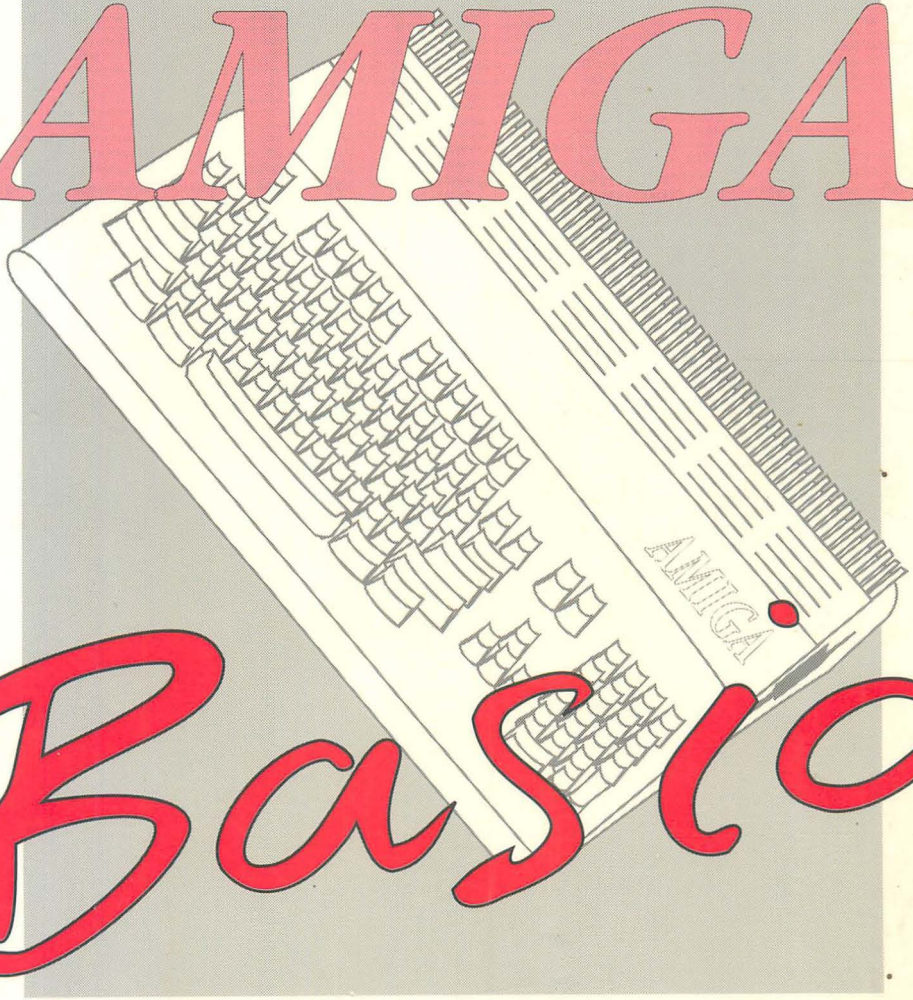
A Dabhand Guide

PAUL FELLOWS

AMIGA Basic



DABS PRESS

AmigaBASIC

A Dabhand Guide

by
Paul Fellows

DABS
PRESS

AmigaBASIC: A Dabhand Guide

Written by Paul Fellows

ISBN 1-870336-87-9

First Edition, first printing February, 1992
Editor: Syd Day

Typesetting: David Atherton, Bruce Smith
Production: Andrew Wygladala
Cover Artwork: Clare Atherton

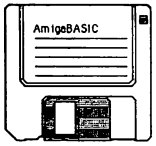
This book was produced using an Apple Macintosh desktop publishing system.

Typeset in 10/12pt Palatino.

Published by Dabs Press, PO Box 48, Prestwich, Manchester M25 7HF. Telephone 061-773 8632. Fax 061-773 8290.

Printed and bound in the UK by BPCC Wheaton, Exeter EX2 8RP.

Contents



1 : Introduction

When you've got your new Amiga computer sitting proudly on the desk, raring to go, you will no doubt want to put it through its paces to see what it can do. Using the Workbench, you can do things such as perform calculations, write letters and even make the computer speak to you. But the capabilities of the Workbench are limited, so what do you do when you feel that you've exhausted them and want to move on to something else?

There are two possible routes. The first is to go out and buy some different software packages. There are hundreds available which will turn your machine into anything from a desktop publishing system to a games console for zapping alien spacecraft.

The second option is to write your own software. This is not as difficult as you might think. The advent of home computers has meant that programming is now a pastime which anyone can participate in and enjoy.

This book is aimed at all those people who want to take this second route and learn how to control their computer. For those of you who have never programmed a computer before, it starts at the beginning. It explains what a program is and takes you, step by step, through the stages involved in writing one. For readers who have programmed other computers, the later chapters examine the features of AmigaBASIC in greater detail. They cover topics such as animation, file handling, synchronised sound production and the use of subprograms.

Extensive use is made of example programs. Short examples are used to illustrate each new point as it is made. In addition, larger programs are built up to show how different features can be combined to create real applications.

The Basics

This chapter covers the groundwork for the rest of the book. It introduces the AmigaBASIC system and demonstrates all the fundamental concepts which are going to be needed at later stages. These include how to use the editor to write and alter programs and how to give commands to load, save and execute these programs. Common problems which you may encounter are examined and some standard terminology is explained.

What Is a Program ?

A person trying to direct a driver to a particular destination has two ways he can go about it. He can travel with the driver and give the instructions, one at a time, at the appropriate stage in the journey. Alternatively, he can give the driver all the instructions before the journey starts, so the driver can execute them in sequence as he goes.

This second approach is similar to programming a computer. A program is just a series of instructions which you enter into the computer and then tell the computer to carry out. It will step through each of these instructions in turn, execute it, and move on automatically to the next one.

Both the driver and the computer can only understand instructions which are in a language that they know. For the driver this may mean English, French, Italian etc. For the Amiga it means BASIC. BASIC is an acronym for 'Beginners' All-purpose Symbolic Instruction Code'. As the name implies it is ideal as an introductory language for beginners. In addition, it is so versatile and powerful that many commercially available software packages are written in it. These properties have led it to become, by far, the most popular language for home computers.

Returning to our driver analogy, there is another point to note at this time. If you give either the driver or the computer the wrong instructions then they won't do as you expected.

For example, if you tell the driver to turn right rather than left, he will almost certainly end up in the wrong place! Similarly, if you tell the

computer to multiply two numbers together, when you really wanted to add them, then the result it produces will be wrong.

Alternatively, you might give them an instruction which doesn't make sense or is ambiguous. In this case, the driver may well curse, return home, ask you to correct his list of instructions, and then try again. Likewise, the computer will print a message to tell you all is not well and stop. You will then have to correct the mistake and try executing the program again.

However, the computer does have an advantage over the driver – it has an endless supply of patience. Don't be afraid of trying things out. The computer won't get annoyed if you keep making mistakes!

Entering AmigaBASIC

To write a program in BASIC, you first have to start up the AmigaBASIC language which lives on the Amiga Extras Disk. To do this:

- Turn on your Amiga.
- When the prompt appears, place the Workbench Disk into the disc drive.
- When the disc drive light goes out, replace the Workbench Disk by your copy of the Amiga Extras Disk.
- Double-click with the left-hand mouse button on the Extras Disk icon.
- Double-click with the left-hand mouse button on the AmigaBASIC icon.

The AmigaBASIC screen should appear. This consists of two windows the main AmigaBASIC window which contains a few lines of text and a smaller empty window whose title is 'LIST'. These will be referred to as the 'Output window' and the 'List window' respectively.

The two windows have different functions. The List window is used to hold BASIC programs. You will see later in this chapter that all programs are typed or loaded into this window and any editing you do takes place in it.

The Output window lets you type commands into the computer: to tell it to execute a program you have written, for example. In addition,

your program can print its results or display its graphics in the Output window. Therefore, this window can be used for all the communication which occurs between you and the computer – your commands to it and its replies to you.

Writing and Running a Program

When you enter AmigaBASIC, you will see that the title bar of the List window is more distinctive than that of the Output window. This is because the List window is selected. Therefore, any text you type at this stage will be entered into the List window rather than the Output window. Try typing the following:

```
CIRCLE (320,100),50
```

This is a BASIC program! It consists of just one 'statement' or instruction. As you may well have guessed, this statement instructs the computer to draw a circle.

'CIRCLE' is an example of a BASIC 'keyword'. This is a word which BASIC recognises and treats in a special way. The rest of the statement provides the other information which BASIC needs in order to be able to draw the circle. The '(320,100)' gives the position of the centre of the circle and the '50' determines its radius.

To run this program select the Output window by clicking in it. Its title bar will change to being the more distinctive of the two and the prompt:

```
Ok
```

will be displayed. Then type:

```
RUN
```

The List window will disappear, leaving the Output window occupying the full width of your monitor. Then the circle you instructed the computer to draw will appear in the centre of this window.

Making Simple Edits

Having just run your first program, the next thing to do is to return to the List window and modify it. The List window is currently hidden behind the Output window, because the Output window was moved to the front when the circle was drawn. To bring the List window to the front type:

```
LIST
```

and the List window will appear again, covering up most of the circle you produced. Select this window by clicking in it. You will notice, if you look closely, that there is a thin orange line or 'cursor' in this window. This marks the position at which any text that you type will appear. You can move this cursor around by pressing the arrow keys, which are clustered to the right of the main keyboard or by positioning the mouse pointer at the spot you want to move to and clicking the left-hand button. Try this now. Don't worry if the computer beeps at you and flashes the screen. It does this whenever you try to move the cursor anywhere that isn't occupied by your program.

What we are aiming to do is to edit the program so that it reads:

```
CIRCLE (420,100),150
```

The first stage is add a '1' in front of the '50'. To do this, move the cursor so that it lies between the ',' and the '5' and press the number '1'. This number will appear to the right of the cursor and the rest of the text will be moved along to make room for it.

Now we want to change the '320' into '420'. Move the cursor between the '(' and the '3' and press the 'Del' key. This will delete the character to the right of the cursor, ie the '3'. Now type the number '4' and the program should look like the one above.

Run the new program, as before, by selecting the Output window and typing:

```
RUN
```

The contents of the window will be cleared and a new, larger circle will be drawn towards the right-hand side of the screen.

Menus and Amiga Key Commands

So far we have met two commands, one to execute a program and another to bring the List window to the front of the screen. We issued these commands by typing RUN and LIST respectively in the Output window. However, there are alternative methods we could have used.

One is to use the BASIC menu bar. Hold down the menu button (this is the right-hand mouse button) and the menu bar will appear at the top of the screen. This has four entries: 'Project', 'Edit', 'Run' and 'Windows'. Pointing at one of these entries displays the contents of that particular menu. To select an item from a menu, point at the item you want and release the menu button.

The LIST command is the same as selecting Show List from the Windows menu, and the RUN command is equivalent to selecting Start from the Run menu. Note that whichever window is selected at the time you give the command to run a program will still be selected after the program has ended, and hence will be brought to the front. Try this now. Select the List window by clicking in it and select Start from the Run menu. When the circle has been drawn, the List window will be brought to the front of the screen and will partially obscure the circle.

If you look through the items on the menus, you see that some of them are followed by the Amiga key symbol and a letter. For example, Start is followed by the Amiga symbol and an 'R'. This describes the third method of running a program – pressing the letter 'R' whilst holding down the right Amiga key. (This key combination will be referred to as 'Amiga-R' in the remainder of this book.) Similarly, you can bring the List window to the front by pressing 'Amiga-L'.

Some commands, which we will come across later, are only available via one of these methods, but with most you have a choice. There is no 'correct' method which you should use. It's up to you to select the one which you prefer.

Creating Another Program

We are now going to start afresh on a new, larger program. To do this either select the Output window and type:

```
NEW
```

or select the New entry from the Project Menu. In either case a requester will be displayed which contains the following message:

```
Current program is not saved
Do you want to save it before proceeding?
```

This is because creating a new program will destroy the one you currently have in the List window. Unless you save your current program onto a disc first, it will be lost forever. In our case this isn't a problem. Our current program is so short that, if we wanted it back, we could type it in again very easily. However, later in this book, we are going to be developing some fairly large example programs which will take a significant length of time to type in. At that stage, the message will act as a very useful reminder that you will lose the program if you go ahead and start a new one without saving the current one first.

You are given three options:

YES	which asks you to give a filename, saves the current program to disc with this name and then clears out the list window ready to start on a new program
NO	which throws away the current program and prepares to start on a new one
CANCEL	which forgets that you ever asked to create a new program and leaves the current one alone

In this case you should select the NO option. The List window should now be empty again. Select it and enter the following:

```
CIRCLE(320,100),100
CIRCLE(160,50),50
CIRCLE(480,50),50
CIRCLE(160,150),50
CIRCLE(480,150),50
```

Instead of typing each line in turn, you may find it easier to type in the first line only and then copy it four times and make the necessary alterations to the copied lines. Try this as follows:

- Type in the first line and press the RETURN key.
- Move the cursor to the start of the line and press the left-hand mouse button.
- With the mouse button still pressed, move the cursor down a line – the top line will change to being black text on an orange background to illustrate that it has been selected.
- Release the mouse button.
- Select Copy from the Edit menu or press 'Amiga-C' – this copies the text which you have selected into the Clipboard.
- Move the cursor to the start of the next line to mark the position that you want to copy the text to.
- Select Paste from the Edit menu or press 'Amiga-P' – this inserts a copy of the contents of the Clipboard into your program at the current position.
- Repeat the last step three more times to give you the five circle statements.
- Edit the last four lines of the program by positioning the cursor at the appropriate places in the text and either deleting or inserting characters.

