANSLATE$("Too bad.")
  SAY TRANSLATE$("You will miss me.")
  SAY TRANSLATE$("I am gone.")
END IF
END
```

**Nonfiction:**

```
SAY TRANSLATE$("You must like non fiction then.")
SAY TRANSLATE$("I enjoy reading computer books.")
SAY TRANSLATE$("Do you want a computer book?")
GOSUB Respond
IF A$="Y" THEN
  SAY TRANSLATE$("You already have one.")
  SAY TRANSLATE$("You got this program from it.")
  SAY TRANSLATE$("Read it again.")
```

```

ELSE
  SAY TRANSLATE$("Too bad. That's all I have")
  SAY TRANSLATE$("in non fiction.")
END IF
END

Respond:
A$=INPUT$(1)
A$=UCASE$(A$)
IF A$ <> "Y" AND A$ <> "N" THEN
  SAY TRANSLATE$("I can only respond to Y or N")
  GOTO Respond
END IF
RETURN

Voices:
Female:
FOR X=0 TO 8
  READ Talkf%(X)
NEXT X
DATA 200,0,145,1,22000,64,10,0,0

Robot:
FOR X=0 TO 8
  READ Talkr%(X)
NEXT X
DATA 200,0,160,0,22000,64,10,0,0
RETURN
DATA 200,0,160,0,22000,64,10,0,0
RETURN

```

Try changing the parameters on the voices and adding more types of books. The example is trivial, but it is fun, and you can change it to do something more practical. If you have an attachment that can read voice input, you can really do a lot with your Amiga and even simple AI.

## SUMMARY

This chapter takes us to the end of the book. Beyond this, there's still more to programming, but from here on out you're on your own! However, you should be able to do most of the things you will need to

do, and with practice, programming will be as simple as working your VCR or driving your car. Developing algorithms can be as fun and challenging as doing a crossword puzzle, and artificial intelligence is just developing as a new area of interest. For the Amiga, these tasks are possible since the computer has so much power.

As a parting suggestion for those of you who will spend time writing BASIC programs for your Amiga, let me reiterate the approach we've stressed throughout this book: keep it simple! All structures in programming can be broken down into bite-sized parts. Each part can be developed into a module, and a collection of modules can be organized into a program. None of the statements, functions or commands are inherently complex. Like computers, they are fundamentally simple. The more complex structures emerge from various parts which are combined to perform complex tasks. However, by keeping the parts distinct and the modules relatively small and separate, programming your Amiga is not only possible, it is a lot of fun too.

## APPENDIX A

### ASCII Character Codes

Dec	Hex	CHR	Dec	Hex	CHR	Dec	Hex	CHR
009	09H	HT	052	34H	4	095	5FH	–
010	0AH	LF	053	35H	5	096	60H	‘
011	0BH	VT	054	36H	6	097	61H	a
012	0CH	FF	055	37H	7	098	62H	b
013	0DH	CR	056	38H	8	099	63H	c
014	0EH	SO	057	39H	9	100	64H	d
015	0FH	SI	058	3AH	:	101	65H	e
016	10H	DLE	059	3BH	;	102	66H	f
017	11H	DC1	060	3CH	<	103	67H	g
018	12H	DC2	061	3DH	=	104	68H	h
019	13H	DC3	062	3EH	>	105	69H	i
020	14H	DC4	063	3FH	?	106	6AH	j
021	15H	NAK	064	40H	@	107	6BH	k
022	16H	SYN	065	41H	A	108	6CH	l
023	17H	ETB	066	42H	B	109	6DH	m
024	18H	CAN	067	43H	C	110	6EH	n
025	19H	EM	068	44H	D	111	6FH	o
026	1AH	SUB	069	45H	E	112	70H	p
027	1BH	ESCAPE	070	46H	F	113	71H	q
028	1CH	FS	071	47H	G	114	72H	r
029	1DH	GS	072	48H	H	115	73H	s
030	1EH	RS	073	49H	I	116	74H	t
031	1FH	US	074	4AH	J	117	75H	u
032	20H	SPACE	075	4BH	K	118	76H	v
033	21H	!	076	4CH	L	119	77H	w
034	22H	”	077	4DH	M	120	78H	x
035	23H	#	078	4EH	N	121	79H	y
036	24H	\$	079	4FH	O	122	7AH	z
037	25H	%	080	50H	P	123	7BH	{
038	26H	&	081	51H	Q	124	7CH	
039	27H	’	082	52H	R	125	7DH	}
040	28H	(	083	53H	S	126	7EH	~
041	29H	)	084	54H	T	127	7FH	DEL
042	2AH	*	085	55H	U			

Dec=decimal, Hex=hexadecimal (H), CHR=character, LF=LineFeed, FF=FormFeed, CR=Carriage Return, DEL=Rubout

Non-ASCII Character Codes

b.	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1				
b.	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1				
b.	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1				
b.	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1				
b.	b.	b.	b.		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
0	0	0	0	00			SP	0	@	P	`	p			NBSP	°	À	Ð	à	Ö
0	0	0	1	01			!	1	A	Q	a	q			i	±	Á	Ñ	á	ñ
0	0	1	0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò
0	0	1	1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó
0	1	0	0	04			\$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô
0	1	0	1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ
0	1	1	0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö
0	1	1	1	07			'	7	G	W	g	w			§	.	Ç	×	ç	×
1	0	0	0	08			(	8	H	X	h	x			¨	,	È	Ø	è	ø
1	0	0	1	09			)	9	I	Y	i	y			©	ı	É	Ù	é	ù
1	0	1	0	10			*	:	J	Z	j	z			ª	º	Ê	Û	ê	û
1	0	1	1	11			+	;	K	[	k	{			«	»	Ë	Ü	ë	ü
1	1	0	0	12			,	<	L	\	l				¬	¼	Ì	Û	ì	ü
1	1	0	1	13			-	=	M	]	m	}			SHY	½	Í	Ý	í	ý
1	1	1	0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ
1	1	1	1	15			/	?	O	-	o				™	¿	Ï	Ë	ï	ÿ

## APPENDIX B USING CLI

### Command Line Interface

On your Workbench disk you have probably noticed the CLI icon. This is an alternative operating system you can use to control your Amiga. CLI stands for "Command Line Interface." It is especially useful for handling DOS (Disk Operating System), and while it is more awkward than using the mouse and icons, it is very powerful. This is a *brief* introduction to CLI and the special commands it has. We will concentrate on things that can be done better with CLI than with the icon system. To learn more about CLI, read Commodore's *AmigaDOS User's Manual*.

Open the CLI icon, and you will open the CLI Window. The following prompt will appear:

```
1>
```

Next to that prompt is where you enter your CLI commands. To give yourself enough room to see what you're doing, widen the window to its full size using the mouse. Once you have a big enough window, enter the following:

```
1> DIR <Press Return>
```

The DIR command works like FILES in BASIC. It shows you what is on the disk.

To experiment, take a blank disk and place it in the external disk drive. (If you have only a single disk drive, this will be a bit tedious, so just make a copy of your Workbench disk and use it.) To prepare the disk from CLI, use the FORMAT command *exactly* in the following sequence (*do* include the words DRIVE and NAME):

```
1>FORMAT DRIVE DF1: NAME ''CLIDISK'' <Press Return>
```

Your Amiga will instruct you to:

```
Insert disk to be formatted and press RETURN
```

When completed you will see:

```
Format cyl 79  
Verify cyl 79  
Disk '*CLIDISK*' formatted and initialized
```

Now to make it a disk that can be “booted” (start up your Amiga), type in:

```
1> INSTALL DF1:
```

After you get the prompt back, you're all set. You could have used any formatted disk, and it is not required to have a special CLI disk, but since there's a good chance of erasing information on the disk, it's better to be safe than sorry. To exit CLI, just type in ENDCLI and press RETURN.

## The CLI Commands

The language of CLI is a separate one from BASIC, and we will only present a short glossary of these commands here along with some brief examples of their use. This is not meant to be a full explanation of CLI but rather a quick reference to CLI commands.

### **ASSIGN Name: Dir/dir/file**

This is a temporary assignment of a logical device name to a filing system directory. Its most practical use is in quickly getting to a file in a subdirectory. For example if you have a file in DF1: subdirectory SORTS with the name BUBBLE you would have to use DF1:SORTS/BUBBLE to pull it out. With ASSIGN, you can issue a single word command to get your file.

Example:

```
ASSIGN SORTO: DF1:SORTS/BUBBLE
```

Now any reference to SORTO: will pull out BUBBLE for you.



**BREAK Process #,Ctrl Char**

The BREAK command sets flags to break at a given process number with control keys C-F. BREAK ALL sets all Ctrl (C-F) flags.

Example:

```
BREAK 3 D
```

sets CTRL-D attention flag to process 3.

**CD Dir:Sub/Sub**

This sets the Current Directory to the drive and subdirectory(ies) specified.

Example:

```
CD DF1:CH19/SORTS
```

would set the current directory to SORTS in drive 1 in subdirectory CH19.

**COPY File TO Dsk/Dir**

Copies file or files from current OR specified drive to target disk and directory. The word ALL as file name will copy all files.

Example:

```
COPY df0:shell to df1:sorts/shell
```

would copy the file 'shell' from drive 0 to drive 1 in the subdirectory called 'sorts.' The existing file called 'shell' in drive 1 would be replaced by the one from drive 0.

**DATE Day-month-year (or Hour:Min)**

Sets date to specified day-month-year or time. Using TO option sends current set date to specified file. The word TOMORROW increments date by one day.

Example:

```
DATE 17-May-88
```

would set date to May 17, 1988.

DATE 10:45

would set time to 10:45.

### **DELETE Filename**

Works like KILL in BASIC. It deletes a single or multiple files up to 10.

Example:

```
Delete sdump.bku
```

would remove a file named 'sdump.bku.'