Then try running it.

Errors

Errors in programs are commonly referred to as 'bugs'. They come in all sorts of shapes and guises. In this section we are going to look at errors caused by mistyping a program. Alter the current program so that the second 'CIRCLE' is misspelt as 'CRICLE':

```
CIRCLE(320,100),100
CRICLE(160,50),50
CIRCLE(480,50),50
CIRCLE(160,150),50
CIRCLE(480,150),50
```

Now try running this program. Everything will be normal until the computer reaches the 'CRICLE'. Then, since this word is not a valid keyword, it fails to recognise it and reacts by beeping and flashing the screen to tell you all is not well. More helpfully, it displays an error requester describing what it thinks is wrong and brings the List window back to the front with the offending statement enclosed in an orange rectangle.

In this example, the error message it should have given is:

```
Undefined subprogram
```

Don't worry about what this really means. At present, it can be interpreted as 'misspelt keyword'.

To continue from this situation, click on the OK gadget in the error requester. Then select the List window so you can make the appropriate correction to the program. Correct the program and then add another mistake to the program by changing the last comma, on that line, into a full stop:

```
CIRCLE(320,100),100
CIRCLE(160,50).50
CIRCLE(480,50),50
CIRCLE(160,150),50
CIRCLE(480,150),50
```

Now try running it. The result should be the same except that the error message this time will be:

```
Syntax error
```

This means that BASIC has recognised the keyword, but the rest of the statement isn't in the correct format.

The above examples cover the two most common situations you are likely to encounter when typing in and running the programs later in this book. Don't worry if this happens, just correct the mistake and try again.

BASIC Lines

When we talk about a BASIC line, we are referring to a program line. This can contain either a single BASIC statement, as in the above examples, or several statements separated by colons:

```
CIRCLE(320,100),100
CIRCLE(160,50),50   : CIRCLE(480,50),50
CIRCLE(160,150),50 : CIRCLE(480,150),50
```

To convert your program to look like this, move the cursor to the start of the third line before the 'C' of 'CIRCLE'. Then press the 'backspace' key: this is represented on the main part of the keyboard as an arrow.

Normally, this deletes the character to the left of the cursor. However, when the cursor is at the start of the line, there isn't a character to be deleted so the preceding carriage return is deleted instead. This results in the contents of the two lines being merged together. Now type in the colon to separate the two statements: and then repeat this process for the last two circle commands.

If you want to split the BASIC line so that the two statements are on separate lines again, all you need to do is to place the cursor at the point you wish to split the line and press the RETURN key. Then delete the colon since it is no longer necessary.

A program line can be up to 255 characters long. It can contain any number of statements, provided that the maximum length is not exceeded. Note that a statement must lie entirely on one line – it cannot be split over two adjacent ones.

One program line is ended, and the next started, by pressing the RETURN key. This inserts a carriage return into the program which will be treated as part of the line but is invisible.

Normally, spaces between items in a statement are ignored. For example:

```
CIRCLE ( 320 , 100 ) , 100
CIRCLE ( 160 ,  50 ) ,  50 : CIRCLE ( 480 ,  50 ) , 50
CIRCLE ( 160 , 150 ) ,  50 : CIRCLE ( 480 , 150 ) , 50
```

is perfectly acceptable. Using spaces usually makes the code more readable. In the above program, the spaces have been used in two ways. The first is to separate the different arguments, so that within each statement the different values are clearly identifiable. The second is to allow corresponding arguments in different statements to be printed in columns. This makes it easier to compare the statements, and see how one differs from the others.

Note, however, that spaces cannot be placed within the keywords themselves. For example typing:

```
CIR CLE (320,100),100
```

produces an error.

However, spaces between the digits of numbers are automatically removed:

```
CIRCLE (3 20,100),100
```

is converted by BASIC to:

```
CIRCLE (320,100),100
```

If a line begins with a REM statement then the line is not executed. This provides a method of adding comments to your programs to explain what is happening. For example:

```
REM Centre circle
CIRCLE ( 320 , 100 ) , 100

REM Corner circles
CIRCLE ( 160 ,  50 ) , 50 : CIRCLE ( 480 ,  50 ) , 50
CIRCLE ( 160 , 150 ) , 50 : CIRCLE ( 480 , 150 ) , 50
```

Loading and Saving

To save your current program, either type:

```
SAVE
```

in the Output window or select the Save As option from the Project menu. A requester will appear asking:

```
Save program as:
```

Click on the box provided and type the name which you wish to give to your program, for example:

```
circles
```

Then click on the OK gadget. You should then hear the disc drive spring into life as your program is stored safely away on the disc.

Now if you type:

```
NEW
```

or select the New entry from the Project menu, the computer will carry out this instruction straight away without giving you a warning. This is because it knows that the current program has already been saved so that there is no risk of you losing it.

To load your program again type:

```
LOAD
```

into the Output window or select Open from the Project menu. Again a requester will be displayed asking:

```
Name of program to load:
```

Click on the box, type:

```
circles
```

and click on the OK gadget. The disc drive will spin briefly and you will be presented with the OK prompt. If you bring the List window to the front, you will see your program displayed inside it, ready to be edited or run etc.

You can type the whole of a SAVE or LOAD command in the Output window without using the requester, if you wish. For example:

```
SAVE "circles"
```

or:

```
LOAD "circles"
```

Note that the name of the program must be enclosed between double quotes.

Leaving BASIC

We will finish this chapter by exiting from BASIC, returning to the Workbench and tidying up. To do this either type:

```
SYSTEM
```

in the Output window or select the Quit option from the Project menu. You will then be back in the Workbench. Before you go on to other things, it is worthwhile, at this stage, tidying up after yourself. Close the window of the Extras Disk by clicking on the close gadget and then open it again by double clicking on the Extras Disk icon. This updates the display so that the new contents of the disc are shown. If you study the contents you should notice that the circles program has appeared. Rather than storing this program where it is, a better method is to create a drawer to keep all your programs in and to move it there. To do this:

- Open the workbench by double clicking on Workbench icon and replacing the disc in the disc drive if necessary.
- Copy the drawer labelled 'Empty' to the Extras by clicking on it, dragging it over to the Extras window and then releasing the mouse button. Again, you will have to swap discs if you have a single drive system.
- Activate the drawer by clicking on it and then rename it by selecting the Rename item from the Workbench menu and replacing the 'Empty' string by the name you want to use, for example 'Myprogs', followed by RETURN.
- Finally, move the circles program to this drawer by dropping the circles icon onto the drawer icon.

When you next enter BASIC, you can load the program by using the name of the drawer as follows:

```
LOAD "Myprogs/circles"
```

Similarly you can save further programs directly into this drawer in the same manner, for example:

```
SAVE "Myprogs/newprog"
```

Note that the '/' character is used to separate the drawer name from the file name in both cases.

In addition, you can obtain a list of all the contents of the drawer by typing:

```
FILES "Myprogs"
```

in the Output window. If you do this you should find two entries for each program, for example:

```
circles
```

and:

```
circles.info
```

The 'circles' file contains the actual BASIC program. The 'circles.info' file contains the definition of the icon used to illustrate this file.

2 : Starting Out

When trying to learn about a new topic, there are three very important words: 'Why', 'How' and 'What'. The introduction in this book aimed to answer the question 'Why should I bother learning how to program my Amiga?'. If you've read this far, you will hopefully be convinced that it's a good thing to do. The first chapter went on to deal with the question 'How do I use the AmigaBASIC system?'. Using the knowledge gained there, we can now move forward and start on the final question which is, 'What can I do with AmigaBASIC?'

BASIC is a very big topic, covering many areas. In this chapter we are going to look at two of the most rewarding ones graphics and sound. Making the computer draw a picture or play a tune is very satisfying and can be achieved by writing a straightforward program, just a few lines long. However, life is not all play! We will also be looking at some of the features of BASIC which will help you write well-structured, easy to read, code.

Points, Lines and Circles

The graphics from a single command has to be placed entirely within a single window. It is possible to create lots of windows and jump between them, with some commands sending their graphics to one and some to another. However, in this chapter we are going to be dealing with the default case in which all graphics output is sent to the BASIC Output window.

To perform any graphics command you need to provide coordinates to tell the computer where it is to place the point, end of line, centre of circle etc. Each output window is made up of 'pixels', which are rectangular dots. The pixel in the top left-hand corner of the Output window is defined to be at position (0,0). Increasing the x-coordinate by one moves a point across by one pixel, and increasing the y-coordinate by one moves it down by one pixel. The limits on the x and y-coordinates depend on the size of the Output window. The maximum theoretical value is 640 pixels horizontally by 256 pixels

vertically (200 on an American monitor) – this is the number of pixels which make up the whole screen. However, the BASIC Output window doesn't occupy the whole screen, so the maximum coordinate is about (617,185), even when the window is fully expanded.

To see how to produce points, lines and circles, type in and run the following program which draws a stick man:

```
CIRCLE (320, 50), 20
LINE   (320, 60) - (320,100)
LINE   (320,100) - (260,140)
LINE   (320,100) - (380,140)
LINE   (260, 70) - (380, 70)
PSET   (315, 50)
PSET   (325, 50)
```

Whenever a position is required, it must be given enclosed in brackets, with the x- and y-coordinates separated by a comma. The PSET command which plots a point, requires just a single position. The CIRCLE command takes one position which determines the centre of the circle, and another value which is used as the radius. The LINE command requires two positions, separated by a hyphen ('–'), which specify the ends of the line.

Instead of giving a position as an absolute location, you can instead supply it as an offset relative to the last graphics position used. To do this, place the keyword 'STEP' before the position. For example, the program above is equivalent to the following:

```
CIRCLE       ( 320, 50), 20
LINE   STEP (   0, 10) - STEP (   0, 40)
LINE   STEP (   0,  0) - STEP (- 60, 40)
LINE   STEP (  60,-40) - STEP (  60, 40)
LINE   STEP (-120,-70) - STEP ( 120,  0)
PSET   STEP (- 65,-20)
PSET   STEP (  10,  0)
```

The values can be interpreted as follows: The first line starts ten pixels below the centre of the circle and ends a further forty pixels below that.

The next line starts where the previous one finished and ends sixty pixels to the left and forty pixels down. The next line starts at an offset

of 60 pixels to the right and 40 pixels up from the end of the previous line. This coincides with where the previous line started, etc.

Using relative positions makes the program longer and, in some ways, more difficult to write, so why should we bother doing it? To answer this question, consider how you would have to alter each program if you decided to draw the man in a different position within the window, say 100 pixels to the left. In the first example, you would have to alter every x-coordinate by subtracting 100 from it. In the second example all you have to do is subtract 100 from the first position. The rest, since they are all relative, don't require changing.

Introduction to Variables

The version of the CIRCLE command which we have encountered so far has been fairly simple. However, this keyword can also be used to produce arcs and ellipses, in which case you have to supply a lot more information. In its fullest form it can end up looking like this:

```
CIRCLE (320,100),50,3,1,2,4
```

If you met this instruction in a program, it wouldn't be surprising if you had difficulty remembering what all the numbers represented. Without thumbing through a manual to look up its syntax you could very well mix up the parameters, and end up drawing a different shape to the one you were expecting.

It would be far easier to know what was going on if the statement looked more like this:

```
CIRCLE (xpos,ypos),radius,col,startang,endang,aspect
```

where 'xpos', 'ypos', etc represented the appropriate values.

You can make your program look like this by using 'variables'. A variable is something which has a name and a value associated with it. The name, for example: 'xpos' or 'radius' allows it to be identified and its value to be either set or read. This value can be changed and read as many times as you like.

Variable names can contain characters, digits or full stops. The rules for naming variables are as follows:

- The name may contain up to 40 characters.
- The name must start with a letter.
- The name may contain only numbers, letters and decimal points.
- The name must not be a BASIC reserved word, eg SIN or COLOR.

Therefore all the following names are allowed:

```
Y
ypos
YPOS
Ypos
Y.position
YPOS1
```

But these are not acceptable:

1posy	Doesn't begin with a letter
Y-pos	Contains a minus sign
POS	BASIC reserved word

Upper-case characters are treated as being equivalent to lower-case ones. For example 'xpos' refers to the same variable as 'Xpos' or 'XPOS'. If you enter a line of BASIC containing keywords in lower-case into the List window, when you press the RETURN key, these keywords are automatically converted into upper-case. Therefore all program listings will contain upper-case keywords. Hence it is a good idea to use lower-case letters for variable names, since this distinguishes them from the keywords and helps to make the program more readable.

The variables we require here all have numeric values associated with them, and so they are known as 'numeric variables'. The other type of variable is the 'string variable' which represents a string of characters – this we will deal with in later chapters. You can assign a value to a numeric variable as follows:

```
LET xpos = 320
```

or, since the use of the LET keyword is optional, just by typing:

```
xpos = 320
```

The value assigned to a numeric variable can be specified as a single number, as above, or the current value of another variable, or an 'expression'. For example:

```
xpos = 100
ypos = xpos
radius  = (xpos + ypos) / 4
```

'(xpos + ypos)/4' is an example of an expression; it is a sequence of numbers and variables together with 'operators' which act on them. The common operators are:

+	Add
-	Subtract
*	Multiply
/	Divide

Try the following program:

```
RANDOMIZE TIMER
xpos = 320
ypos = 100
radius = RND*60
CIRCLE (xpos,ypos),radius
```

This assigns values to the variables 'xpos', 'ypos' and 'radius', and then uses those values to draw a circle. The value assigned to radius is determined by the expression 'RND*60'. RND is a BASIC keyword which returns a random number in the range 0 to 1. Therefore the value assigned to radius will be somewhere in the range 0 to 60. The RANDOMIZE TIMER command causes the random number generator to be 'reseeded' with the number of seconds which have passed since

random numbers each time the program is run, thus giving you a different size circle each time.