```
Delete df0: #? ALL
```

would remove all files from drive 0.

### **DIR [drive:subdirectory]**

Displays the contents of current drive and/or specified drive and subdirectory.

Example:

```
Dir df1:ch19
```

would display on the screen all of the files in the subdirectory 'ch19' on drive 1.

### **DISKCOPY FROM drive TO drive**

Copies the entire disk.

Example:

```
DISKCOPY FROM df1: to df0
```

would copy the disk in drive 1 to drive 0.

### **ECHO \$**

Displays argument on screen. Used in a stream of commands with RUN.

Example:

```
Echo "Current drive"
```

would print that message on screen.

**ED Filename {Size}\***

Uses built-in text editor to edit files.

Example:

```
ED Names\Friends
```

would edit a file called "Friends" in the subdirectory "Names."

*\*See Chapter 3 of AmigaDOS User's Manual for instructions on using this editor.*

**EDIT Filename\*\***

Edits with built-in line editor text files.

*\*\*To use this editor see Chapter 4 of the AmigaDOS User's Manual.*

**ENDCLI**

Quits CLI and returns to icon system.

Example:

```
ENDCLI
```

**EXECUTE Commandfile**

Acts as automatic command executer of commands in a given file with or without arguments. Used for executing "DOS Programs" written with the ED editor.

Example: Enter the editor, using

```
ED Test 1<RETURN>
```

Once in the editor, type in:

```
ECHO "This is a test of execute" <Press RETURN>
```

```
DIR df0: <Press RETURN>
```

```
DIR df1: <Press RETURN>
```

Press the ESC key, and when the asterisk appears at the bottom of the screen, type an "X" and press RETURN. The file will be saved to disk. When you see the 1> prompt, you can EXECUTE the file "Test 1" by typing:

Execute test 1

All of the commands you put in the file will now be executed. See your *AmigaDOS User's Manual* for details.

### **FAILAT N**

Sets error code where command sequence will stop execution if code level is exceeded. Error codes are usually 5, 10 or 20, and so a level should not be too great above 20. The default limit is 10.

Example:

```
FAILAT 15
```

### **FAULT N**

Displays fault code message.

Example:

```
Fault 111
```

would return Fault 111:Error 111.

### **FILENOTE Filename COMMENT String**

Attaches a note to specified file.

Example:

```
FILENOTE test 1 comment "This is an example of execute"
```

would show

```
test 1
:This is an example of execute
```

when the LIST command is given.

### **FORMAT DRIVE df NAME string**

Prepare a disk for use in the Amiga format.

Example:

```
FORMAT Drive df1: name "Filedisk"
```

would format a disk in drive 1 with the name "Filedisk."

**IF Condition Action ENDIF**

Works something like the IF statement in BASIC except it works with a command list. Words NOT, WARN, ERROR, FAIL EXISTS and EQ are conditionals.

Example:

```
IF EXISTS test1
EXECUTE test1
ELSE
ECHO "It's not here Jack"
ENDIF
```

**INFO**

Returns information about files.

Example:

```
INFO
```

would show status of both drives.

**INSTALL Drive**

Makes a formatted disk able to boot Amiga.

Example:

```
INSTALL df1:
```

**JOIN File1 File2 AS Filen**

Combines several files into large single file.

Example:

```
JOIN test1 test2 test3 AS combinefile
```

would join files named 'test1,' 'test2' and 'test3' into single file called 'combinefile.'

**LAB String**

Makes labels in sequence of command. Works something like module labels in BASIC programs.

Example:

```
[In command sequence created in ED.]
IF EXISTS test1
SKIP DOIT
ELSE
ECHO "It's not here Jack"
ENDIF
LAB DOIT
EXECUTE test1
```

would jump to line LAB DOIT in true condition.

### **LIST Dir**

Returns information about file or directory without sorting.

Example:

```
LIST df1:test 1
```

would show information about file named 'test1' on drive 1.

### **MAKEDIR Dir.Name**

Creates a new subdirectory.

Example:

```
MAKEDIR df1:ch19/sorts
```

would create a subdirectory 'sorts' in the subdirectory ch19 on drive 1.

### **NEWCLI CON:X,Y,W,H,T**

Creates a new CLI window beginning at horizontal X, vertical Y at W width and H height. The T is the window's title.

Example:

```
NEWCLI CON:10/10/100/100/Bilzcli
```

makes a new window in the upper left hand corner of the screen called "Bilzcli." To get rid of the window, use ENDCLI while in that particular CLI window. It will not affect the first CLI window.

**PROMPT %N**

Changes the current prompt in the current CLI window.

Example:

```
PROMPT "%#>"
```

would turn prompt into #>.

**PROTECT Filename R,W,D,E**

Allows user to specify which options—Read, Write, Delete and Execute—the user does not want protected.

Example:

```
PROTECT test2 rwe
```

would allow everything except deletion of test2.

**QUIT Errorcode**

Exits command sequence if a certain error code is encountered.

Example:

```
QUIT 30
```

If a return code of 30 is encountered, then terminate.

**RELABEL Drive Diskname**

Changes the volume name of a disk.

Example:

```
RELABEL df0: "workshop"
```

would change the current name of the disk in drive 0 to 'Workshop.'

**RENAME NameOld AS NameNew**

Renames a file.

Example:

```
RENAME test1 AS test2
```

would change the name of the file 'test1' to 'test2.'

**RUN Command1 + Command2**

Allows putting multiple commands together in single module from immediate execution mode.

Example:

```
RUN dir df1: +  
dir df0:
```

would display first the directory of df1 and then of df0.

**SEARCH File string**

Returns line with search string in files.

Example:

```
SEARCH Sort "six"
```

would return the line the word "six" was found on in the file 'Sort'

**SKIP Label**

Jumps to labeled line. Usually used as a conditional jump.

Example: See LAB example above.

**SORT Filename TO Filename2**

Sort the contents of a given file into a specified new file.

Example:

```
SDRT Test1 TO TestSorted
```

will alphabetize the contents of Test1 and write the results to disk in the file TestSorted.

**STACK {N}**

Either shows stack size or sets stack size.

Example:

```
STACK
```

would show stack size.

```
STACK 6000
```

would set stack to 6000 bytes.



**STATUS N {FULL}**

Returns information about CLI processes currently existing.

Example:

```
STATUS FULL
```

would return the stack size, global vector and CLI command.

**TYPE Filename**

Displays contents of file. Typically used with sequential file to read it for information. Good for reading files created with ED.

Example:

```
TYPE test2
```

would display the contents of the file “test 2” to screen, but it would not execute any commands that may be in the file.

**WAIT N MINS {UNTIL HH:MM}**

Stops execution until given time or amount of time in seconds.

Example:

```
WAIT 3 MINS
```

would stop execution for three minutes.

```
WAIT UNTIL 10:45
```

would stop execution until quarter of eleven.

**WHY**

Explains the return code.

Example:

```
1>asd <RETURN>
```

```
Unknown command asd
```

```
1>WHY <RETURN>
```

```
Last command failed because object not found
```



## BASIC GLOSSARY

This glossary is designed as a quick reference for using the many statements, functions and commands you have learned in this book. In some cases requiring a lengthy explanation, the reader is referred to the body of the book for further discussion of the keyword. The examples are to show format rather than to provide detailed usage.

**ABS()** Gives the absolute value of a number or variable.

```
PRINT ABS(123.45)
```

**AND** Logical operator used in equations (assignments) and logical expressions.

```
IF A$ <> "Y" AND A$ <> "N" THEN GOTO Find
ON A$= "Y" AND SUM AND CT GOTO Terminate
A = A$ = "Y" AND B$ = "Y"
```

**APPEND** Adds data to end of existing sequential text file.

```
OPEN "TEXT.TXT" FOR APPEND AS #1
```

**AREA {STEP} (X,Y)** Specified point to be part of polygon that can be drawn with AREAFILL.

```
AREA (10,20)
AREA STEP (0,9)
```

**AREAFILL** Fills the inside of a polygon specified by the last two AREA statements. Mode 0 fills with PATTERN and 1 inverts filled area.

```
AREAFILL
```

**ASC()** Returns ASCII value of first character in string.

```
PRINT ASC ("W") or A$ = "Amiga" : PRINT ASC(A$)
```

**ATN()** Returns arctangent of number or variable.

```
PRINT ATN (333)
```

**BEEP** Emits bell sound.

```
IF AN$ <> "Y" AND AN$ <> "N" THEN BEEP
```

**BREAK ON {OFF,STOP}** Turns BREAK event trap on, off or suspends it.

```
BREAK ON  
ON BREAK GOSUB Hold.It
```

**CALL** Goes to machine subroutine at a given OFFSET from DEF SEG with argument of variable passed to subroutine.

```
DEF SEG = &H4000 : A = 0 : CALL A (V$)
```

**CDBL()** Changes variable to double precision number.

```
X = 10.7 : Y# = CDBL(X)  
PRINT Y#
```

**CHAIN** Preserves and passes variables from one program to another.

```
CHAIN "df1:Sort"
```

**CHR\$()** Returns the character represented by given ASCII value.

```
IF EF$ <> CHR$(13) THEN PRINT
```

**CINT()** Converts number into integer. Rounds up if fractional value is 0.5 or greater. Rounds down if fractional value is less than 0.5.

```
X = 5.49  
PRINT CINT(X)
```

**CHDIR** Specifies current directory, including subdirectory.

```
CHDIR "df1:CH19"
```

**CIRCLE (X,Y),R,C,S,E,A** Draws a circle beginning at X,Y with radius of R. Optionally, the color C, starting angle S, ending angle E and aspect A may be included.