Loops

So far, all the programs we have looked at have been executed in a linear fashion. That is to say, the computer started at the top of the program, executed each instruction in turn until it reached the end and then stopped. This is not always the most convenient way to write a program. Consider, for example, how to write a program to draw two random circles on the screen. One obvious way is as follows:

```
REM First circle
xpos = RND*620
ypos = RND*180
radius = RND*60
CIRCLE (xpos,ypos),radius
REM Second circle
xpos = RND*620
ypos = RND*180
radius = RND*60
CIRCLE (xpos,ypos),radius
```

This is fine for drawing two circles, but what would you do if you wanted to draw two hundred? It would be very inconvenient to have to repeat the circle drawing code two hundred times. BASIC has a solution to this problem. It supplies FOR and NEXT statements which may be used to execute a block of the program a specified number of times. These statements are placed so that they surround the block to be repeated. Try the following example:

```
FOR count = 1 TO 200
  xpos = RND*620
  ypos = RND*180
  radius = RND*60
  CIRCLE (xpos,ypos),radius
NEXT count
```

Note that the lines between the FOR and NEXT statements are indented by two characters. This makes it clear to anyone reading the program which lines are being repeated. It is not necessary to do this, but it is a good habit to get into. It is also very easy to do, since

pressing the TAB key inside the List window will move the cursor two characters to the right. Then, when you press the RETURN key, the cursor will move down a line and across so that it is positioned underneath the first character of the previous line. This means that all subsequent lines are indented by the same amount until you actively start a line at a different position.

The variable 'count' is called the 'control variable'. It is used to control the number of times the block is executed. In this case, count is set initially to one. When the NEXT statement is reached, count is increased by one and the block of code is repeated. This continues until count becomes greater than 200, in which case the computer will move on to the statement after the NEXT.

In the example given above, the control variable is used only to record the number of times the circle drawing code is executed. It is not referred to in this block. Therefore the actual values it takes are irrelevant. All that matters is the number of steps required to reach the upper limit. For example the statement:

```
FOR count = 1 TO 200
```

could be replaced by:

```
FOR count = 0 TO 199
```

or even:

```
FOR count = -100 TO 99
```

and the result would be the same.

You are allowed to refer to the control variable inside the block. For example, alter the program above to read as follows:

```
FOR count = 1 TO 200
  xpos = RND*620
  ypos = RND*180
  radius = count
  CIRCLE (xpos,ypos),radius
NEXT count
```

This example produces two hundred circles at random positions on the screen. The radius of the first is one, that of the second is two, and so on.

The amount by which the control variable changes each time round the loop is called the 'step size'. In the above example no step size was stated explicitly, so the default value of one was used. Other values can be used as follows:

```
FOR count = 1 TO 200 STEP 5
  xpos = RND*620
  ypos = RND*180
  radius = count
  CIRCLE (xpos,ypos),radius
NEXT count
```

This produces 40 circles at random positions. The radius of the first will be one, that of the second six, and so on. Note that, in this case, count will never take the value 200. The final value used will be 196.

The step size can be negative so that the control variable is decreased each time. For example:

```
FOR count = 200 TO 1 STEP -5
  x = RND*620
  y = RND*180
  radius = count
  CIRCLE (xpos,ypos),radius
NEXT count
```

Again this produces 40 circles, but in this case they gradually become smaller.

Using Colour

So far, our graphics have all been drawn in white. We are now going to look at how we can use some of the other colours which are available to us on the Amiga.

One method is to specify the colour to use by giving an extra argument to the PSET, LINE and CIRCLE commands. For example, try the following:

```
FOR count = 1 TO 20
  xpos = RND*620
  ypos = RND*180
  radius = RND*60
  rd4 = radius/4
  CIRCLE (xpos,ypos),radius,3
  PSET   (xpos-rd4,ypos),1
  PSET   (xpos+rd4,ypos),1
  LINE   (xpos-rd4,ypos+rd4) - (xpos+rd4,ypos+rd4),2
NEXT count
```

This draws the circles in colour three, the points in colour one and the lines in colour two. These numbers correspond to the following colours:

Colour 1	White
Colour 2	Black
Colour 3	Orange

The other colour number we could have used is colour 0:

Colour 0	Blue

However, this is the same as the background so any graphics we drew in it would not have been visible!

If you don't specify a colour, then the current 'foreground colour' will be used. By default this is colour one, which is why all the lines, circles and points we drew previously were drawn in white. You can set a different foreground colour using the COLOR command. For example, if you type:

```
COLOR 3
```

then all subsequent PSET and LINE commands etc, which don't specify a colour to use, will be drawn in red.

In addition to setting a different current foreground colour, you can also set a different current background colour by giving a second value to the COLOR command. For example:

```
COLOR 3,1
```

sets the current foreground colour to colour three (orange), and the current background colour to colour one (white).

Altering the current background colour has no immediate effect. The results of this action will only be seen when subsequent commands are given which use the background colour. For example:

```
COLOR 3,1
CLS
```

The COLOR command (note American spelling) sets the current background to colour one. The CLS command then clears the contents of the Output window, leaving it displayed in the current background colour.

The Palette

The bad news I have to give you at this point is that, under default conditions, the Output window can only display four different colours at once. This is why only four colour numbers, zero to three, have been used. Trying to use any other value would have produced an error. We will see, in a later chapter, how you can arrange to display more colours together in a window.

However, this doesn't mean that you are limited to producing pictures in white, black, blue and orange. You can use any four colours you like from the range which the Amiga provides.

The way in which other colours are obtained is by making the colour numbers represent different colours. For example, colour 0 doesn't have to remain as blue, you can define it to be green or grey or pink etc. To do this, you use the PALETTE command.

PALETTE takes four parameters. The first is the colour number, which is a whole number, currently limited to being in the range 0–3. The other three are fractional values in the range 0 to 1.00, which specify the amounts of red, green and blue you want the colour to contain. For example:

```
PALETTE 2, 0.1, 0.5, 0.9
```

defines colour two to contain a small amount of red, a moderate amount of green and a large amount of blue, the result being a shade of purple.

The general rule is: the higher the number, the brighter the colour. Keeping the amounts of each of the three colours equal produces a grey scale between black (all three set to zero) and white (all three set to one). By altering the ratios between the amounts of red, green and blue, all the other colours can be obtained. Table 2.1 below attempts to describe the colours produced by certain combinations of red, green and blue.

	Red	Green	Blue
Black	0.00	0.00	0.00
Dark grey	0.30	0.30	0.30
Light grey	0.70	0.70	0.70
White	1.00	1.00	1.00
Red	1.00	0.00	0.00
Green	0.00	1.00	0.00
Blue	0.00	0.00	1.00
Yellow	1.00	1.00	0.00
Cyan	0.00	1.00	1.00
Magenta	1.00	0.00	1.00
Pink	1.00	0.40	0.40

Table 2.1. Colour Combinations.

Note that, re-defining the colour assigned to a colour number, affects all the graphics currently displayed which are drawn using that colour number, as well as any subsequent graphics produced using it.

Rectangles

The LINE command isn't limited to producing lines, it can also be used to draw rectangles. To draw a rectangle instead of a line, add an extra parameter, 'b', to the end of the statement. Then the pairs of coordinates will be taken as being the opposite corners of the box. For example:

```
LINE (100,80) - (200,145),1,b
```

Alternatively you can add a 'bf' instead and this will produce a 'solid' rectangle, ie the interior of it will be filled in:

```
LINE (100,80) - (200,145),1,bf
```

Plotting solid rectangles enables you to see the different colours which are available more clearly. Try typing in and running the following program:

```
COLOR 1,0
PALETTE 0,0,0,0
CLS
RANDOMIZE TIMER
LINE ( 80,20) - (560,160),1,bf
LINE (180,40) - (460,140),2,bf
LINE (280,60) - (360,120),3,bf
FOR count = 1 TO 20
  red = RND
  green = RND
  blue = RND
  PALETTE 3, red, green, blue
  PALETTE 2, red*.8, green*.8, blue*.8
  PALETTE 1, red*.4, green*.4, blue*.4
  FOR delay = 1 TO 5000 : NEXT delay
NEXT count
```

The first three commands ensure that the background is displayed in black. Then the program reseeds the random number generator and draws three solid rectangles, gradually decreasing in size, inside each other. Each of these is drawn in a different colour number: one, two or three. Finally, the program enters a loop which it repeats twenty times.

Inside the loop, the program chooses random values for the amount of red, green and blue to be used for a particular colour. Colour number three is assigned this colour. Colour number two is assigned a slightly darker shade by reducing all the values to 80% of their original size. Similarly colour number one is assigned a darker shade still at 40% of the original. The innermost FOR...NEXT loop is used as a 'delay loop'. Making the computer count from 1 to 5000, before changing the palette again, ensures that each set of colours is displayed for a reasonable length of time.

At the end of this sequence, you could find yourself with an unreadable colour for the text. The screen should revert back to the default palette settings when the program has finished running. However, you may need to remind it to do so. To do this:

- Select the List window.
- Edit the program, for example by typing a space.
- Select the Output window.

You should then be returned to white text on a blue background.

Loops within Loops

Using two loops at once, one within the other, is quite often a very useful thing to be able to do. Consider the problem of drawing circles, centred at regular intervals, on the screen. This requires one loop to step through the x-coordinates. Then for each x value, another loop is needed to step through the y-coordinates. For example:

```
FOR xpos = 20 TO 600 STEP 20
  FOR ypos = 20 TO 160 STEP 20
      CIRCLE (xpos,ypos), 5
  NEXT ypos
NEXT xpos
```

This produces circles of radius five placed 20 units apart. The same result could be obtained in a slightly different way:

```
FOR ypos = 20 TO 160 STEP 20
  FOR xpos = 20 TO 600 STEP 20
      CIRCLE (xpos,ypos), 5
  NEXT xpos
NEXT ypos
```

The only difference between these two programs is the order in which the circles are drawn. In the first one they are drawn a column at a time, starting from the left-hand side of the screen. In the second they are drawn a row at a time, starting from the top of the screen.

In both cases, one of the loops is contained wholly within the other. This loop is said to be 'nested'. Overlapping loops are not allowed. For example:

```
FOR xpos = 20 TO 600 STEP 20
  FOR ypos = 20 TO 160 STEP 20
      CIRCLE (xpos,ypos), 5
  NEXT xpos
NEXT ypos
```

would produce the error message:

```
NEXT without FOR
```

At first sight, this message may seem to be confusing since there appears to be the correct number of FOR and NEXT statements. The way BASIC comes to its conclusion about what is wrong is as follows:

1) The xpos loop was started first.

2) The ypos loop was started second.

3) Since one loop must be totally within another, the ypos loop must be completed before the xpos loop can step on to its next value.

4) The NEXT xpos statement is found before the ypos loop has ended.

5) Therefore the NEXT ypos statement is missing.

Since BASIC insists that each NEXT must apply to the most recent FOR, it is not necessary to state the control variable after a NEXT. There is only one loop which it can correctly refer to. For example:

```
FOR xposXT
```

is taken as being equivalent to:

```
FOR xpos = 20 TO 600 STEP 20
  FOR ypos = 20 TO 160 STEP 20
      CIRCLE (xpos,ypos), 5
  NEXT ypos
NEXT xpos
```

Arcs and Ellipses

We mentioned above that the CIRCLE command can be used for producing arcs and ellipses as well as circles. The parameters which it can take are as follows:

```
CIRCLE (xpos,ypos),radius,col,startang,endang,aspect
```

The starting and ending parameters are the start and end angles of the arc. These must be supplied in radians not degrees. Figure 2.1. (below) shows the values of the different positions around the perimeter:

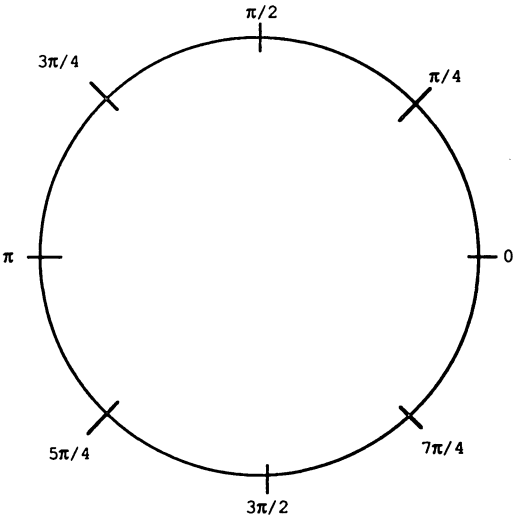


Figure 2.1. The starting and ending parameters are the start and end angles of the arc. These must be supplied in radians not degrees.

Note that the arc will be drawn in an anti-clockwise direction. Therefore, if you give a start angle of 0 and an end angle of π then the top half of the circle will be drawn. Conversely, if you give a start angle of π and an end angle of 0 the bottom half of the circle will appear.

If either of the angles is negative, then it will be treated as though it was positive for calculating the relevant position, but a line will be drawn connecting that particular end of the arc to the centre.