```
CIRCLE (100,100),20
```

**CLEAR {data,stack}** All variables and arrays are reset to zero.

```
CLEAR  
CLEAR,1000,200
```

**CLOSE** Closes specified OPEN file.

```
CLOSE #1  
CLOSE [Closes all OPEN files]
```

**CLS** Clears screen and places cursor in upper left-hand corner of screen.

```
CLS
```

**COLLISION(OS)** Returns collision information with an OBJECT.SHAPE (OS). A negative number from -1 to -4 indicates collision with one of the borders, top, left, bottom, or right respectively.

```
X=COLLISION(N)  
X=ABS(X)  
ON X GOSUB Top,Left,Bottom,Right
```

**COLLISION {ON,OFF,STOP}** Sets up enable, disable or suspension of COLLISION.

```
COLLISION ON
```

**COLOR F,B** Sets (F)oreground, and (B)ackground color in relationship to PALETTE color established with PALETTE statement.

```
COLOR 0,1
```

**CLNG()** Changes numeric variable into long-integer format.

```
A&=CLNG(A%)
```

**COMMON** Used with CHAIN command to pass variables between programs.

```
COMMON A$,B,C%  
CHAIN "Names.Numbers"
```

**CONT** Continue program after a STOP or Amiga-period has been detected.

```
CONT
```

**COS()** Returns the cosine of variable or number.

```
C=COS(321)
```

**CSNG()** Changes variable or value to single-precision.

```
X# = 10.232221233#
PRINT CSNG(X#)
```

**CSRLIN** Returns the vertical location of the current cursor.

```
LOCATE 20,1 : V = CSRLIN : PRINT V
```

**CVI, CVS, CVD** Changes strings into numeric variables of integer, single or double precision. Generally used in random access files to convert stored strings into numeric variables.

```
FIELD #1 5 AS A$, 7 AS B$, 20 AS C$
GET #1
A%= CVI (A$)
B = CVS (B$)
C# = CVD (C$)
```

**DATA** Strings or numbers to be READ.

```
DATA 17857,Rancho Bernado, "Play it again, Sam"
```

**DATE\$** Special string to be defined from Workbench 'Preference' file as month/day/year.

```
D$ = DATE$ : PRINT D$
```

**DECLARE FUNCTION** (This advanced machine language function was not covered.) Used to call up value in given library and returns parameters in function specified.

**DEF FN()** Defines a function for simple real variable.

```
DEF FN A(X) = X * X
PRINT FN A(5)
[Result = 25]
```

**DEF SEG** Sets address of current segment of memory. That address is set at zero (0) for offset in subsequent address access.

```
DEF SEG = &HBB00
POKE 0,65 : POKE 1,10
```

**DEF {INT,LNG,SNG,DBL,STR}** Defines variables beginning with given characters as being integer, long integer, single precision, double precision or strings, respectively. Defined variables do not require sign after variable.

```
DEFSTR B
Bear="Bear"
PRINT Bear

DEFINT A-M
INPUT "Number attending meeting";L
```

**DELETE** Deletes line, range of lines or labeled line.

```
DELETE 40-90
DELETE Dataline
```

**DIM** Allocates maximum range of array.

```
DIM A$ (100)
```

**END** Terminates running of program and exits to Immediate mode.

```
END
```

**EOF()** Sets flag to -1 if end of file has been found and to 0 if not.

```
WHILE NOT EOF(1)
```

**ERASE** Specified array is erased.

```
ERASE Names$
```

**ERL** Variable for line with error.

```
IF ERL = 50 THEN Handle.Error
```

**ERR** Variable for error code.

```
IF ERR = 10 THEN Err.Check
```

**ERROR N%** Simulates error.

```
ERROR 11 [Division by zero error simulated]
```

**EXP()** Returns e to indicated power.

```
PRINT EXP (7)
```

**FIELD** Specifies space for variable in random access files.

```
FIELD #1, 1 AS A$, 9 AS V$
```

**FILES** Shows files on disk.

```
FILES <RETURN>  
FILES "df1:" <RETURN>
```

**FIX** Returns integer of number.

```
PRINT FIX (324.65)  
[Result = 324]
```

**FOR/NEXT/STEP** Sets up loop with specified bottom and top limit incremented or decremented by optional STEP at NEXT.

```
FOR X= 5 TO 500 STEP 50  
PRINT X  
NEXT X
```

**FRE()** Returns available memory.

```
PRINT FRE(0)
```

**GET {Files}** Reads record from specified random file.

```
GET 1
```



---

**GET(Xa,Ya)-(Xb,Yb),A {Graphics}** Reads the special graphics array in terms of the specified coordinates as the upper left- and lower right-hand corners of a screen area.

```
GET(1,1)-(8,8),C%
```

**GOSUB/RETURN** Branches to subroutine at given line number and comes back to the next line number after the GOSUB after encountering RETURN.

```
GOSUB Define
PRINT A$
....
Define:
A$ = "Amiga"
RETURN
```

**GOTO** Branches to given line label or number.

```
GOTO Center.It
GOTO 200
```

**HEX\$(D)** Returns the hexadecimal value of given decimal number, D.

```
PRINT HEX$(10)
```

**IF—THEN—ELSE** Sets up conditional logic for execution.

```
IF Query$ = "M" THEN Start ELSE END
```

**IF/THEN/ELSE/ELSEIF/END IF** Block conditional.

```
IF A$ = V$ THEN
  A=10
  B=55
  ELSEIF A$=G$ THEN
  A=9
  B=88
END IF
```

**INKEY\$** Reads single character from keyboard input.

```
Get.Key:
  AN$ = INKEY$ : IF AN$ = "" THEN Get.Key
  IF AN$ = "E" THEN END
```

**INPUT** Halts program execution until string or numbers entered and RETURN key is pressed. May enter message within INPUT statement.

```
INPUT "First word-> "; Vocab$(X)
INPUT "Enter single dimension number-> "; S
INPUT "ENTER INTEGER NUMBER -> "; N%
PRINT "Press RETURN to continue";
INPUT CR$
```

**INPUT#** Reads data from OPEN file or specified device.

```
INPUT #1, Info$
```

**INPUT\$(N)** Halts execution until N number of key presses have been made. Returns string of length N from keyboard or file.

```
PRINT "Pick AB OR BA ->";
  C$ = INPUT$(2)
  IF C$ = "AB" THEN Look.Up ELSE Look.Down

  X$ = INPUT$(1,1) {from file}
```

**INSTR(A\$,B\$)** Examines B\$ for occurrence of A\$ and returns position of first character of B\$. [Optional: INSTR(P,A\$,B\$) where P equals the starting position in A\$ to begin search.]

```
FULLNAME$ = "Jack B. Quick"
LASTNAME$ = "Quick"
N = INSTR(FULLNAME$,LASTNAME$)
PRINT MID$(FULLNAME$,N)
```

**INT()** Returns integer value of number or variable.

```
PRINT INT(98.76)
```

**KILL** Deletes file from disk. Extender must be used with this command.

```
KILL "Dumb.Sort"
```

**LBOUND() {UBOUND}** Finds (L)ower and (U)pper boundaries of array.

```
DIM A$(255)
PRINT LBOUND(A$),UBOUND(A$)
[Results = 0,255]
```

**LEFT\$(,)** Returns specified number of characters from a given string beginning with character at far left.

```
Bybye$ = "So Long"
PRINT LEFT$ (A$,2)
[Result = So]
```

**LEN** Returns the length in terms of number of characters of a specified string.

```
PRINT LEN(Names$)
```

**LET** Optionally used in assigning value to variables.

```
LET NoVic = 900
```

**LIBRARY** (Advanced machine language using routine not covered.) Opens machine language subprograms. Used with CALL.

**LINE (H1,V1)-(H2,V2),C,BF** Draws line from coordinates (H)orizontal (V)ertical 1 to H2,V2 in optional color C, optional (B)ox and optional (F)ill.

```
LINE (100,100)-(150,150)
LINE -(200,50)
LINE (10,10)-(80,80),1,BF
```

**LINE INPUT** Accepts entire line of up to 254 characters ignoring delimiters such as commas.

```
LINE INPUT "Last name,First name";LF$
```

**LINE INPUT #** Accepts entire line of up to 254 characters from sequential file.

```
LINE INPUT #1,LF$
```

**LIST** Lists program currently in memory to screen. Specify line label to have it listed at top of list window or range of lines for partial listing.

```
LIST
LIST compare
LIST 30-80
```

**LLIST** Lists program currently in memory to printer.

```
LLIST
```

**LOAD** Loads program specified from disk or tape. Optionally, LOAD "FILENAME",R to load and run program.

```
LOAD "LaSorta"
LOAD "LaSorta",R
```

**LOC()** Used in determining position in current 128 byte "record" in sequential file. With random access files, returns the last record number of read or write.

```
N = LOC(1)
PRINT "Record #";N
```

**LOCATE R,C** Sets (R)ow and (C)olumn of next PRINT. Places cursor at R,C.

```
LOCATE 10,5: PRINT "HERE"
```

**LOF()** Returns length of file.

```
OPEN "NameAd" AS #2
PRINT LOF(2)
CLOSE
```

**LOG()** Returns natural logarithm (to base E) of specific number or variable.

```
PRINT LOG (19)
```

**LPOS()** Returns current horizontal position of printer.

```
Pnow = LPOS(0)
IF Pnow > 50 THEN LPRINT
```

**LPRINT** Outputs information to printer.

```
LPRINT "Print this"
LPRINT Address$,K,Record%
```

**LPRINT USING** Outputs to printer with PRINT USING format. (See PRINT USING.)

```
LPRINT USING "$#####.##";Money
```

**LSET** Used in random access files for moving data into buffer. Must use different name than used with INPUT. If LSET is used, spaces pad the extra positions automatically. (See also RSET.)

```
LSET City$ = C$
```

**MENU N,ID,S, "Name"** Sets pulldown menus for use in BASIC program. Individual (N)umber for each menu, an (ID)entification for each option and a (S)tate of on, off or checked. Name is optional string. MENU(0) returns the number of last chosen menu (N), and MENU(1) returns the last chosen option number (ID). MENU RESET is used to return menu state to default. It should be used at the end of every BASIC program that uses the MENU statement.

```
MENU 2,0,1,"Sort File"
```

**MENU {ON,OFF,STOP}** Enables, disables or suspends use of MENU event trapping.

```
MENU ON
ON MENU GOSUB Menu.Select
```

**MERGE** Load ASCII program into memory without removing current program.

```
MERGE "SectionB"
```

**MID\$( , , )** Returns a portion of a string beginning with the nth character from the left for the number of characters indicated in the third position.

```
Mouth$ = "YakettyYak"
PRINT MID$(A$,3,3)
[Result = ket]
```

**MKI\$, MKL\$, MKS\$, MKD\$** Used in random access files to convert integer, long integer, single precision or double precision numeric values into strings.

```
LSET V$ = MKL$(V&)
```

**MOD** Returns the "modulo" or remainder of a division result.

```
PRINT 8 MOD 3
[Results = 2]
```

**MOUSE** Returns information about left mouse button and mouse position.

```
IF MOUSE(0) = 1 THEN GOSUB Fire
X=MOUSE(1)
Y=MOUSE(2)
PSET (X,Y)
```

**MOUSE {ON,OFF,STOP}** Enables, disables or suspends mouse event trapping.

```
MOUSE STOP
```

**NAME** Used to rename files from BASIC.

```
NAME "Part" AS "Whole"
```

**NEW** Clears program and variables in memory.

```
NEW
```

**NOT** Logical negation in logical expression.

```
IF A NOT B THEN C = R
C = NOT (D AND E)
```

**OBJECT** (See chapter on sprites for the several OBJECT. prefaced statements.)

**OCT\$(N)** Returns octal value of decimal value N.

```
PRINT OCT$(55)
```

**ON** Sets up computed GOTO, GOSUB, with MENU, MOUSE, TIMER, ERROR, BREAK, COLLISION, or variable to branch line.

```
ON A GOSUB First,Second,Third
ON ERROR GOTO Handle.Error
ON MENU(0) GOTO Get.Menu
ON TIMER GOTO Time.Set
ON MOUSE GOSUB Squeek
ON BREAK GOSUB Halt.Report
ON COLLISION GOSUB Check.Side
```

**OPEN** Accesses channel to input/output device of OUTPUT, INPUT or APPEND.

```
OPEN "Address.Book" FOR OUTPUT AS 1
```

**OPTION BASE** Set minimum array value to 0 or 1.

```
OPTION BASE 1
DIM City$(100)
FOR X=1 TO 100
  READ City$(X)
NEXT X
```

**OR** Logical OR in logical expression.

```
IF A=10 OR B = 20 THEN GOTO 190
C = D OR E OR K
```

**PAINT(H,V),C,BC** “Paints” a specified area with optional color and border color.

```
CLS
COLOR 0,1
CIRCLE (100,100), 50,2
PAINT (220,190),2,2
```

**PALETTE ID,R,G,B** Sets “paint brush” color with ID from 0–31 to intensity of (R)ed, (G)reen and (B)lue of 0–1.

```
PALETTE 3,.4,1,.31
COLOR 1,3
CLS
```

**PATTERN L,P** Creates the texture of lines and fill areas.

```
FOR X=0 TO 3
  READ Texture%(X)
NEXT
PATTERN &HFFF,Texture%
```

**PEEK {PEEKL,PEEKW}** Returns memory byte’s contents of given decimal address location (8 bit). PEEKL returns long integer word (32 bit address) and PEEKW returns short integer word (16 bit).

```
D = PEEK (3000)
IF PEEK (12300) = 5 THEN GOTO What.Next
```

**POINT (X,Y)** Returns the color in PALETTE ID value of pixel at location X,Y.

```
PRINT POINT (100,100)
```

**POKE {POKEL,POKEW}** Inserts given value in specified decimal memory location as 8 bit word. POKEL inserts as 32 bit word and POKEW inserts as 16 bit word.

```
POKE 0,10 (Sets memory location 0 to decimal value 10)
```

**POS()** Gives the current horizontal position of the cursor.

```
PRINT "Cursor here";: PRINT POS(0)
```

**PRESET [STEP] (X,Y),C** Set pixel at location X,Y to background color C. (See also PSET.)

```
PRESET 100,100
```

**PRINT** Outputs string, number, expression, function or variable to screen.

```
PRINT "Anything";N%,V$,K&
```

**PRINT USING** Outputs formatted strings or numbers to screen.

```
PRINT USING "$#####.##";33.23
```

**PRINT #** Prints (writes) output to disk file or OPEN logical file.

```
80 PRINT #1, Names$
```

**PRINT#, USING** Writes to file in PRINT USING format. (See PRINT USING).

```
PRINT #1, USING "#####.##";Bucks
```

**PSET [STEP] (X,Y),C** Turns on pixel at position X,Y with color C. Step is an optional parameter for position relative to last pixel.

```
PSET(100,55),2
```

**PTAB(H)** Sets next print position to horizontal pixel position H. (Note the different positions created by TAB( ) and PTAB( ) in example below.)

```
PRINT TAB(55) "About here"
PRINT PTAB(55) "About here"
```



**PUT (H,V),G,A {Graphics}** Sends colors to horizontal and vertical coordinates on screen in graphic array with optional action defaulted to XOR.

```
PUT (X,100),G%  
PUT (X,100),G%,AND
```

**PUT# {Files}** Used in random access files to write data to disk. File number and field number must be specified.

```
FIELD #2, 10 AS City$  
LSET C$ = City$  
PUT #2,Record%
```

**RANDOMIZE** Seeds random number generator with optional numeric value.

```
RANDOMIZE TIMER
```

**READ** Enters DATA statement's contents into variable.

```
READ Total  
READ X$(9)  
DATA 123,Harold
```

**REM** Non-executable statement. Allows remarks in program lines.

```
CR$ = CHR$(13): REM Carriage return
```

**RESUME** Goes to first statement of line where error occurred in error-handling routine.

```
ON ERROR GOTO Fix.It  
ERROR 6  
END  
Fix.It:  
PRINT "There's a problem." : RESUME
```

**RETURN** Returns program to next line after GOSUB command

```
RETURN
```

**RIGHT\$ ( , )** Returns the rightmost n characters of given string.

```
Amiga$= "Amiga" : PRINT RIGHT$(A$,2)  
[Result = ga]
```

**RND()** Generates a random number less than 1 and greater than or equal to 0 in Amiga BASIC.

```
PRINT RND(5)
INT (RND (1) * (N) + 1) (Generates whole random numbers
from 1 to N, with N being the upper limit of desired numbers.)
INT (RND*(N2+2-N1)+N1) (Generates whole random numbers from N1
to N2.)
```

**RSET** Used in random access files for moving data into buffer. Must use different name than used with INPUT. If LSET is used, spaces pad the extra positions automatically.

```
RSET CITY$ = C$
```

**RUN** Executes program in memory.

```
RUN
```

**SAVE** Records program on disk.

```
SAVE "Souls"
SAVE "df0: Money"
```

**SADD(W\$)** Returns the address of the first byte of string W\$.

```
PRINT SADD("What next")
PRINT SADD (G$)
```

**SAY Talk\$, mode-array.** Sends output to voice synthesizer. (See Chapter 14 for description of mode-array.)

```
SAY TRANSLATE$("Listen to this")
```

**SCREEN ID,W,H,D,M** Specifies current window as screen number ID, with W width, H height, D depth and M mode. (See chapter on graphics for using these options.)

```
SCREEN 1,100,100,2,1
```

**SCROLL (X1,Y1)-(X2-Y2),R,D** Scrolls the rectangle defined by X1-Y2 R pixels to the right and D pixels down. (Negative values reverse direction.)

```
SCROLL (10,10)-(20,20),10,10
```

**SGN** Returns sign of numeric value with 1 = positive, 0 = 0 and -1 = negative.

```
Sign = SGN(-22)
PRINT Sign
```

**SHARED** Specifies in subprogram what variables it shares with values from main program. These variables do not have to be passed as parameters.

```
SHARED A$,V( )
```

**SIN()** Returns the sine of variable or number.

```
PRINT SIN(123)
```

**SLEEP** Suspends program until initiating event trap is sprung to reinitiate action.

```
SLEEP
```

**SOUND F,D,[V,Vo]** Emits sound of (F)requency (20–15000 Hz) and (D)uration (0–77) with optional V volume from 0–255 and (Vo)lume 0–3 from speaker. Defaults to 0.

```
SOUND 2000,50,255,0
```

**SPACE\$(N)** String of N spaces.

```
Form$ = SPACE$(30)
PRINT "Name" $ "Invoice #"
```

**SPC()** Skips specified number of spaces in PRINT statement.

```
PRINT SPC(29); "HERE"
```

**SQR()** Returns the square root of variable or number.

```
PRINT SQR(49)
```

**STICK()** Returns horizontal and vertical coordinates of joystick.

```
STICK(0) = horizontal value joystick A
STICK(1) = vertical value joystick A
STICK(2) = horizontal value joystick B
STICK(3) = vertical value joystick B
X = STICK(0) : Y = STICK(1)
PSET X,Y
```

**STOP** Halts execution and prints line number where break occurs. (CONT command will restart program at next instruction after STOP command.)

```
STOP
```

**STR\$( )** Converts number/variable into string variable.