The following program demonstrates some of the different combinations you can use:

```
pi = 3.14159
end1 = pi/4
end2 = 3*pi/4
CIRCLE (110, 50),60,1, end1, end2
CIRCLE (250, 50),60,1,-end1, end2
CIRCLE (390, 50),60,1, end1,-end2
CIRCLE (530, 50),60,1,-end1,-end2
CIRCLE (110,130),60,1, end2, end1
CIRCLE (250,130),60,1, end2,-end1
CIRCLE (390,130),60,1,-end2, end1
CIRCLE (530,130),60,1,-end2,-end1
```

The aspect determines the shape of the circle/ellipse. If you specify an aspect ratio of one, then the shape will measure the same number of pixels horizontally as it does vertically. However, since the pixels are not square, an ellipse will be produced. In fact, the pixels are roughly twice as high as they are wide, so the height of the ellipse will appear to be approximately twice that of the width.

Increasing the aspect ratio to two produces a shape which is the same height as the previous ellipse but half the width. Hence it will appear about four times as high as it is wide. Decreasing it to 0.5 produces a shape which is the same width as the original ellipse but is half the height. This makes it roughly circular.

Therefore, for a 'circle' whose radius is 'rad', the following applies:

	no.pixels vertically	**no.pixels horizontally**
If aspect = 1	rad	rad
If aspect > 1	rad	rad/aspect
If aspect < 1	rad*aspect	rad

If you don't specify an aspect, the default value of 0.44 is used. On standard American monitors this produces circles. Unfortunately, British monitors are different and the default value actually produces ellipses which are wider than they are high. A more accurate value to use in Britain is 0.56.

The following program produces a display of ellipses enclosed within a circle:

```
cirval = .56
xpos   = 320
ypos   = 90
rad1   = 120
rad2   = 120*cirval
CIRCLE(xpos,ypos),rad1,1,,,cirval
FOR ratio = 2 TO 10
  asp1 = cirval/ratio
  asp2 = cirval*ratio
  CIRCLE(xpos,ypos),rad1,1,,,asp1
  CIRCLE(xpos,ypos),rad2,1,,,asp2
NEXT ratio
```

All the ellipses are centred at the same position. The first one has a radius of 120 and an aspect of 0.56. Using the table above, this means that its dimensions, in pixels, are as follows:

```
height = radius*aspect = 120*0.56
width  = radius        = 120
```

Thus it appears to be circular.

The program then loops through different values, between two and 10, for the ratios of the width to the height. For each value it draws two ellipses. The first has a radius of 120 and an aspect of 0.56/ratio (which is always less than one). Therefore its dimensions are:

```
height = radius*aspect = 120*0.56/ratio
width  = radius        = 120
```

This means that they always touch the circle at the left and right-hand sides but the heights of successive ones decrease.

The second has a radius of 120*0.56 and an aspect of 0.56*ratio (which is always greater than one). Its dimensions are:

```
height = radius                = 120*0.56
width  = radius/aspect         = 120*0.56/(0.56*ratio)
                               = 120/ratio
```

This means that they always touch the circle at the top and bottom but the widths of successive ones decrease.

Making Sounds

Sound is another area which gives 'visible' results. The Amiga comes equipped with four sound channels and, using BASIC, you can produce sounds on any or all of these. In this chapter we are going to concentrate on producing a single sound at a time. Later in this book we'll look at how the channels can be synchronised and different types of sound created.

To produce a note, you have to supply two pieces of information: its frequency and its duration. These characteristics of a note are examined below.

Frequency

The sounds we hear from musical instruments, people's voices, machinery etc are all composed of mixtures of waves. A pure, musical sound contains just one wave, whereas discordant noise from machinery contains many different waves, all jumbled together. Each wave oscillates at a different frequency and, the higher the frequency, the higher the pitch of the sound. For example Figure 2.2 shows the difference between high and low:

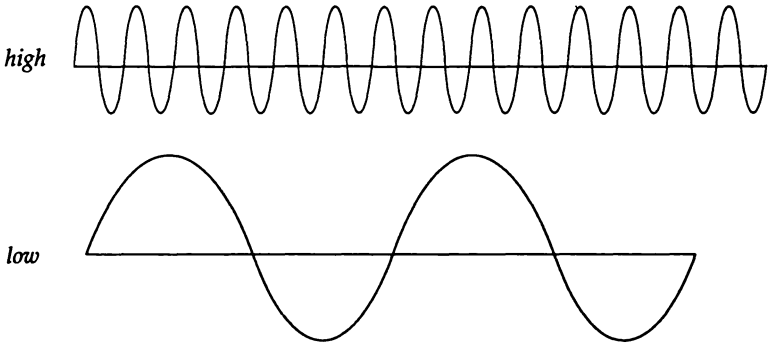


Figure 2.2. The difference between high and low frequencies.

The frequency is measured in 'Hertz' or 'cycles per second'. Table 2.2. shows the frequencies of the notes in the octave, starting at middle C.

Note	Frequency
C	523.25
C#	554.37
D	587.33
D#	622.25
E	659.26
F	701.00
F#	740.00
G	783.99
G#	830.61
A	880.00
A#	932.33
B	993.00
C	1046.50

Table 2.2. Note Frequency.

If you double the frequency of a note, you produce the note which is exactly one octave higher. Consequently, halving the frequency gives a note one octave lower. Using this rule and the table above, you should be able to calculate the frequency of any note. To speed up this process, the Figure 2.3 below gives the frequencies of the notes covered by the treble and bass staves.

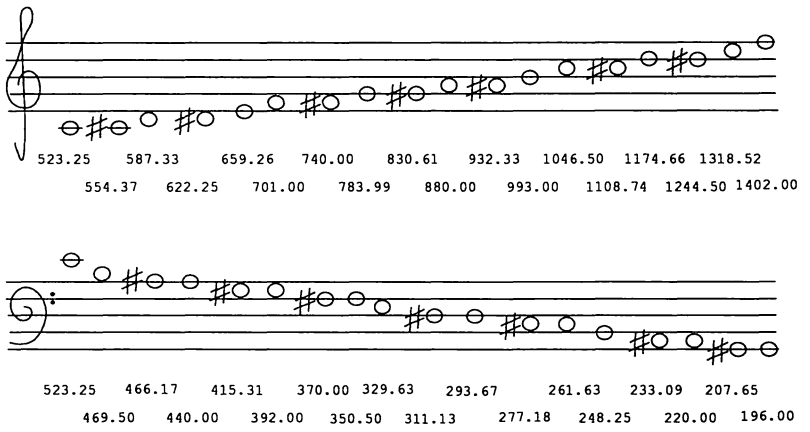


Figure 2.3. Frequencies of the notes covered by the treble and bass staves.

The computer is capable of producing sounds which range from 20 Hertz to 15000 Hertz. This gives you a range of more than nine octaves. Any values you give will be rounded so that they lie in this range. This means that specifying a frequency which is less than 20 Hertz will produce a 20-Hertz sound, ie the lowest note possible. Similarly specifying a note greater than 15000 Hertz produces the highest note possible.

Duration

The duration of a note is the length of time the note lasts. A duration of 18.2 will produce a note which lasts for one second. There is a simple, linear relationship between the value of the duration and the number of seconds the note sounds for. Therefore doubling this value will give a note lasting two seconds, trebling it will give one lasting three seconds etc. The values which are allowed are those in the range 0 (no time at all) to 77 (just under four and a quarter seconds). The following table lists the standard ranges for the most common tempos. For each it gives the number of beats a minute and, calculated from this, the duration value of each beat.

Presto	168 - 208	5.3 - 6.5
Allegro	120 - 168	6.5 - 9.1
Moderato	108 - 120	9.1 - 10.1
Andante	76 - 108	10.1 - 14.4
Adagio	66 - 76	14.4 - 16.5
Larghetto	60 - 66	16.5 - 18.2
Largo	40 - 60	18.2 - 27.3

Producing a sound

The keyword used for producing a sound is, appropriately enough, SOUND.

In its simplest form, the syntax it takes is as follows:

```
SOUND frequency, duration
```

For example:

```
SOUND 523.25, 18.2
```

produces a middle C lasting for one second.

You can also specify how loud the note is to be. This should be a value in the range 0 (silent) to 255 (loudest). If you don't provide a value, the default of 127 which is the middle of the range is used. Try the following program:

```
SOUND  523.25, 9.1,  31
SOUND  587.25, 9.1,  63
SOUND  659.26, 9.1,  95
SOUND  701.00, 9.1, 127
SOUND  783.99, 9.1, 159
SOUND  880.00, 9.1, 191
SOUND  993.00, 9.1, 223
SOUND 1046.50, 9.1, 255
```

Note that, when you type this into the list window, the '.00' string will be replaced by a '!'. The zeros after the decimal point are irrelevant so BASIC has removed them but has left the '!' to remind you that the number is a floating point number not an integer.

When you run this program, the computer will play the scale of C major with each of the notes being louder than the previous one.

An example tune

Sound is another area where variables and expressions are a great help. Instead of numbers like '523.25' which are meaningless to most people, you can use letters to represent the notes. For example, you might start the variable name with the letter of the note, 'a' - 'g'. Then the second letter could be a 'n', 's' or 'f' to specify if a natural, sharp or flat is wanted. Finally the last character could be a digit to indicate which octave it is.

It is a good idea to use expressions for the durations of the notes. For example you could specify the duration of the different types of notes as follows:

```
quaver      0.5   * dur
crotchet    1.0   * dur
minim       2.0   * dur
semibreve   4.0   * dur
```

Then you can alter the tempo of the tune just by assigning a different value to the variable 'dur'.

The following program shows these tips in action:

```
REM Set up variables

g3n =  783.99
a4n =  880.00
b4n =  993.00
c4n = 1046.50
d4n = 1174.70
e4n = 1318.50
f4s = 1480.00
g4n = 1568.00

dur = 8.0

REM Produce the sounds

SOUND g3n,2.0*dur
SOUND a4n,0.5*dur
SOUND c4n,0.5*dur
SOUND b4n,0.5*dur
```

```
SOUND a4n,0.5*dur
SOUND d4n,1.0*dur
SOUND d4n,1.0*dur
SOUND d4n,0.5*dur
SOUND e4n,0.5*dur
SOUND b4n,0.5*dur
SOUND c4n,0.5*dur
SOUND a4n,1.0*dur
SOUND a4n,1.0*dur
SOUND a4n,0.5*dur
SOUND c4n,0.5*dur
SOUND b4n,0.5*dur
SOUND a4n,0.5*dur
SOUND g3n,0.5*dur
SOUND g4n,0.5*dur
SOUND f4s,0.5*dur
SOUND e4n,0.5*dur
SOUND d4n,0.5*dur
SOUND c4n,0.5*dur
SOUND b4n,0.5*dur
SOUND a4n,0.5*dur
SOUND g3n,2.0*dur
SOUND a4n,0.5*dur
SOUND c4n,0.5*dur
SOUND b4n,0.5*dur
SOUND a4n,0.5*dur
SOUND d4n,1.0*dur
SOUND d4n,1.0*dur
SOUND d4n,0.5*dur
SOUND e4n,0.5*dur
SOUND b4n,0.5*dur
SOUND c4n,0.5*dur
SOUND a4n,1.0*dur
SOUND a4n,1.0*dur
SOUND a4n,0.5*dur
SOUND c4n,0.5*dur
SOUND b4n,0.5*dur
SOUND a4n,0.5*dur
SOUND g3n,0.5*dur
SOUND d4n,0.5*dur
SOUND a4n,0.5*dur
SOUND b4n,0.5*dur
SOUND g3n,1.0*dur
```

3 : Interacting with the User

Virtually all commercially available programs 'interact' with the user. That is, they ask the user to supply information or instructions and then respond in different ways, depending on the particular piece of data they are given.

An obvious example is provided by the type of program aimed at testing a child's maths ability. These write a simple sum on the screen and wait for the child to type in an answer. Then they typically respond by printing 'right – well done' or 'wrong – try again', depending on whether the answer was correct or not.

Arcade games, such as Space Invaders, display a different form of behaviour. These don't stop and wait for you to type something. Instead they enter a particular mode of behaviour and carry on in the same manner until you interrupt them by pressing a key or button. Then they temporarily stop what they were doing and react to your input.

The chapter looks at these methods of interaction in more detail and shows how a program can act in different ways, depending on the input it gets.

Handling Text

Before we can contemplate having a question and answer session with the computer, we have to know how to handle text. We saw in the previous chapter how numeric variables can be used to hold numbers. In a similar manner, BASIC provides string variables, which may be used to store strings of characters, ie words and phrases. Each string can be up to 255 characters long and can contain upper– and lower-case letters, spaces, punctuation characters etc. The following shows a few examples of strings being assigned to string variables.

```
name1$ = "Winston"
question$ = "How old are you?"
age$ = "21"
address$ = "10, Downing Street; London."
```

Note that, in each case, the variable name ends in a '$'. This identifies it as being a string variable rather than a numeric variable. Numbers and strings have to be kept separate from each other. You cannot assign a number to a string variable or vice versa. In the example above, age$ was assigned a string containing the two characters '2' and '1', and not the number 21.

The simplest way to write a message in the Output window is to use PRINT. For example:

```
name1$ = "Tom"
PRINT "Hello ";name1$,"How are you?"
```

outputs the following:

```
Hello Tom      How are you?
```

A question mark can be used instead of the word PRINT in a PRINT statement in order to save time. Therefore you could enter the above program as:

```
name1$ = "Tom"
? "Hello ";name1$,"How are you?"
```

Note that, when you press RETURN at the end of the line, the question mark will be converted into the word PRINT.

The PRINT statement can be followed by a list of items to be printed, separated either by commas, semi-colons or spaces. The effect of a semi-colon or space is to make the next item start immediately after the previous one. The effect of a comma is to print the next item at the start of the 'next zone'. By default, a zone is 14 characters wide. Therefore a comma will cause the item to be printed starting at character position one or 15 or 29 or 43 etc, depending on how far across it is currently.