```
T = 123 : T$= STR$(T) : TT$= "$" + T$ + ".00"
```

**STRIG** Checks to see if joystick button has been pressed. Requires STRIG ON to activate. The STRIG function returns -1 if STRIG(0) pressed since last STRIG(0) function call, otherwise zero. STRIG(1) returns -1 if button is currently being pressed or zero if not. [On second stick STRIG(2) and STRING(3) do the same respectively.]

```
STRIG(0) ON
B = STRIG(0)
IF B THEN GOSUB Bang
```

**STRING\$(N,ASCII) or STRING(N1,S\$)** Creates a string of N length made up of ASCII values 0-255 or of S\$.

```
M$ = STRING$(1,14)
PRINT M$ " MUSIC " M$

A$ = "*"
AS$ = STRING$(10,A$)
FOR I = 1 TO 4 : PRINT AS$ : NEXT
```

**SUB Name(Params) STATIC.** Begins a subprogram. END SUB and EXIT SUB either mark the end of the subprogram or make up conditional exit of subprogram. STATIC indicates use of local variables within subprogram. Arrays specify the number of *dimensions* in the array, not elements.

```
SUB Center(A$,E$(1)) STATIC
```

**SWAP V1,V2** Switches contents (V)ariable1 with (V)ariable2.

```
SWAP A$,B$
SWAP T$(1), T$(X+1)
SWAP M%,N%
```

**SYSTEM** Returns to Workbench or CLI.

```
SYSTEM
```

**TAB()** Sets horizontal tab from within a PRINT statement.

```
PRINT TAB(20);"Position across"
```

**TAN()** Provides the tangent of number or variable.

```
PRINT TAN(K)
```

**TIME\$** Returns time as set in Workbench "Preferences."

```
PRINT TIME$  
  
Get.Time:  
TIME$  
IF VAL(RIGHT$(T$,2)) > 30 THEN 200  
GOTO Get.Time  
PRINT T$
```

**TIMER {ON,OFF,STOP}** Event trapping based on time. TIMER is function that returns value based on seconds after midnight.

```
TIMER ON  
ON TIMER(5) GOSUB There
```

**TRANSLATE\$** Transforms strings into phonemes used with SAY statement and voice synthesizer.

```
SAY TRANSLATE$ ("If I can get out of here")
```

**TRON and TROFF** Turns on trace function for display of line numbers in program execution. (Turned off with TROFF.)

```
TRON
```

**UBOUND** (See LBOUND)

**UCASE\$** Transforms string into upper case string.

```
AN$=UCASE$(AN$)
```

**VAL()** Used to convert string to numeric value.

```
ValGirl$="21"  
PRINT VAL(ValGirl$)
```

**VARPTR()** Returns address of variable.

```
K=22
PRINT VARPTR(K)
```

**WAVE, V, WD** Creates wave definition in array WD with voice V. (**Note:** a special array is created.)

```
DIM Hear(255)
FOR X=0 TO 255
  Hear(X)=X
NEXT
WAVE 2,Hear
```

**WHILE . . WEND** Loop statement that continues until untrue condition is met.

```
WHILE V$ <> "Quit"
INPUT V$
PRINT V$
WEND
```

**WIDTH** Sets screen output width to specified limit to 255.

```
WIDTH 62
```

**WINDOW N, "Name",(X1,Y1)-(X2,Y2),T,SID {WINDOW CLOSE,WINDOW OUTPUT,WINDOW(N)}** Creates a window in given X1-Y2 rectangle of given type T and screen id (SID). (See Chapter 11 for full explanation of all parameters.)

```
WINDOW 2,"Calculation",(10,20)-(100,100),15
```

**WRITE** Sends output to screen while it ignores all delimiters except commas but with no screen formatting. Commas and quotation marks act as delimiters but are printed.

```
WRITE 1,2,3 ; "Gets it all"
[Result = 1,2,3;"Gets it all"]
```

**WRITE#** Works like PRINT# with files except it does not place spaces in front of positive numbers and operates like WRITE with respect to commas and quotation marks.