A particular number of spaces can be output in different ways. For example, two predefined strings 'str1$' and 'str2$' can be separated by 10 spaces either by typing:

```
PRINT str1$ "          " str2$
```

or more simply:

```
PRINT str1$ SPC(10) str2$
```

SPC is a keyword which can only be used as part of a PRINT statement. Its argument specifies the number of spaces to be output and can be any number in the range 0 to 255.

The characters ',' and ';' can also be used to determine where subsequent PRINT statements start printing. If the list of items is followed by a semi-colon or comma, then the next PRINT statement will start printing on the same line. Otherwise, a carriage return will be printed, and so the next PRINT statement will start at the beginning of the next line down. For example:

```
PRINT "Hello ";
PRINT "there"
PRINT "John"
```

produces:

```
Hello there
John
```

Asking for Input

We often want a program to stop and wait for the user to type something. This is very straightforward with BASIC. The following program shows how it can be done:

```
PRINT "What is your name";
INPUT name1$
PRINT "Pleased to meet you ";name1$
```

When you RUN this program, the INPUT command prints a question mark on the screen and waits for you to type in a string and press the Return key. For example, you might type:

```
Fred
```

The string you type is assigned to the variable 'name1$', and the computer will respond by printing:

```
Pleased to meet you Fred
```

In the above program, the first PRINT statement was used to print a message on the screen indicating the type of response which was required. Without it, you would be presented with just a question mark. Since you have seen the program, you know what it expects you to type. However, anyone not knowing what the program was trying to do would have no idea what sort of information to enter. Therefore giving a message is very important. This message can be incorporated into the INPUT instruction as follows:

```
INPUT "What is your name ";name1$
PRINT "Pleased to meet you ";name1$
```

Note that a question mark is automatically given after the INPUT string. In some cases, you may wish to print out a message which is not a question. If this is so, you can suppress the question mark by replacing the semi-colon by a comma:

```
INPUT "Enter your name :",name1$
PRINT "Pleased to meet you ";name1$
```

Try running this program and typing:

```
Fred
```

You should find that the message printed is just the same as before, because the spaces you typed before the first character have been ignored. Now run it one more time and type:

```
Smith, Fred
```

This will produce the message:

```
?Redo from start
```

This is because the comma is treated as a separator between two items of data. Therefore the computer was expecting you to type in one string and you have actually given it two: "Smith" and "Fred". The message means that the computer wants you to re-enter your response.

If you require more than one piece of information, you can use several INPUT statements, as in the following program:

```
INPUT "Enter your surname :",surname$
INPUT "Enter your first name :",name1$
PRINT "Pleased to meet you ";name1$;" ";surname$
```

In this case you will be expected to enter two strings separately, for example:

```
Smith
Fred
```

and you will be rewarded with the message:

```
Pleased to meet you Fred Smith
```

An alternative method is to ask for several pieces of information to be given at once by placing several variable names, separated by commas, after the INPUT string. The above program can be modified to demonstrate this as follows:

```
INPUT "Enter your surname and first
name:",surname$,name1$
PRINT "Pleased to meet you ";name1$;" ";surname$
```

This program expects two strings, separated by commas, to be given on the same line. In this case the input:

```
Smith, Fred
```

is correct, whereas if you type just:

```
Fred
```

the '?Redo from start' message will be generated because you have entered too few strings.

Acting on Information Received

Once you, as the programmer, have been given some information by the user, you will no doubt want to do something with it. The above examples used the strings given to personalise a message. This is a very nice touch, often used to make the computer appear more 'human' and approachable. However, you might find that you want to do different things depending on what the user typed. BASIC provides the answer to this in the form of an IF statement, which enables the computer to make a choice about whether or not to execute a statement or group of statements. In its simplest form this can be used as follows:

```
INPUT "What is 2 + 2"; ans%
IF ans% = 4 THEN PRINT "Right - well done"
```

The IF is followed by a 'conditional expression'. This is an expression which gives the value true or false. If the result of the conditional expression is true the computer executes the statement after the THEN. In the example shown, the conditional expression is true when ans% is equal to four, and is false otherwise. If the user types four then the message 'Right – well done' will be printed.

Note that, in this example, the variable name is terminated by a '%'. This acts like the '$' symbol to denote that the variable is of a particular type. But whereas the '$' indicates that a variable is a string variable, '%' indicates that it is an 'integer variable'. Integer variables are a subclass of numeric variables which can hold only whole numbers.

By default, a variable name without a special character on the end is treated as a 'real variable', which can hold both whole and fractional numbers. One disadvantage of real variables is that they don't hold numbers totally accurately. If the result of a calculation means that a real variable should contain the value seven, then it might well actually contain 6.999999 or 7.000001. Therefore, you should never test

two real numbers for equality, since you cannot rely on them being exactly the same.

Many of BASIC's statements act on integers. For example, the graphics commands use integer coordinates. However, in most cases, these statements will happily accept real numbers and use the nearest integer value. Strictly speaking though, the programs we have written such as:

```
FOR count = 1 TO 200
  xpos = RND*620
  ypos = RND*180
  radius = RND*60
  CIRCLE (xpos,ypos),radius
NEXT count
```

should really be written as:

```
FOR count% = 1 TO 200
  xpos% = INT(RND*620)
  ypos% = INT(RND*180)
  radius% = INT(RND*60)
  CIRCLE (xpos%,ypos%),radius%
NEXT count%
```

where INT is a function which takes a real number as an argument and returns the largest integer less than or equal to it.

Returning now to the main issue of:

```
INPUT "What is 2 + 2"; ans%
IF ans% = 4 THEN PRINT "Right - well done"
```

If the user types any number other than four, then nothing will happen. This isn't very informative for the poor old user. What we really ought to do is to give a different message if the answer is wrong. To do this we can add an ELSE clause onto the end of the line:

```
INPUT "What is 2 + 2"; ans%
IF ans% = 4 THEN PRINT "Right - well done" ELSE PRINT
"Wrong"
```

Now, if the answer is correct, the message of congratulations will be given as before. But if the answer is wrong, the rather terse message 'Wrong' will be printed on the screen.

The THEN and ELSE keywords can be followed by more than one statement:

```
INPUT "What is 2 + 2"; ans%
IF ans% = 4 THEN PRINT "Right - well done" ELSE PRINT
"Wrong":BEEP
```

This time, giving the wrong answer will print the message 'Wrong', produce a short sound and flash the screen.

You can have as many statements as you like following the THEN and ELSE keywords, provided that they will all fit on one BASIC line. (A line can contain a maximum of 255 characters.) However, you will probably find that the program becomes difficult to understand well before this limit is reached. Unless the actions, to be performed following the THEN and ELSE, are short and simple it is better to use the second version of the IF statement:

```
INPUT "What is 2 + 2"; ans%
IF ans% = 4 THEN
  PRINT "Right - well done"
ELSE
  PRINT "Wrong"
  BEEP
END IF
```

This version is distinguished from the first by the fact that the THEN has nothing following it on the same line. In this version, the statements which are executed conditionally are split over several lines of the program. Depending on the result of the conditional expression, either the block of statements occurring between the THEN and the ELSE, or between the ELSE and the END IF, are executed. Note that the END IF is vital. Without it, BASIC cannot tell which statements are under the influence of the IF, and so gives an error message.

This version of the IF statement can be taken one step further by introducing 'ELSEIF' blocks. For example:

```
INPUT "What is 2 + 2"; ans%
IF ans% = 4 THEN
  PRINT "Right - well done"
ELSEIF ans% = 5 THEN
  PRINT "Your answer is one too many"
ELSEIF ans% = 3 THEN
  PRINT "Your answer is one too few"
ELSE
  PRINT "Wrong"
  BEEP
END IF
```

Now, if ans% is not equal to four, a further test is made to see if ans% has the value five. If so, the message 'Your answer is one too many' is printed. If not, then yet another test is made, this time to see if ans% is three. If so the message 'Your answer is one too few' is printed. It is only if all the tests produce the result false (ie the answer was not three, four or five) that the statements after the ELSE are executed.

Looking for Input

INPUT is fine if you want your program to stop and ask for a piece of information. But what do you do if you don't want the program to stop and give a prompt, you just want to see if the user has pressed a key. The answer is to use INKEY$. This returns either the null string, "", if nothing has been pressed, or a string containing the first character read from the keyboard buffer otherwise. For example, type in the following (leaving out the comments if you wish) :

```
PALETTE 0,0,0,0    : REM Colour 0 black
PALETTE 1,1,0,0    : REM Colour 1 red
PALETTE 2,0,1,0    : REM Colour 2 green
COLOR 1,0          : REM Select colour 0 for background
CLS                : REM and clear the screen
xpos = 320         : REM Start off roughly in the
ypos = 100         : REM centre of the window
col = 1            : REM and in red
FOR loop = 1 TO 5000
  a$ = INKEY$
  IF a$ <> "" THEN
       col = 3 - col
       IF a$ = "x" THEN xpos = xpos + 10: col = 2
       IF a$ = "z" THEN xpos = xpos - 10: col = 3
```

```
  END IF
CIRCLE (xpos,ypos),20,col,,,.56
NEXT
```

Now run this program. It will start by drawing a red circle on a black background. If you press any key then the circle will switch to being green, press another and it will change back to red again. Now try pressing the 'x' key. (Note that only a lower-case x will be responded to – an upper-case one will be ignored.) When you do, INKEY$ will return the string "x" to a$ and xpos will be increased, so that the circle will be drawn to the right of the previous one. Similarly, pressing 'z' will cause the circle to move to the left, '#' to move up and '/' to move down.

Note that, if you hold one of the keys down for a while, when you let go the circles will carry on moving in the direction you selected. This is because the computer frequently checks the keyboard and, if there is a key held down it places this character in the keyboard buffer. These characters are removed one at a time by INKEY$. Since characters can be put in the keyboard buffer more frequently than INKEY$ is called to remove them, then the buffer gradually builds up a backlog of characters. When you eventually let go, it takes a while for the program to clear this backlog. If you hold a key down for too long then the buffer becomes full and characters are lost.

Conditional Loops

Often, you don't want to use INKEY$ just once or for a fixed length of time. Instead you want to keep on using it until it finds a character. For example, a game must continually check to see if the user has pressed a key. While the keyboard buffer is empty the game can get on with dealing with its side of the action. However, as soon as it finds a character it must respond appropriately.

To carry out this kind of action you need to construct a WHILE...WEND loop. As long as a particular condition is true, the statements inside the loop will be executed. But soon as the condition becomes false, the program moves on to execute the statements after the end of the loop. A simple example illustrating this is given below:

```
WHILE INKEY$ = ""
```

```
  xpos = RND*620
  ypos = RND*180
  radius = RND*60
  CIRCLE(xpos,ypos),radius
WEND
```

This program continues plotting random circles on the screen until you press a key.

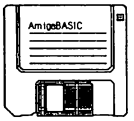
This concept can be used to amend the circle moving program we had above. Change it to the following:

```
PALETTE 0,0,0,0    : REM Colour 0 black
PALETTE 1,1,0,0    : REM Colour 1 red
PALETTE 2,0,1,0    : REM Colour 2 green
COLOR 1,0          : REM Select colour 0 for background
CLS                : REM and clear the screen
xpos = 320         : REM Start off roughly in the
ypos = 100         : REM centre of the window
col = 1            : REM and in red
xoff = 0           : REM No movement wanted
yoff = 0           : REM initially
WHILE xpos > 0 AND xpos < 620 AND ypos > 0 AND ypos <
180
  a$ = INKEY$
  IF a$ <> "" THEN
       col = 3 - col
       IF a$ = "x" THEN xoff =  1 : yoff =  0
       IF a$ = "z" THEN xoff = -1 : yoff =  0
       IF a$ = "#" THEN xoff =  0 : yoff = -1
       IF a$ = "/" THEN xoff =  0 : yoff =  1
  END IF
  xpos = xpos + 10*xoff
  ypos = ypos +  5*yoff
  CIRCLE (xpos,ypos),20,col,,,.56
WEND
END
```

Every time round the WHILE...WEND loop, the position of the centre of the CIRCLE is updated by adding on an x-offset and y-offset. Initially, these offsets are zero, so the circle stays in the centre of the screen. Pressing one of the direction keys assigns a non-zero number to either xoff or yoff, so the circle starts moving in a particular direction. It will continue moving in that direction until a different

window, which causes the WHILE loop to terminate and so ends the 'game'.

This program is better than the previous version in that it only requires you to press a direction key once to make the circle move, rather than having to press it continuously. This means that characters will not build up in the buffer and so the response to a key press is immediate, rather than delayed.



4 : Writing Large Programs

The larger the the programs you produce, the more disciplined you must be as you write them. To prevent too many bugs creeping in, you have to keep the program comprehensible. This means that variable names must be meaningful, comments must be used when necessary, and the program must be well structured so that the flow of control is easy to follow.

This chapter looks at some of the techniques which can be used to help achieve these ideals. And, since some programs will inevitably contain bugs, the ways in which you can track bugs down and eliminate them.

Coping with Variables

Consider the case of wanting to read in the surnames of thirty different children. You could do this as follows:

```
INPUT "Please give me a name: ",surname1$
INPUT "Please give me a name: ",surname2$
INPUT "Please give me a name: ",surname3$
.....................................
INPUT "Please give me a name: ",surname30$
```

However, this is a very long winded way of going about things. The way it should be done is with a FOR loop. What currently prevents a loop being used is the fact that the variable name is different in each case.

Arrays

There is a way around this problem though, and that is to use an 'array'. Arrays are groups of variables which share the same name. For example, you can define an array called 'surname$' which holds thirty different strings.