```
WRITE #1,City$,State$,Invoice
```

# Index

- !, 27–28, 134
- " , 55, 59. *See also* String variables
- #, 27–28, 132–35
- \$, 29, 132–35
- %, 27–28
- &, 27–28, 134
- ' , 22–23, 64
- ( ), 38–39
- \*, 12
- +, 134. *See also* Addition
  - with INPUT statement, 51–52, 55, 109
  - with LINE INPUT statement, 243
  - with ON statement, 89
  - with PRINT statement, 20
  - with PRINT USING statement, 134
  - with WRITE statements, 59
- , 134
- /, 12
- ;, 3, 20, 80
- ;;, 20–21
- ?, 11, 52
- ?Redo from start error message, 52
- \, 12–13, 134
- ^, 12, 134
- ABS function, 40
- Acceleration, of moving sprites and bobs, 201–2
- Addition, 12, 38. *See also* Mathematical operations
- Address book program, 237–41
- ⌘ key, 8, 9, 15
- Algorithms, 293–94
  - for artificial intelligence, 307–11
  - bubble sorts, 294–98
  - rearrangements for sorts, 301–7
  - shell sorts, 298–301
- ALL statement, 229–30
- Alphabet, phonetic, 223–24
- ALT key, 16–17
- Amiga BASIC
  - advanced features of, 4
  - compared to other BASIC languages, 3
  - entering, 6
  - functions in, 39–40. *See also* Functions
  - loading, 5–6
- AND statement, 84–86
  - with PUT statement, 192–93
- APPEND statement, 236–37
- Apple Macintosh, 3, 280
- AREAFILL statement, 179–80, 182
- AREA statement, 179–80
- Arguments, 95
- Array variables, 99–101.
  - See also* Integer array; Variables
  - boundaries of, 102–4
  - bubble sorts and, 294–95
  - as buffers, 104–6
  - as buffers for sequential files, 234
  - for creating graphs, 169–70
  - DIM statement and, 101–2, 107
  - multi-dimensional, 106–11
  - passed between programs
    - with CHAIN statement, 229–31
    - with PATTERN statement, 182
  - in subprograms, 111–13
  - with WAVE statement, 210, 217–18
- Arrow keys, 14, 16–17
- Artificial intelligence, 307–11
- ASCII format, 227–28, 279,
  - 290. *See also* CHR\$ function
- Aspect ratios, 179
- ATDT, 290–91
- ATN function, 40
- Autodial, 290

- B (Box), 168
- BACKSPACE key, 14, 15, 26
- Bar graphs, 169–74
- BASIC languages, 3. *See also* Amiga BASIC
- BASIC window. *See* OUTPUT window
- Baud rate, 278–79, 281, 289–90
- Binary numbers, 180–81
- Bits, 281
- Block structure, of
  - IF/THEN/ELSE statements, 307–11
- Bobs
  - crashing, 202–5
  - creating, 205–6
  - displaying, 198–200
  - moving, 200–202
- Box (B), 168
- Branching, 77
  - with computed GOTO and GOSUB statement, 88–92
  - with IF/THEN/ELSE statement, 78–86
  - with ON MENU statement, 142–43
  - with relationals, 82–86
  - with subroutines, 87–88
- British Pound symbol, 264–65
- Bubble sorts, 294–98
- Buffers, 104–6, 232–33
- Bytes, 247–48
  
- Calculations, sequential, 35–37
- Calendar, 137–38
- CALL statement, 93–97, 111–12
- Carriage return, in sorting, 304–7. *See also* RETURN key
- Case (alphabetic), 26, 57–58
- CDBL function, 29
- Centering, 116, 136–37
- CHAIN statement, 229–31
- CHDIR statement, 232
- Checkerboard pattern, 181–83
- Check marks, in menus, 145, 148
- CHR\$ function, 304
  - decoding, 264–66
  - and printer control, 267–71
  - in telecommunications, 282
- CINT function, 29
- CIRCLE statement, 177–79
  - with MOUSE functions, 196
- CLNG function, 29
- CLOSE statement, 233–36
- COLLISION statement, 202–5
- Color printers, 262–63
- COLOR statement, 135–37. *See also* PALETTE statement
- Columns, 154–55
- Commands, 8–9, 11. *See also* Functions; Statements
- FILES, 6–8
- LIST, 9–10, 11
- NEW, 10, 26
- COMMON statement, 230–31
- Concatenation, 30–31, 126
- Copy, 14–15
- COS function, 40
- Counters, in FOR/NEXT loops, 72–73
- CSNG function, 29
- CSRLIN function, 152–55
- CTRL key, 264, 266
- Cursor, 6
  - locating position of, 152–55
  - with WINDOW statements, 157
- Cut, 14–15
  
- Daisy wheel printers, 262
- Data-bits, 281
- Data entry
  - with INKEY\$ statement, 57
  - with INPUT\$ statement, 55–57
  - with INPUT statement, 51–54
  - with READ and DATA statements, 60–62
  - with UCASE\$ statement, 57–58
- Data files. *See* Random access files; Sequential files
- DATA statement, 60–62, 124–25
  - for creating menus, 139
  - with multi-dimensional arrays, 107–8
  - with SAY statement for speech synthesis, 221
  - with SOUND statement for storing music, 215–16
- DATE\$ function, 137–39
- Debugging
  - structured programming and, 5, 49, 62
  - variable naming and, 26
- Decimal point, formatting with PRINT USING statement, 133
- DEFDBL function, 28
- Deferred mode. *See* Program mode
- DEF FN, 44–45
- DEFINT function, 28
- DEF LNG function, 28
- DEFSTR function, 28, 31
- Descriptive labels, 3, 20, 63. *See also* Labelling
- DIM statement, 101–2, 107
  - with subprograms, 111
- Direct mode. *See* OUTPUT window
- Disks
  - backing up, 2–3
  - drawers and files on, 7–8, 232
  - and loading Amiga BASIC, 5
- Division, 12
- Documentation, 22–23
- Dot matrix printers, 262
- Downloading, 279, 282, 283, 290
- Drives. *See* Disks
  
- Echo, 55–57
- Editing programs, 13–17
- Ellipses, 179
- ELSE statement. *See* IF/THEN/ELSE statement
- END IF statement, 308
- END statement, 68
- EOF function, 235
- Epson printers, 267–68
- Error messages
  - ?Redo from start, 52
  - illegal function call, 223
  - Subscript out of range, 102
  - Type mismatch, 30, 61, 242
- ESC key, 265–66
- ESC sequences, 267–70
- EXP function, 41
- Exponentiation, 12
  
- F (Fill), 168
- FIELD statement, 248
- F1 key, 282
- Files, 6–8. *See also* Random access files; Sequential files
- COM1:, 280–82



- disks and, 232  
 LPT1; 272  
 printers and, 270, 272  
 screen and, 272  
 SCRNI; 272  
 for sprites and bobs, 199-200  
 transferring via modem. *See*  
 Telecommunications
- FILES command, 6-8
- Fill (F), 168
- FIX function, 41
- Flagging  
 in menu programs, 148-49  
 in WHILE/WEND loops, 74-75
- Flashing effect, 136
- FOR/NEXT loops, 68-70  
 and array variables, 100-101,  
 103-4  
 counters in, 72-73  
 for creating graphs, 169-70  
 for creating musical scales,  
 213-14  
 to format output with  
 substrings, 119  
 for moving graphics, 186-87,  
 190-93  
 nested, 70-71  
 steps in, 71-72  
 variable names in, 71
- Formal parameters, 95
- Formatting, 129-30  
 with COLOR statement,  
 135-37  
 with functions, 130-32  
 printer, 274-75. *See also*  
 Printer control  
 with PRINT statement, 19-21  
 with PRINT USING statement,  
 132-35  
 program listings, 21-22, 63  
 sequential files, 241-45  
 strings. *See* Strings  
 with WRITE statements, 59
- Function keys, 265-66
- Functions, 9. *See also*  
 Commands; Statements  
 ABS, 40  
 ATN, 40  
 built in, 39-44  
 CDBL, 29  
 CHR\$. *See* CHR\$ function  
 CINT, 29  
 CLNG, 29  
 COS, 40  
 CSNG, 29  
 CSRLIN, 152-55  
 DATE\$, 137-39  
 DEFDBL, 28  
 defined, 44-45  
 DEFINT, 28  
 DEFLNG, 28  
 DEFSTR, 28, 31  
 EOF, 235  
 EXP, 41  
 FIX, 41  
 HEX\$, 44  
 INSTR, 122-23  
 INT, 29, 41, 43, 172  
 LBOUND, 102-4  
 LEFT\$, 117-20, 123, 254  
 LEN, 116-17  
 LOC, 281-82  
 LOG, 41, 299-300  
 LPOS, 271  
 MID\$, 117-20, 122-23  
 MOD, 12, 154  
 MOUSE, 193-97  
 OCT\$, 44  
 OPTION BASE, 102-3  
 POS(0), 152-55  
 RIGHT\$, 117-20  
 RND, 42-44, 166-67  
 SIN, 40, 210  
 SPACE\$, 127, 130-32  
 SPC, 130-32  
 SQR, 10-11, 42  
 STR\$, 32, 125  
 TAB, 116, 130-32  
 TAN, 41  
 TIMES\$, 42-43, 79-80, 120-22  
 UBOUND, 102-4  
 VAL, 31-32, 124-25
- GOTO statement, 78-81  
 computed, 88-92  
 relationals and, 82-86
- Graphics  
 CIRCLE statement, 177-79  
 COLLISION statement, 202-5  
 creating colors with PALETTE  
 statement, 164-68  
 creating shapes, 168-74  
 creating sprites, 197-98  
 displaying sprites and  
 bobs, 198-200  
 drawing ellipses with aspect  
 ratios, 179  
 filling with PAINT statement,  
 176-77  
 filling with PATTERN  
 statement, 179-83  
 lighting pixels with PSET  
 statement, 163-64  
 moving sprites and bobs,  
 200-202  
 moving with MOUSE  
 functions, 193-97  
 moving with PRESET  
 statement, 185-87  
 moving with PUT and GET  
 statements, 187-93  
 relative plots, 174-76  
 text in, 172-74
- Graphics array. *See* Integer array  
 Graphs, 169-74
- "Hayes" protocol, 290
- HEX\$ function, 44
- Hexadecimal numbers,  
 44, 180-83  
 determining decimal  
 equivalent, 267
- GET statement  
 for graphics, 187-93  
 for random access files, 251
- Ghost toggle, 149
- GOSUB statement, 87-88. *See*  
*also* Subroutines  
 with COLLISION statement,  
 202-4  
 computed, 88-92  
 in menu programs, 143  
 in window programs, 159
- IBM PC, 3
- IF/THEN/ELSE statement, 78-81  
 block structure for artificial  
 intelligence, 307-11  
 relationals and, 82-86
- Illegal function call error  
 message, 223
- Immediate mode. *See* OUTPUT  
 window

- INKEY\$ statement, 57  
 in telecommunications programs, 282  
 in window programs, 157–58
- Ink jet printers, 262
- INPUT\$ statement, 55–57  
 with telecommunications, 281–82
- INPUT statement, 51–54.  
 See also LINE INPUT statement  
 with array variables, 100–101, 169  
 with CHAIN statement, 229–30  
 checking with LEN function, 117  
 entering more than one value with, 109  
 with sequential files, 234–35  
 in window programs, 157–58
- INSTR function, 122–23
- Integer array  
 with PUT/GET statements, 188, 189  
 with SAY statement, 220–22, 225
- Integer division, 12–13
- Interactive data entry. See Data entry
- INT function, 29, 41, 43  
 for creating graphs, 172
- Intonation, in speech synthesis, 225
- Inverse output, 135–37
- Japanese Yen symbol, 264–65
- Jumping, 16–17. See also Scrolling from loops, 74
- Labelling. See also Descriptive labels  
 in graphics, 172–74  
 with PRINT statements, 58–59
- Laser printers, 262
- LBOUND function, 102–4
- LEFT\$ function, 117–20, 123  
 for searching random access files, 254
- LEN function, 116–17
- LINE INPUT statement, 242–44.  
 See also INPUT statement
- Line numbers, 3, 63. See also Descriptive labels
- LINE statement, 168–74  
 with MOUSE functions, 196  
 relative plotting of, 175
- LIST command, 9–10, 11. See also LLIST statement
- LIST window, 6  
 editing programs in, 13–14  
 running programs from, 9  
 scrolling in, 16–17, 49
- LLIST statement, 263. See also LIST command
- Loading, Amiga BASIC, 5–6
- Local variables, 94–97, 112
- LOCATE statement, 130–32  
 for centering strings on screen, 136–37  
 with cursor position functions, 153  
 for text in graphics, 172–74
- LOC function, 281–82
- LOG function, 41  
 for shell sorts, 299–300
- Logical operations, 83, 90–92
- Loops, 67  
 counters in, 72–73  
 FOR/NEXT. See FOR/NEXT loops  
 jumping out of, 74  
 nested, 70–71  
 steps in, 71–72  
 WHILE/WEND, 73–75, 235
- LPOS function, 271. See also POS function
- LPRINT statement, 263–64. See also PRINT statement
- LPRINT USING statement, 273–74. See also PRINT USING statement
- LPT1: filename, 272
- LSET statement, 249–50, 254
- Macintosh, Apple, 3, 280
- Masks, in OBJECT.HIT statement, 205
- Mathematical operations, 12–13  
 built-in functions, 39–42  
 defined functions, 44–45  
 generating random numbers, 42–44
- hexadecimal and octal numbers, 44  
 parentheses in, 38–39  
 precedence of, 37–39  
 sequential calculations, 35–37
- Mathematical relations. See Relations
- Memory  
 clearing programs from, 10, 26  
 placing arrays in, 182  
 saving with DIM statement, 102  
 saving with OPTION BASE function, 103
- Menus, 141–42. See also MENU statements  
 creating, 138–39, 142–45  
 opening, 8  
 refreshing, 149  
 toggling, 145–49
- MENU statements, 142–45.  
 See also Menus
- MENU ON/OFF/STOP statement, 142
- MENU RESET statement, 142
- ON MENU statement, 142–43, 240–41  
 in telecommunications programs, 289–91
- MERGE statement, 227–28
- Microsoft BASIC. See Amiga BASIC; BASIC
- MID\$ function, 117–20, 122–23
- Modems, 277–78  
 autodial, 290  
 baud rate, 278–79, 281, 289–90
- MOD function, 12, 154
- Modular programming, 5, 62–64. See also Structured programming
- Mouse. See also MOUSE functions  
 for changing window sizes, 17  
 for controlling moving graphics, 193–97  
 editing programs with, 13–15, 26  
 with menus, 141–42  
 for running programs from the LIST window, 9
- MOUSE functions, 193–97. See also Mouse

- Multi-dimensional arrays, 106–11
  - and subprograms, 112–13
- Multiplication, 12, 38. *See also* Mathematical operations
- Music. *See also* SOUND statement
  - notes, 212–15
  - songs, 215–16
  
- Narrator, 222–25
- Nested IF/THEN/ELSE statements, 308
- Nested loops, 70–71. *See also* Loops
- NEW command, 10, 26
- NEXT statement. *See* FOR/NEXT loops
- NOT statement, 84–86
- Numbers, 12–13. *See also*
  - Mathematical operations;
  - Numeric variables
  - binary, 180–81
  - bytes for storing, 247–48
  - hexadecimal, 44, 180–83, 267
  - octal, 44
  - random, 42–44
- Numeric variables. *See also* Numbers; Variables
  - assigning, 24–25
  - changing, 29
  - converting to string, 32, 125, 251–52
  - defining, 28
  - naming, 25–26
  - precision of, 27–29
  - in WHILE/WEND loops, 73–74
  
- OBJECT statements
  - OBJECT.AX/AV, 201–2
  - OBJECT.HIT, 203–5
  - OBJECT.ON/OFF, 199–200
  - OBJECT.SHAPE, 198–200
  - OBJECT.START/STOP, 201
  - OBJECT.VX/VY, 200–201
  - OBJECT.X/Y, 199–200
- ObjEdit program, 197–98, 205–6
- Oblong shapes, 179
- OCT\$ function, 44
- Octal numbers, 44
  
- Online networks, 279
- ON MENU statement, 142–43, 240–41. *See also* MENU statements
- ON statement, 88–92
  - in menu programs, 143
- OPEN statement
  - with "COM1:" filename, 280–81
  - with "LPT1:" filename, 272–73
  - with random access files, 248–49
  - with "SCRN:" filename, 272
  - with sequential files, 233–36
- Operations. *See* Mathematical operations
- OPTION BASE function, 102–3
- OR statement, 84–86
  - with PUT statement, 192–93
- OUTPUT window
  - commands in, 8–9
  - displaying files in, 6–8
  - LISTing programs from, 9
  - PRINT statement in, 11
- Ovals, 179
- Overhead bits, 281
  
- Padding strings, 126–27, 249, 254
- PAINT statement, 176–77
- PALETTE statement, 164–68
- Parameters
  - formal, 95
  - of integer array with
    - SAY statement, 220–22, 225
  - with MENU statement, 142–45, 149
  - with OBJECT.HIT statement, 205
  - for opening the COM1:
    - file, 280–81
  - with PALETTE statement, 165
  - with SCREEN statement, 159–60, 166–67, 188
  - with SOUND statement, 210–11
- Parity, 281
- Parsing strings, 126–27, 249
- Paste, 14–15
- PATTERN statement, 179–83
- Pauses, in programs
  - using INPUT statement, 54, 55–56
  
- using TIMES\$ function, 79–80, 121–22
- Phonetic transcription, 222–25
- Pi, 178
- Pixels, 156, 163–64, 181, 189
- POS(0) function, 152–55. *See also* LPOS function
- Pound symbol, British, 264–65
- Precedence, of mathematical operations, 37–39
- PRESET statement, 185–87
- Printer control
  - CHR\$ and, 267–71
  - with LPOS function, 271
  - with LPRINT USING statement, 273–74
  - with OPEN "LPT1:" statement, 272–73
  - text output, 263–64
  - typefaces, 268–70
  - with TAB, SPACE\$ and SPC functions, 274–75
- Printers, types of, 261–63
- PRINT statement, 9, 11. *See also*
  - LPRINT statement
  - with INPUT statement, 52
  - labelling with, 58–59
  - with numbers, 12
  - output formatting and, 19–21, 31
  - with sequential files, 233–34, 244–45
  - with text, 12
- PRINT USING statement, 132–35. *See also* LPRINT USING statement
  - with sequential files, 241–45
- Programming
  - avoiding shortcuts in, 37
  - creative, 47–48
  - defining, 2
  - experimenting, 2–3
  - structured, 4–5, 47–51, 62–64, 104
- Program mode. *See* LIST window
- Programs
  - address book, 237–41
  - for artificial intelligence, 309–11
  - editing, 13–17
  - formatting with TAB key, 21–22, 63

- Programs (*continued*)  
 listing 9–10, 11  
 ObjEdit, 197–98, 205–6  
 pauses in, 54, 55–56, 79–80, 121–22  
 PRINT USING statements  
   at beginning of, 135  
   progressive copies of, 54  
   remarks in, 22–23, 64  
   running, 9  
   saving, 54  
   from subroutines using  
     MERGE statements, 227–28  
   telecommunications, 283–91  
   timing with TIME\$ function, 121–22  
   using parts of with CHAIN statement, 229–31
- Project window, 10
- Prompts, for INPUT statements, 52–53
- PSET statement, 164  
 with MOUSE functions, 195  
 with PALETTE statement, 167  
 with PRESET statement, 185–87  
 STEP statement with, 174–75
- PUT statement  
 for graphics, 187–93  
 for graphics with MOUSE functions, 196–97  
 with random access files, 250–51
- Radians, 178
- Random access files  
 adding new records to, 252–54  
 buffers and variables in, 249–54  
 changing a record, 256–59  
 creating, 247–49  
 number of records in, 251  
 searching, 254–56
- RANDOMIZE statement, 42–43  
 with PALETTE statement, 166–67
- Random numbers, 42–44
- READ statement, 60–62, 124–25
- Relationals  
 with computed GOSUB statement, 90–92  
 in IF/THEN/ELSE statements, 82–86  
 with strings, 92–93, 294–95
- Relative plots, 174–76
- REM statement, 22–23, 64
- Reserved words. *See* Commands; Functions; Statements
- Resolution, screen, 160
- RESTORE statement, 61–62  
 with multi-dimensional arrays, 107–8
- RETURN key, 6–7. *See also* Carriage return
- RETURN statement, 87. *See also* GOSUB statement
- RIGHT\$ function, 117–20
- RND function, 42–44  
 with PALETTE statement, 166–67
- RS232 connector, 279–80
- RS232 port, 277–78
- Save As option, 54
- SAVE statement, 227–28
- SAY statement  
 integer array with, 220–22  
 with TRANSLATE\$ statement, 219–22
- Screen. *See also* SCREEN statement; Windows  
 creating, 159–60  
 creating columns on, 154–55  
 as a file with OPEN "LPT1:" statement, 272  
 formatting. *See* Formatting  
 locating cursor position on, 152–55  
 scrolling, 16–17, 49, 151–52  
 width of, 23–24, 291
- SCREEN statement, 159–60. *See also* Screen  
 and moving graphics, 188  
 with PALETTE statement, 166–67
- SCRN: filename, 272
- Scrolling, 16–17, 49  
 control of, 151–52
- Sequential calculations, 35–37
- Sequential files  
 appending, 236–37  
 buffers and, 232–33  
 creating address book with, 237–41  
 formatting with PRINT USING statement, 241–45  
 input from disk, 234–36  
 output to disk, 233–34  
 sorting, 304–7
- Shared variables, 95–97
- Shell sorts, 298–301
- SHIFT key, 15, 16–17
- SIN function, 40  
 with WAVE statement, 210
- Sorts  
 bubble, 294–98  
 rearrangements with  
   bundled, 301–7  
   shell, 298–301
- SOUND statement, 209–10. *See also* Speech synthesis  
 for creating musical notes, 212–15  
 for creating songs, 215–16  
 SOUND WAIT/RESUME, 211  
 with WAVE statement, 210–12, 217–19
- SPACE\$ function, 127, 130–32, 275
- SPC function, 130–32, 275
- Speech synthesis  
 for artificial intelligence, 308–11  
 stress in, 224–25  
 using SAY and TRANSLATE\$ statements, 219–22  
 using SAY statement and phonetic transcription, 222–25
- Sprites  
 crashing, 202–5  
 creating, 197–98  
 displaying, 198–200  
 moving, 200–202  
 for telecommunications program, 283
- SQR function, 10–11, 42
- Statements, 11. *See also* Commands; Functions  
 ALL, 229–30  
 AND, 84–86, 192–93  
 APPEND, 236–37  
 AREA, 179–80  
 AREAFILL, 179–80, 182

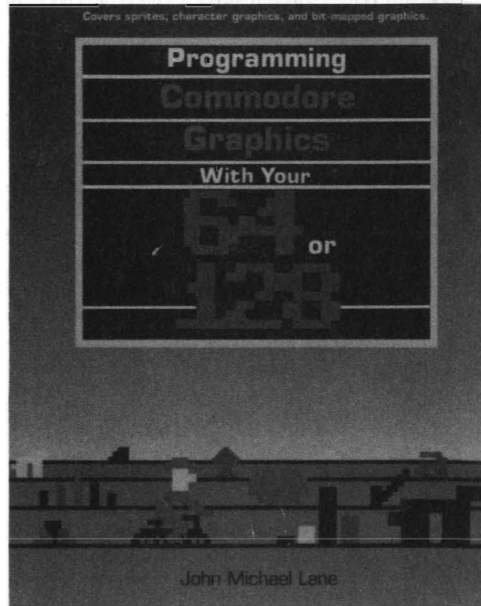
- CALL, 93–97, 111–12  
 CHAIN, 229–31  
 CHDIR, 232  
 CIRCLE, 177–79, 196  