The individual members of an array are called 'elements'. They are identified by a 'subscript'. This is an integer indicating the element's position within the array. The lowest value which a subscript may

have is called the 'lower bound' and the highest value is called the 'upper bound'.

Normally, the first element in an array has a subscript of 0, the second a subscript of one etc. For example, for our array 'surname$':

```
surname$(0) is the first element
surname$(1) is the second element
.......
surname$(n) is the (n+1)th element
```

Often it is more convenient for the first element of an array to have the subscript one. You can specify that this is to be the case by using the statement:

```
OPTION BASE 1
```

This would mean that the elements of 'surname$' become:

```
surname$(1) is the first element
surname$(2) is the second element
......
surname$(n) is the nth element
```

OPTION BASE can only take the values zero or one. Therefore the first element has to have a subscript of zero or one.

Dimensioning and Assigning to an Array

If an array is to contain more than ten elements, you need to tell BASIC how big it is to be. You do this by using a DIM statement. For example:

```
OPTION BASE 1
DIM surname$(30)
```

allocates space in the computer's memory for thirty string elements, each called 'surname$', but each having a different subscript, one to 30.

Arrays may hold values of any type, ie floating point numbers, integers or strings. For example:

```
OPTION BASE 1
DIM temperature(21)
```

allocates space for 21 floating point numbers.

The DIM statement initialises each of the elements of an array. If it is a numerical array the elements are set to zero. If it is a string array the elements are set to the null string. The elements may then be individually assigned values, just like any other variables. For example:

```
temperature(1) = 20.5
temperature(3) = (temperature(1) + temperature(2))/2
```

The subscript need not be specified as a number. Instead, a variable can be used. For example, we can go back to our original problem and solve it as follows:

```
OPTION BASE 1
DIM surname$(30)
FOR person% = 1 TO 30
  INPUT "Please give me a name: ",surname$(person%)
NEXT
```

Any arithmetic expression may be used as a subscript. Since the subscripts can only be integers, any expression which gives a floating point result has the number rounded to the nearest integer value.

Multi-dimensional Arrays

The examples shown above are of 'one-dimensional' arrays, ie they may be thought of as a line of variables. More dimensions may be used by providing more subscripts to identify an individual variable. For example, with two-dimensional arrays individual variables are identified by two subscripts.

A two-dimensional array may be defined as follows:

```
OPTION BASE 1
DIM colour%(80,40)
```

This allocates space for 3200 elements, each called colour% and each identified by two subscripts:

```
colour%( 1,1) colour%( 1,2) colour%( 1,3)..colour%( 1,40)
colour%( 2,1) colour%( 2,2) colour%( 2,3)..colour%( 2,40)
............  ............. ............  .. ..............
colour%(80,1) colour%(80,2) colour%(80,3) .. colour%(80,40)
```

Arrays may have as many dimensions as you like up to a maximum of 255. However, one-, two- and three-dimensional arrays are the most useful.

The elements of a two-dimensional array can be thought of as the positions on a piece of paper or the screen. Each position is at a certain distance from one of the sides (given by the first subscript), and a certain distance from either the top or bottom (given by the second subscript). The array could be used to hold an item of information about such positions. For example, the array 'colour%' above could hold the colour of every pixel within a rectangle, 80 pixels across and 40 pixels high:

```
REM Choose the colours to use:
PALETTE 0,0,0,0
PALETTE 1,1,0,0
PALETTE 2,0,1,0
PALETTE 3,0,0,1
REM Draw 50 random circles in each colour; 1, 2 and 3
FOR col% = 1 TO 3
  FOR count% = 1 TO 50
      pos%   = INT(RND*620)
      ypos%   = INT(RND*180)
      radius% = INT(RND*60)
      CIRCLE(xpos%,ypos%),radius%,col%
  NEXT
NEXT
REM Draw a thick rectangle which contains 80x40 pixels
left.x% = 280
top.y%  =  80
LINE (left.x%,top.y%) - STEP(81,41),0,b
LINE (left.x%-1,top.y%-1) - STEP(83,43),0,b
REM Define an array to hold the colours of each pixel of
the REM rectangle
OPTION BASE 1
DIM colour%(80,40)
REM Read the colour of each point
FOR xpos% = 1 TO 80
  FOR ypos% = 1 TO 40
```

```
  colour%(xpos%, ypos%) =
POINT(left.x%+xpos%,top.y%+ypos%)
  NEXT
NEXT
REM Now reproduce the rectangle
CLS
FOR xpos% = 1 TO 80
  FOR ypos% = 1 TO 40
  PSET(left.x% + xpos%,top.y% +
ypos%),colour%(xpos%,ypos%)
  NEXT
NEXT
```

This program introduces a BASIC 'function' called POINT. A function can be thought of as a statement which returns a value. We met another example of a function in a earlier chapter which was RND. The value returned by RND is a random number in the range 0-1. In the case of POINT, the value returned is an integer in the range 0-3, giving the colour of the pixel whose x- and y- coordinates are passed as arguments. The other possible value which POINT can return is –1, this occurs if the coordinates given lie outside the current Output window.

Using three-dimensional arrays the model can be taken one stage further. The third dimension can be used to specify the height of a position relative to the plane of the piece of paper or screen. This allows information about a three-dimensional volume to be held.

The physicists among you may see the potential for treating the fourth dimension as representing time. However, beyond that, the dimensions fail to have any meaning in the real world.

Rules About Subscripts

When using arrays, remember that if you DIM an array using a particular number of subscripts, each element of the array must be referenced with the same number of subscripts:

```
OPTION BASE 1
DIM colour%(80,40)
colour%(1,1,1) = 1
```

produces the error 'Subscript out of range'. A correct version would be:

```
OPTION BASE 1
DIM colour%(80,40)
colour%(1,1) = 1
```

In addition the numbers used as subscripts must be within the correct range, ie between the lower and upper bound:

```
OPTION BASE 1
DIM colour%(80,40)
colour%(100,20) = 1
```

gives the error message because the first subscript must be between one and 80. Similarly:

```
OPTION BASE 1
DIM colour%(80,40)
colour%(1,50) = 1
```

gives an error because the second subscript lies outside the range one to 40.

Making Editing Easier

As programs get bigger, using just the arrow keys to move around them in the editor becomes a slow process. Fortunately, there are faster ways of moving through programs.

Scrolling

As we have already seen, the four arrow keys normally act as follows;

Move up by one line
Move down by one line
Move right by one character
Move left by one character

However, if the cursor is already at the edge of the display in the direction being moved, the following actions also occur:

Scroll right by three-quarters of a display
Scroll left by three-quarters of a display

To move faster in a particular direction, the arrow keys can be used in combination with the SHIFT key or ALT keys. These move the cursor to the following positions, scrolling as necessary:

SHIFT — ↑	Move backwards by one windowful
SHIFT — ↓	Move forwards by one windowful
SHIFT — →	Move right by three-quarters of a display
SHIFT — ←	Move left by three-quarters of a display
ALT — ↑	Move to the beginning of the program
ALT — ↓	Move to the end of the program
ALT — →	Move to the far right of the current line
ALT — ←	Move to the far left of the current line

Line Numbers and Labels

Any of you who have used BASIC on other machines will probably have been surprised by the absence of line numbers in the programs we have written so far. You can use line numbers, if you wish, in AmigaBASIC. However you don't have to. For example, a program such as:

```
CIRCLE(320,100),100
CIRCLE(160,50),50
CIRCLE(480,50),50
CIRCLE(160,150),50
CIRCLE(480,150),50
```

can equally well be written as:

```
10 CIRCLE(320,100),100
20 CIRCLE(160,50),50
30 CIRCLE(480,50),50
40 CIRCLE(160,150),50
50 CIRCLE(480,150),50
```

You can use any whole number between 0 and 65529 for a line number. However, it is important to note that AmigaBASIC executes

each line of a program sequentially, regardless of any line number it has. It does not execute them in numerical order of the line numbers. The numbers are there purely as markers so you can refer to particular positions in a program.

Instead of starting a line with a line number, you can start it with a 'label'. This must begin with a letter, end with a colon and contain a maximum of 39 letters, numbers or full stops in between. For example:

```
large.cir: CIRCLE(320,100),100
small.cirs:
CIRCLE(160,50).50
CIRCLE(480,50),50
CIRCLE(160,150),50
CIRCLE(480,150),50
```

It is better to use labels rather than line numbers since they can be made more descriptive and so help to document your code. The real use of line numbers or labels is to mark positions of the code which you want to 'jump' to in some way.

When listing a program, you can open the List window so that a particular line is at the top of the window. To do this, you have to select the Output window and type LIST, followed by the line number or label of the line required. For example:

```
LIST small.cirs
```

Hence it is a good idea to start sections of a program with a label, so that you can find them easily in the editor.

Keeping It Structured

We have already seen some features of AmigaBASIC which help to structure a program. FOR and WHILE loops are two examples. These allow a block of statements to be repeated several times, without the need of having multiple copies of the statements within the code. Another is the IF construct. This allows alternative pieces of code to be executed, depending on certain conditions, the code being in a very readable form. However, there are other structures which we have not yet encountered.

Subroutines

Quite often you may find that a program contains the same block of code several times at different places. This is obviously wasteful, but what can be done about it? One solution is to use a 'subroutine'. This is a block of code which starts with a line number or label, and ends with the keyword 'RETURN'. To execute this block of statements, the main body of the program just needs to issue the command 'GOSUB' followed by the relevant line number or label. Then all the statements in the subroutine will be executed until the 'RETURN' is reached, at which point BASIC will return to the statement in the main body of the program immediately after the 'GOSUB', and continue executing from there. The following diagram (Figure 4.1) should help to illustrate the process:

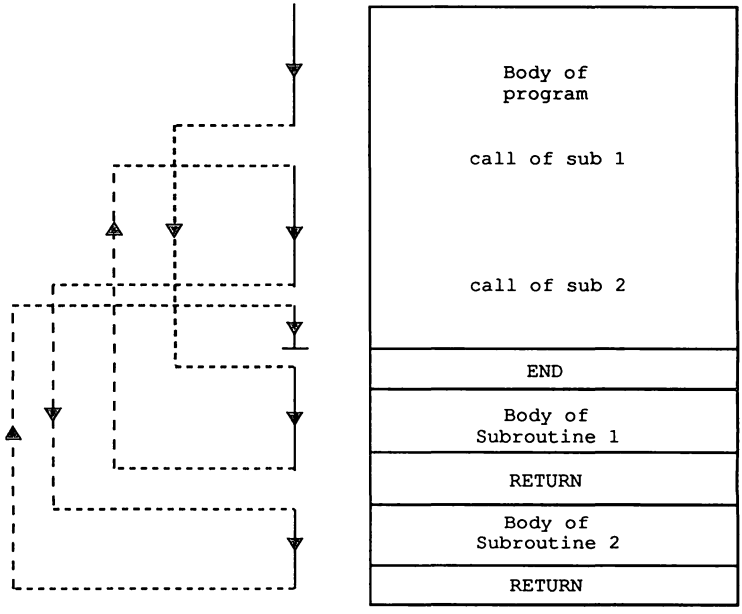


Figure 4.1. Program flow at GOSUBs.

The above demonstrates the typical structure of a program containing subroutines. The main body of the program occurs first, followed by the keyword 'END', followed by the body of the subroutines. Note that the 'END' statement is necessary to tell BASIC to stop executing

the code once the bottom of the main body of the program has been reached. Without it, BASIC would continue executing the code until it reached the end of the text. This means that it would 'fall through' into the first subroutine, and execute it as though it were part of the main program. However, when it reached the 'RETURN' statement at the end, the error message 'RETURN WITHOUT GOSUB' would be given. This is because the subroutine wasn't entered properly via a GOSUB call so BASIC doesn't know where it is to return to.

'END' may be printed at the end of all programs. However, normally the end of the main body of the program coincides with the end of the text, so the keyword is not necessary.

The following illustrates how a subroutine may be used:

```
REM Initialise the colours
PALETTE 0,0,0,0
PALETTE 1,.2,0,0
PALETTE 2,.2,.2,0
PALETTE 3,0,.2,0
REM Draw 3 solid circles vertically
xpos% = 320
FOR ypos% = 40 TO 120 STEP 40
  col% = ypos%/40
  FOR radius% = 1 TO 20
      CIRCLE (xpos%,ypos%),radius%,col%
  NEXT
NEXT
REM Now loop round the traffic light sequence
REM five times
FOR count% = 1 TO 5
  PALETTE 2,.2,.2,0 : PALETTE 1,1,0,0
  GOSUB delay
  PALETTE 2,1,1,0
  GOSUB delay
  PALETTE 1,.2,0,0 : PALETTE 2,.2,.2,0 : PALETTE 3,0,1,0
  GOSUB delay
  PALETTE 3,0,.2,0 : PALETTE 2,1,1,0
  GOSUB delay
NEXT
REM finish by displaying all three lights
PALETTE 1,1,0,0
PALETTE 2,1,1,0
PALETTE 3,0,1,0
END
REM Subroutine to pause for a while
delay:
```

```
  FOR c% = 1 TO 5000
  NEXT
RETURN
```

Subprograms

A better method of keeping code well structured is to make use of 'subprograms'. In some respects a subprogram is similar to a subroutine. Both are groups of BASIC statements which perform a particular task and which have a name or label assigned to them. However the syntax used to define and call them is slightly different. The following shows how the above program can be altered to use a subprogram for the delay code, rather than a subroutine:

```
REM Initialise the colours
PALETTE 0,0,0,0
PALETTE 1,.2,0,0
PALETTE 2,.2,.2,0
PALETTE 3,0,.2,0
REM Draw 3 solid circles vertically
xpos% = 320
FOR ypos% = 40 TO 120 STEP 40
  col% = ypos%/40
  FOR radius% = 1 TO 20
      CIRCLE (xpos%,ypos%),radius%,col%
  NEXT
NEXT
REM Now loop round the traffic light sequence
REM five times
FOR count% = 1 TO 5
  PALETTE 2,.2,.2,0 : PALETTE 1,1,0,0
  CALL delay
  PALETTE 2,1,1,0
  CALL delay
  PALETTE 1,.2,0,0 : PALETTE 2,.2,.2,0 : PALETTE 3,0,1,0
  CALL delay
  PALETTE 3,0,.2,0 : PALETTE 2,1,1,0
  CALL delay
NEXT
REM Finish by displaying all three lights
PALETTE 1,1,0,0
PALETTE 2,1,1,0
PALETTE 3,0,1,0
REM Subprogram to pause for a while
SUB delay STATIC
  FOR count% = 1 TO 5000
  NEXT
END SUB
```

The body of the subprogram is started by the statement 'SUB' and ended by the statement 'END SUB'. 'SUB' must be followed by the name of the subprogram which may be up to 40 characters long and then the keyword 'STATIC'. The code of the subprogram can be called from anywhere in the main body of the program by using the keyword 'CALL', followed by the name of the subprogram.