 CLOSE, 233–36  
 COLLISION, 202–5  
 COLOR, 135–37  
 COMMON, 230–31  
 DATA. *See* DATA statement  
 DIM, 101–2, 107, 111  
 ELSE. *See* IF/THEN/ELSE statement  
 END, 68  
 END IF, 308  
 FIELD, 248  
 FOR/NEXT. *See* FOR/NEXT loops  
 GET, 187–93, 251  
 GOSUB. *See* GOSUB statement  
 GOTO. *See* GOTO statement  
 IF/THEN/ELSE. *See*  
   IF/THEN/ELSE statement  
 INKEY\$, 57, 157–58, 282  
 INPUT\$, 55–57, 281–82  
 INPUT. *See* INPUT statement  
 LINE INPUT, 242–44  
 LINE, 168–74, 175, 196  
 LLIST, 263  
 LOCATE. *See* LOCATE statement  
 LPRINT, 263–64  
 LPRINT USING, 273–74  
 LSET, 249–50, 254  
 MENU. *See* MENU statements  
 MERGE, 227–28  
 NEXT. *See* FOR/NEXT loops  
 NOT, 84–86  
 OBJECT. *See* OBJECT statements  
 ON, 88–92, 143  
 OPEN. *See* OPEN statement  
 OR, 84–86, 192–93  
 PAINT, 176–77  
 PALETTE, 164–68  
 PATTERN, 179–83  
 PRESET, 185–87  
 PRINT. *See* PRINT statement  
 PRINT USING, 132–35, 241–45  
 PSET. *See* PSET statement  
 PUT. *See* PUT statement  
 RANDOMIZE, 42–43, 166–67  
 READ, 60–62, 124–25  
 REM, 22–23, 64  
 RESTORE, 61–62, 107–8  
 RETURN, 87  
 SAVE, 227–28  
 SAY, 219–22  
 SCREEN, 159–60, 166–67, 188  
 SOUND. *See* SOUND statement  
 STEP. *See* STEP statement  
 SWAP, 295–96  
 THEN. *See* IF/THEN/ELSE statement  
 TRANSLATE\$, 219–22  
 UCASE\$, 57–58  
 WAVE, 210–12  
 WHILE/WEND, 73–75, 235  
 WIDTH, 23–24, 131  
 WINDOW, 155–60  
 WRITE, 59–60, 245
- Static variables, 94–97, 112
- STEP statement  
   with AREA statement, 179–80  
   in FOR/NEXT loops, 71–72  
   for moving graphics, 192  
   with PSET statement, 174–75
- Stop-bits, 281
- STR\$ function, 32  
   and substrings, 125
- Stress, with speech synthesis,  
 224–25
- Strings, 115. *See also* String  
 variables; Substrings  
   bytes for storing, 247  
   centering, 116, 136–37  
   concatenating, 126  
   created from substrings, 119  
   determining length with LEN  
   function, 116–17  
   padding, 126–27, 249, 254  
   relationals and, 92–93, 294–95  
   searching for substrings, 122–23  
   truncating, 126–27, 249
- String toggle, 148
- String variables. *See also* Strings;  
 Variables  
   assigning, 31  
   as buffers for sequential  
   files, 236  
   concatenating, 30–31, 126  
   converting to numeric, 31–32,  
   124–25, 252  
   naming, 29  
   with PRINT USING statement,  
   135  
   in random access files, 249–51
- Structured programming,  
 4–5, 47–51, 62–64  
   array variables as buffers in,  
   104
- Subdirectories, 7, 232
- Subprograms, 5, 93–97.  
   *See also* Subroutines  
   array variables in, 111–13
- Subroutines, 87–88. *See also*  
 Subprograms  
   with COLLISION statement,  
   202–4  
   with computed GOSUB  
   statement, 88–92  
   to create programs with  
   MERGE statement, 227–28  
   with ON MENU statement,  
   142, 240–41  
   in window programs, 159
- Subscript out of range  
 error message, 102
- Substrings. *See also* Strings  
   with DATE\$ function, 137–38  
   functions for finding, 117–20  
   numbers and, 124–25  
   for searching random access  
   files, 254  
   searching strings for, 122–23  
   with TIME\$ function, 120–22
- Subtraction, 12. *See also*  
 Mathematical operations
- SWAP statement, 295–96
- TAB function, 116, 130–32,  
 274–75
- TAB key, formatting program  
 listings with, 21–22, 63
- TAB stops  
   defined by WIDTH statement,  
   23–24  
   writing sequential files  
   with, 245
- TAN function, 41
- Telecommunications  
 COM1: file and, 280–82  
 creating program for, 283–89  
 modems, 277–80  
 using program for, 289–91

- Text, 12. *See also* String variables in graphics, 172-74
- THEN statement. *See* IF/THEN/ELSE statement
- Three-dimensional arrays. *See* Multi-dimensional arrays
- Tick mark ('), 22-23, 64
- TIME\$ function, 42-43, 79-80, 120-22
- Toggling, in menus, 145-49
- Top down programming, 4-5, 48-51. *See also* Structured programming
- TRANSLATE\$ statement, 219-22
- Truncating strings, 126-27, 249
- Two-dimensional arrays. *See* Multi-dimensional arrays
- Typefaces, 268-70
- Type mismatch error message, 30, 61, 242
- UBOUND function, 102-4
- UCASE\$ statement, 57-58
- Uploading, 279, 282, 290
- VAL function, 31-32  
with substrings, 124-25
- Variables. *See also* Array variables; Numeric variables; String variables  
as buffers for sequential files, 234  
with cursor position functions, 152-53  
descriptive, 36  
with formatting statements, 131, 241-45  
in FOR/NEXT loops, 69, 71  
in IF/THEN/ELSE statement, 80-81  
with INPUT statements, 52  
in menu programs, 148-49  
parallel, 94-96  
passed between programs with CHAIN statement, 229-31  
with READ and DATA statements, 60-62, 124-25  
relationals and, 83  
shared, 95-97  
static, 94-97, 112  
string-number conversions, 31-32, 124-25, 251-52  
in subprograms, 93-97
- Velocity, of moving sprites and bobs, 200-201
- Voice. *See* SOUND statement; Speech synthesis
- WAVE statement, 210-12
- WEND statement. *See* WHILE/WEND loops
- WHILE/WEND loops, 73-75  
with EOF function, 235
- WIDTH statement, 23-24  
setting with INPUT statement, 131
- Windows, 17, 23-24, 155-60.  
*See also* LIST window; OUTPUT window; Screen
- WINDOW statement, 155-60
- WRITE statement, 59-60  
with sequential files, 245
- XOR, 189, 192
- Yen symbol, Japanese, 264-65
- Zeroes, formatting with PRINT USING statement, 132-33



**ANOTHER COMMODORE BOOK FROM SCOTT, FORESMAN  
AND COMPANY**

**PROGRAMMING COMMODORE  
GRAPHICS WITH YOUR  
64 OR 128**



By John Michael Lane, 224 pages, softbound, **\$14.95**.  
Code: 18084

Improve your graphics programming on the Commodore 64 and 128 computers with this clear and well-written introduction to the world of Commodore graphics. This highly readable tutorial starts with the simplest applications and works up to the more complex.

You'll learn how to

- create sprite animation
- design your own character images
- use machine language subroutines
- work with multicolor character modes
- create bar, line, and pie graphs and more

PROGRAMMING COMMODORE GRAPHICS WITH YOUR 64 OR 128 also provides 3 applications programs—for business graphics, a math bingo game, and a racing game with sprites—that show a variety of graphics techniques in action.



**HERE'S HOW TO ORDER:**

Contact your local bookstore or computer store, or send the handy order form below to:

**Scott, Foresman and Company**  
**Professional Publishing Group**  
1900 East Lake Avenue  
Glenview, IL 60025

**In Canada, contact**  
Macmillan of Canada  
164 Commander Blvd.  
Agincourt, Ontario  
M1S 3C7

YES, please send me \_\_\_\_\_ copies of PROGRAMMING  
COMMODORE GRAPHICS WITH YOUR 64 OR 128, \$14.95, 18084

Please check method of payment:

Check/Money Order       MasterCard       VISA

Amount enclosed \$ \_\_\_\_\_

Credit Card No. \_\_\_\_\_ Exp. Date \_\_\_\_\_

Signature \_\_\_\_\_

Name (please print) \_\_\_\_\_

Address \_\_\_\_\_

City \_\_\_\_\_ State \_\_\_\_\_ Zip \_\_\_\_\_

Add applicable sales tax, plus 6% of total for shipping.

Full payment must accompany your order.

Mail order form to: Scott, Foresman and Company  
Professional Publishing Group  
1900 East Lake Avenue  
Glenview, IL 60025

A18523

**HERE'S HOW TO RECEIVE YOUR FREE CATALOG OF THE LATEST  
COMPUTER BOOKS FROM SCOTT, FORESMAN AND COMPANY**

Simply mail in the coupon below to receive your free copy of our latest catalog featuring computer and management books, and find out how they can benefit you.

- YES, please send me my **free** catalog of your latest computer and management books! I am especially interested to learn more about your books on
- Programming
  - Business Applications
  - Networking and Telecommunications
  - Other \_\_\_\_\_

Name (please print) \_\_\_\_\_

Address \_\_\_\_\_

City \_\_\_\_\_ State \_\_\_\_\_ Zip \_\_\_\_\_

**Mail coupon to:** Scott, Foresman and Company  
Professional Publishing Group  
1900 East Lake Avenue  
Glenview, Illinois 60025



# Program Your Amiga Quickly and Easily

## Reviewers say:

"... exceptionally clear, fun-to-do examples really speed up the Amiga Microsoft BASIC learning process."

"If your time is of value, this is the book for you."

Master Amiga Microsoft BASIC with this easy-to-understand book. You'll find a complete, step-by-step guide to Microsoft BASIC for the Amiga, and numerous, interesting examples of advanced features, designed for both beginning and intermediate users.

Noted computer author Bill Sanders helps you discover how to

- create and customize your own software for your individual needs
- teach your children with learning games you can write yourself
- write database and multi-tasking programs
- integrate different programs to work together
- call up your favorite electronic information service using the book's terminal program

**The Amiga Microsoft BASIC Programmer's Guide** gives special attention to the color graphics and voice synthesizer — two of Amiga's most unique features. It also provides extensive coverage of pull-down menus and special mouse control, using practical program examples, including a special artificial intelligence program.

With **The Amiga Microsoft BASIC Programmer's Guide**, learning how to unleash your Amiga's potential has never been easier.



**Dr. William B. Sanders** is a professor at San Diego State University and is the author of numerous computer books, including THE ELEMENTARY ATARI ST. He is also the author of several commercial software programs.

**Scott, Foresman and Company**