The program illustrates two interesting facts about subprograms which do not apply to subroutines. The first is that the code of the subprogram is only executed when it is CALLed. At the end of the traffic light loop, the program will execute the final three PALETTE statements and then stop. The flow of control will not fall through into the body of the subprogram. Therefore an 'END' statement is not necessary.

The second is that by default, the variables used in the subprogram are entirely independent from those of the main program. Assigning a value to a variable within the subprogram does not affect the value of any variable in the main program which has the same name. Therefore the variable 'count%' can be used within the subprogram without it affecting the loop variable, also called 'count%', of the loop from which the subprogram is called.

Passing Parameters

What makes subprograms so useful is that fact that they can take 'parameters'. These are variables which are passed values from the main program. In the example above, the length of time each sequence is displayed for is equal. In real life, the red and green stages would last far longer than the intermediate, amber, stages. One way of achieving this would be to have two different subprograms: one which counted to 5000, and the other which counted to, say, 20000. A better way is to alter the current subprogram as follows:

```
SUB delay(limit%) STATIC
  FOR count% = 1 TO limit%
  NEXT
END SUB
```

This now has one parameter which determines the value counted to and hence the length of the delay. The main program also has to be altered so that each time delay is called, a value is passed to it. For example calls to delay have to take the following kind of format:

```
CALL delay(5000)
```

or alternatively:

```
period% = 20000 : CALL delay(period%)
```

This means that the main program can influence the action of the subprogram. , in this example only one subprogram need be used, even though different length delays are required.

Updating Parameters

By passing a constant value, such as 5000, the main program is sending information to the subprogram. However, when the main program passes a variable, the subprogram has the opportunity to send information back. For example, what would you expect the following to print?

```
a% = 1
b% = 1
CALL double(a%)
PRINT a%
PRINT b%
SUB double(num%) STATIC
  num% = num% * 2
  a% = 3
  b% = 3
END SUB
```

The values actually printed are:

```
2
1
```

The subprogram has its own local copies of the variables a% and b%, so the assignments to them do not alter the a% and b% in the main program. However, it is being passed a% as a parameter. The value of

the parameter is doubled by the subprogram, and hence the value of a% is doubled. If you don't want the values of the main program's variables to be altered by calling a subprogram, then you can enclose them in brackets. For example:

```
a% = 1
b% = 1
CALL double((a%))
PRINT a%
PRINT b%
SUB double(num%) STATIC
  num% = num% * 2
  a% = 3
  b% = 3
END SUB
```

will print:

```
1
1
```

The situation is summarised in the table below:

Argument passed	Effect on variable
variable	updated
(variable)	unaltered

It is a very important point to remember. Forgetting to use brackets, when you meant to, can lead to all sorts of problems and it is very difficult to spot this type of mistake when reading through a program.

Local and Shared Variables

An alternative way of referring to and/or updating the main program's variables inside a subprogram, is to use the SHARED statement. For example:

```
a% = 1
b% = 1
CALL treble
PRINT a%
PRINT b%
SUB treble STATIC
SHARED a%,b%
```

```
  a% = 3
  b% = 3
END SUB
```

gives the results:

```
3
3
```

The shared statement makes the variables inside the subprogram refer to the same variables as the main program.

Any variables which are not parameters and are not SHARED are local to the subprogram. The values of local variables are preserved between calls to a subprogram. For example:

```
FOR loop% = 1 TO 5
  CALL count
NEXT
SUB count STATIC
a% = a% + 1
PRINT a%
END SUB
```

will print:

```
1
2
3
4
5
```

When the subprogram is first entered, its local variable a% will be initialised to zero. The body of the subprogram increases a% by one and prints its value. The next time count is entered, a% still has the same value, so this time adding one gives the result two, etc.

Subprograms: Other Points to Note

There are a few more points which need to be made here about subprograms. The first is that you cannot nest them, ie one subprogram cannot contain the definition of second subprogram. However a subprogram can call other subprograms.

The second is that a program cannot contain two subprograms with the same name. An error message will be given if this occurs.

Finally, you can return control, before the end of the subprogram is reached, by using the 'EXIT SUB' command. For example, the subprogram, used earlier to generate a pause, could be improved as follows:

```
REM Subprogram to pause for a while
SUB delay(limit%) STATIC
  IF limit% < 1 THEN EXIT SUB
  FOR count% = 1 TO limit%
  NEXT
END SUB
```

This then checks that a valid length of time has been passed as a parameter and, if not, it does nothing. Otherwise it pauses as before.

Functions

The last structure which we will look at here, very briefly, is the 'function'. We've already met some BASIC functions; RND and POINT. However, BASIC also allows you to create your own. The following illustrates their syntax:

```
DEF FNadd(val1,val2) = val1 + val2
```

The definition starts with 'DEF FN' followed by the name to be assigned to the function. This must be adjacent to the DEF FN, with no spaces in between. Then any parameters to be passed are given, enclosed in round brackets and separated by commas. Finally the definition contains an equals sign and an expression which specifies the value which the function is to return.

User defined functions can be used in expressions. For example:

```
PRINT FNadd(a,b)
```

or alternatively:

```
sum = FNadd(a,b) + FNadd(c,d)
```

or:

```
sum = FNadd(FNadd(a,b),FNadd(c,d))
```

etc.

There are a few rules about the use of functions which are listed below:

- The function definition must be executed before the function can be called. Otherwise a 'Undefined user function' error will be generated.
- A program can contain more than one definition of a function, in which case the most recently executed definition will be used.
- Functions can return either numeric or string values. However, if the result of the expression in the function definition does not match the type of variable being assigned to when the function is called, a 'Type mismatch' error will occur.
- Function definitions cannot occur within subprograms.

Because their definitions can only contain a single, one-line expression, functions are of limited use. They have been included here mainly for reference should you encounter any in other programs.

Merging Programs Together

At some stage, as you write more and more programs, you will probably find that you would like to include the whole or part of one program within a second. An easy way of combining two programs is provided by the MERGE command which takes one program and adds it to the end of the current program. The only complication is that the program being merged has to be stored in a particular format.

Normally, when a BASIC program is saved, it is stored on disc in a special, compact form. Each of the keywords is represented as a single 'token' or number, rather than being held as the characters making up its name. For example, PRINT is stored as the single number 172 rather than the five numbers which represent the letters 'P', 'R', 'I', 'N' and 'T'.

However, in order to be merged a program must look like ordinary untokenised text. In order to store a program in this format, the 'a' option must be used when saving it. For example:

```
SAVE "prog2",a
```

Thus the sequence to merge one program 'prog2' onto the end of another program 'prog1' is as follows:

1) Create 'prog1'
2) Save it as a BASIC program (SAVE "prog1")
3) Create 'prog2'
4) Save it as a text file (SAVE "prog2",a)
5) Load the first program into memory (LOAD "prog1")
6) Merge the second program onto the end (MERGE "prog2")

What To Do With Data

All programs require data. Even the first program we wrote, which drew a single circle on the screen, required three pieces of information: the x- and y-coordinates of the centre of the circle and its radius. So far, all the data we have used has been mixed in with the code. For example, if we've wanted to draw five circles at specific positions, we've written the following type of program:

```
CIRCLE 320,100,50
CIRCLE 160,50,25
CIRCLE 480,150,25
CIRCLE 160,150,25
CIRCLE 480,50,25
```

However, it is better, for programs which are going to handle large amounts of data, to keep the program and the data separate.

Reading and Defining Data

BASIC provides a pair of very useful keywords for dealing with data. These are DATA and READ. The DATA statement is used to store items of data within a program. The READ statement is used to access these items. For example, if you wish to draw a series of circles at fixed points on the screen, you can do so as follows:

```
FOR count% = 1 TO 5
  READ xpos%,ypos%,radius%
  CIRCLE (xpos%,ypos%),radius%
NEXT count%
DATA 320,100,50
DATA 160,50,25
DATA 480,150,25
DATA 160,150,25
DATA 480,50,25
```

This program produces exactly the same result as the previous program.

When the program is run, the READ statement looks through the program until it finds the first DATA statement. It then takes the first item of data (the number 320) and assigns this to the variable xpos%. Similarly it assigns the second item of data to ypos% and the third to radius%. The next time round the loop the READ statement carries on reading data from where it left off. This time it reads the numbers 160, 50 and 25 into xpos%,ypos% and radius%. This happens five times, once each time round the loop. Therefore each set of three numbers is read in turn.

The DATA statements can be followed by one or more items of data separated by commas. In the above example the data was split, so that the three items of data, which were assigned by each READ, were together on one line. This was only to make it clear what each value was to be used for, either the x-coordinate, y-coordinate or radius. The following would work just as well:

```
FOR count% = 1 TO 5
  READ xpos%,ypos%,radius%
  CIRCLE (xpos%,ypos%),radius%
NEXT count%
DATA 320,100,50,160,50
DATA 25,480,150,25,160,150,25,480,50,25
```

DATA statements can contain a mixture of numbers and strings. You must make sure, though, that the type of each item of data matches the type of the variable it is being read into. For example:

```
FOR count% = 1 TO 5
  READ name1$, age%
```

```
  PRINT "My name is ";name1$;" and I am ";age%;" years
old"
NEXT
DATA "Tom", 4,"Dick",5,"Harry",6
DATA "Jack",3,"Jill",3
```

Normally you can leave out the quotation marks around strings. In the above program, for example, they are not necessary. However, they are needed if you want to include commas in the string or if you want the string to start or end with spaces:

```
READ A$,B$
PRINT A$
PRINT B$
DATA How are you?
DATA Well, I hope.
```

produces:

```
How are you?
Well
```

This is because the second READ reads characters until it comes across the comma after the 'Well' and concludes that this ends the item of data.

To obtain both sentences in full, change the program to be as follows:

```
READ A$,B$
PRINT A$,B$
DATA How are you?
DATA "Well, I hope"
```

The DATA statements may occur anywhere in the program but it is best to keep them on a line of their own. If BASIC reaches a DATA statement when it is executing a program, it ignores it and goes on to the next line. It only uses the DATA statements when it encounters a READ.

When it attempts to READ the first item of data, it scans through the lines of the program from the top until it finds the first DATA statement and uses the first item of data on this line. The next READ

uses the second item, and so on until the DATA statement has no more items of data left, in which case the next DATA statement is searched for and used.

If there is too much data then the extra items are just left unread. However, if there is insufficient data, BASIC produces the error message:

```
Out of data
```

This indicates that it has tried to READ an item of data but found that there was none left unread.

Re-using Data Statements

The keyword, RESTORE, may be used to set the data-pointer to the start of a DATA statement, or any line above it. This allows the data statements to be used in a different order from how they occur in the program. They can even be used more than once, for example:

```
FOR count% = 1 TO 5
  RESTORE
  number% = INT(RND*3.99) + 1
  FOR count2% = 1 TO number%
       READ number$
  NEXT count2%
  PRINT number$
NEXT count%


DATA one,two,three,four
```

Each time round the main loop the RESTORE statement sets the data-pointer to point to the first item of data. A number in the range one to four is generated at random, and this number of items of data are read. The final item read is printed. So if the random number was three, three items of data would be read and the final one, the string 'three', would be printed.

You can restore to a specific line of data by using a label or line number. For example, the following produces the same results as the one above, but in a different manner:

```
FOR count% = 1 TO 5
  number% = INT(RND*3.99) + 1
  IF number% = 1 THEN
       RESTORE 1
  ELSEIF number% = 2 THEN
       RESTORE 2
  ELSEIF number% = 3 THEN
       RESTORE 3
  ELSE
       RESTORE 4
  END IF
  READ number$
  PRINT number$
NEXT count%
1 DATA one
2 DATA two
3 DATA three
4 DATA four
```

This time, the RESTORE is followed by a number. The data-pointer is set to the first DATA statement on or after the line of this number. Then the next item of data is read and printed. So if the random number was three, the data-pointer would be set to point at the DATA statement on line three and the data on this line, the string 'three', would be printed.

Error Handling

In this section we are going to distinguish between two different types of errors which cause the program to stop. The first type are syntax errors in the program itself. For example, mistyped keywords and labels, forgotten END statements, unmatched FOR and NEXT statements etc. These are easy to find since they will be reported the first time the code, in which they occur, is executed.

The second type are errors due to interactions between the program and the data it is acting on. In this case the program may work perfectly well in some cases, but fail in others. For example, consider the following program:

```
INPUT "Please give me a number";num
PRINT "The reciprocal of ";num;" is ";1/num
```

This program works fine for most numbers, but cannot cope with the value zero. This causes the error message 'DIVISION BY ZERO' to be reported.

This error is not due to the program being wrong, it is caused by the program being given data that it cannot handle. Therefore it could be classed as being a 'user fault' rather than a mistake by the programmer. However, it is up to the programmer to anticipate all the different responses that the user can give and prepare the program to handle them.

One method of coping with errors is to set up 'error handlers'. These are routines which are called whenever an error occurs. For example:

```
ON ERROR GOTO errorhand
FOR loop% = 1 TO 5
  INPUT "Please give me a number :",num
  PRINT "The reciprocal of ";num;" is ";1/num
NEXT
END
errorhand:
IF ERR = 11 THEN
  PRINT "infinite"
ELSE
  ON ERROR GOTO 0
END IF
RESUME NEXT
```

The first statement of the program sets up an error handler which starts at the label 'errorhand:'. Unless an error occurs, the code starting at this point is not executed. However, if an error does occur, BASIC will immediately jump to this label and execute the code in the error handler.

In this example, the error handler uses ERR to find out which particular error has occurred. This function returns the number associated with the last error, so allowing the particular error to be identified. A full list of errors and error numbers are given in appendix B.

The error, which the program is interested in trapping, is 'Division by zero' whose error number is 11. If this is the value returned, then the error handler prints out the string 'infinite' and then executes the

'RESUME NEXT' statement, which instructs BASIC to continue executing the program at the statement following the one on which the error occurred. If any other error has been generated, then it executes an 'ON ERROR GOTO 0'. This turns error trapping off and so allows BASIC to print out the error in the usual way.

The 'RESUME' statement is used to signify the end of the error handler, and to instruct BASIC to continue executing the program. It can take one of four forms:

Statement	Execution resumes at
RESUME	The statement which caused the error
RESUME 0	The statement which caused the error
RESUME NEXT	The statement after the one which caused the error
RESUME <line>	The statement at the line number or label given

Note that RESUME statements can only be used within error handlers.

Different error handlers can be used at different points in the code. Whenever an error occurs, the most recently executed ON ERROR GOTO statement is used to determine the address of the error handler to use.

Debugging

Sooner or later, everybody writes a program which is syntactically correct but which doesn't do what they intended it to when it is run. AmigaBASIC provides some good facilities to help track down the mistakes when this occurs.

Stepping Through a Program

Instead of running a program as normal, you can 'step' through it. To do this, start executing the program by selecting Step from the Run menu or pressing 'Amiga-T'. This will execute the first statement and then wait. Selecting Step, or pressing 'Amiga-T' again, executes the next one and so on.

If you bring the List window to the front at any stage, this will contain your program with an orange rectangle surrounding the statement which has just been executed. This allows you see the path taken through the code.

Stepping through a program is a never ending activity. When the end of your program is reached, the next step takes you back to the beginning and the whole process starts again. To break out of this vicious circle, select the Continue option from the Run Menu. This will execute the rest of the program as normal and stop when it reaches the end.

If you have a large program which is causing problems, you will often have a good idea roughly where the error is occurring. Unless the error is thought to be close to the beginning, stepping through right from the start of the program is a time-wasting activity. To avoid doing this, you can place a STOP statement in the program just before the area of code suspected of being wrong. This will stop BASIC executing your program at that position. You can then use the Step option to move through the subsequent statements one at a time.

Examining and Resetting Variables

While a program is temporarily suspended, you can find out the value of a variable, for example col%, by selecting the Output window and typing:

```
PRINT col%
```

This helps to isolate any positions in a program where a variable is being assigned the wrong value.

In addition, you can alter a value using the LET statement, for example:

```
LET col% = 1
```

This is particularly useful if you think you have found out where a program is going wrong. It lets you set the variables to what you think are the correct values and continue execution to see if that fixes the problem.

Applying These Techniques

To practise these techniques, see if you can find the two mistakes in the following program, by stepping through it and examining the values of its variables at various stages:

```
.palette
PALETTE 0,0,0,0
PALETTE 1,.4,.4,.4
PALETTE 2,.7,.7,.7
PALETTE 3,1,1,1
CLS


.start
OPTION BASE 1
DIM sales%(12)


input_figures
max% = 0
FOR month% = 1 TO 12
  INPUT,"Monthly sales figure :";sales%(month%)
  IF sales%(month%) > max% THEN max% = sales%(month%)
NEXT


.axes
xoffset% =  20
yoffset% = 160
width%   = 600
height%  = 100
LINE (xoffset%,yoffset%) - STEP(width%,0),1
LINE (xoffset%,yoffset%) - STEP(  0,-height%),1


.max_height
pixels.per.unit = max% / height%
.bar_chart
col% = 1
FOR month% = 1 TO 12
CALL bar(month%)
NEXT


.bar
SUB bar(month%) STATIC
SHARED xoffset%, yoffset%, width%, height%, sales%, col%
month% = month% - 1
```

```
x1% = month%*xoffset%
x2% = month%*xoffset% + width%
y1% = yoffset%
y2% = yoffset% - INT(pixels.per.unit*sales%(height%))
col% = 3 - col%
LINE (x1%,y1%) - (x2%,y2%),col%,bf
END SUB
```

Currently, the program goes into a loop. To stop it, select Stop from the Run Menu or press 'Amiga-.' (Amiga and the full stop key).

What the program is meant to do is to input 12 values, representing the sales figures for 12 months, and to display a bar chart of the results. When correct, its description is as follows:

The program loops round reading in the values. These are assumed to be positive integers, but no check is made to ensure this. As each value is entered, it is placed in the array sales% and the maximum value so far is updated. When all twelve have been entered, it draws axes on the screen, 600 pixels wide and 100 pixels high, and calculates the number of pixels which represent each unit of sales to ensure that the highest bar reaches the top of the axes.

Then it loops round again and for each month calls a subprogram to draw the appropriate bar. The height of each bar is given by the sales for the month multiplied by the number of pixels per unit, calculated as described above. The bottom corner for each bar is a distance (month% – 1)*width% away from the corner of the axes. Hence month one starts 0 pixels from the left, month two width% pixels away, month three 2*width% pixels away and so on.

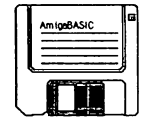
The bars are plotted in alternating shades of grey, the colour numbers used being two for the first, one for the second, two for the next etc.

The mistakes are given below.

'pixels.per.unit' should be SHARED in the subprogram – currently it is assumed to be local and so is initialised to zero.

'month%' should be enclosed in brackets when it is passed to the subprogram, so that assignments to the parameter within the subprogram do not affect the variable in the main program. Currently the parameter is being decreased by one in the subprogram and so

month% is decreased from one to 0 inside the loop. The NEXT statement increases it back to one again which means that the first bar is plotted repeatedly and the loop never terminates.



5 : Manipulating Text

We have seen that BASIC can handle strings of characters but so far we have made very little use of them. Strings input by the user have been treated as 'indivisible' items – no attempt has been made to analyse the words and letters which they contain. In addition, any text printed on the screen has been printed where BASIC decided to put it – we have not put much effort into controlling its position.

One of the hardest parts of writing a program is designing the interface between the computer and the user. This problem is in two parts. Firstly, we have to try to act intelligently with the text which the user provides. Then we have to output our responses in an attractive manner. Little things like printing titles off-centre, losing parts of sentences off the edge of the screen, or splitting words between two lines can ruin the image of a program.

Good demonstrations of both of these are provided by 'adventure games'. These are games which allow the player to explore a fantasy world. This world usually consists of lots of different rooms and locations containing treasures to collect, puzzles to solve and monsters to defeat. The player moves about by giving instructions such as 'Go north' and 'Enter hut', and the computer acts as the player's eyes and ears, describing the location he is in and any interesting objects which there are nearby. These objects can be collected and used by giving commands such as 'Get diamonds' and 'Light lamp'. More sophisticated programs can understand a more complicated syntax such as 'Throw the axe at the giant'. These programs have to be able to extract the individual words of interest from a given string. These are then checked against lists of known verbs and nouns and acted on if understood.

A more serious example is a database. Its function is to allow information to be stored and retrieved quickly and easily. To retrieve information the user has to give it a 'pattern' to use and it will then search through all the records it has and list the ones which match this pattern. For example, if names and addresses are being stored, a

simple pattern would be the name 'Joe Brown'. It will then give the addresses of all the people it knows with that name. To be of any real use, the database will have to allow 'wildcarded patterns'. These contain symbols which can match certain classes of letters. One commonly used one is '*' which matches any number of any letters. For example '* Brown' will give you the address of anyone whose surname is Brown, whatever their first name. Thus a database must be capable of searching strings to see if they contain a particular sequence.

String Expressions

Some operators, such as the logical operators, obviously only make sense when acting on integer operands. However, others can be used on string variables as well.

Comparing Strings

All the comparison operators can be used on string variables. The '=' and '<>' operators test whether two strings are identical or not. The other operators need to be looked at more closely to see how they work.

Every character is represented within the computer by a number in the range zero to 255. The system used to determine the value associated with a particular character is known as ASCII. This stands for 'American Standard Code for Information Interchange'. Virtually all computers use this system, which means that information can be exchanged between them easily.

BASIC provides a pair of functions for converting characters to their ASCII number-codes and back again. These are ASC and CHR$. For example:

```
FOR character% = ASC("A") TO ASC("Z")
  PRINT CHR$(character%);
  PRINT character%
NEXT
FOR character% = ASC("a") TO ASC("z")
  PRINT CHR$(character%);
  PRINT character%
NEXT
```

This program loops through all the letter A to Z and then a to z and for each prints out its ASCII value.

It is these ASCII values which are used when comparing strings. For example:

```
"BILL" < "FRED"
```

gives the result 'true' since the letter 'B' has an ASCII value which is less than that for the letter 'F'. Similarly

```
"BILL" < "BOB"
```

is also 'true'. In this case the first letters are the same so the next two are compared and the ASCII value for 'I' is less that the ASCII value for 'O'. In addition

```
"BILL" < "BILLY"
```

is 'true'. No character at all is less than 'Y'. Finally

```
"BILL" < "bill"
```

is 'true'. All upper-case characters have ASCII values which are less than the lower-case characters. The previous program illustrates this.

You need to be aware that upper- and lower-case characters are represented differently, particularly when you are dealing with input from a user. For example, the simple request:

```
INPUT "Do you wish to continue (Y/N)"; ans$
```

would probably be complained about if it only accepted 'Y' or 'N' and ignored 'y' and 'n'.

The problem of recognising what the user has typed becomes harder if longer strings are permitted. For example there are four different variations of NO : 'NO', 'No', 'no' and 'nO' and eight of YES : 'YES', 'Yes', 'yes', 'YEs', 'yeS', 'yES', 'yEs' and 'YeS'.

Rather than testing a string which has been input against all the possible permutations of upper- and lower-case letters, it is easier to

convert it to upper-case and then just do one test. To do this, the function UCASE$ can be used. This takes a string as an argument and returns it with all the characters converted into upper-case. For example:

```
answer$ = ""
WHILE answer$ <> "YES"
  INPUT "Do you wish to continue (YES/NO)"; ans$
  answer$ = UCASE$(ans$)
  IF answer$ = "NO" THEN END
WEND
REM rest of program
.....................
```

This extract from a program starts by asking the user to enter either 'YES' or 'NO'. The string typed is converted to upper-case. Then, if it matches the string 'NO', the program ends. If it matches the string 'YES' then the program will move onto the statement after the WEND. Otherwise, the loop will be repeated and the question asked again.

Joining Strings Together

The '+' operator is used to 'join together', or more correctly speaking 'concatenate', two strings. For example:

```
name1$ = "Winston"
surname$ = "Churchill"
PRINT name1$ + " " + surname$
```

Running this program produces:

```
Winston Churchill
```

Note, however, that the other arithmetic operators are meaningless when applied to strings and produce an error message.

To obtain a string containing multiple copies of a character, the STRING$ function can be used. This takes two arguments. The first is an integer specifying the length of the string to be returned. The second is either the ASCII code of the character wanted, or a string starting with the character wanted. For example, the following three programs produce the same output:

```
REM Prog1
FOR loop% = 1 TO 10
  PRINT STRING$(loop%,ASC("*"))
NEXT


REM Prog2
FOR loop% = 1 TO 10
  PRINT STRING$(loop%,"*")
NEXT


REM Prog3
FOR loop% = 1 TO 10
  PRINT STRING$(loop%,"*/+-")
NEXT
```

If the character to be repeated is a space, then an alternative method is to use SPACE$. This takes just one argument, which is the number of spaces required.

Converting Between Numbers and Strings

It has been emphasised, throughout this book, that strings and numbers are fundamentally different. You cannot assign a number to a string variable and vice versa. However, at times this may cause a problem. For example you may have a string and want to treat its characters as a number. AmigaBASIC provides routines to handle this kind of situation.

The function VAL takes a string of digits and converts it into a number. For example:

```
string1$ = "42"
string2$ = "37"
num1% = VAL(string1$)
num2% = VAL(string2$)
PRINT num1% + num2%
```

will output the value 79.

VAL ignores any space or tab characters at the start of the string and returns the value of string, up to the first character which cannot be treated as part of a number. For example:

```
PRINT VAL ("  12 High Street")
```

prints the value 12. Remember that it is not only digits which can be treated as valid components of a number. For example:

```
PRINT VAL("1E2")
```

prints the value 100. Although the 'E' is a not a digit, 1E2 is a valid way of representing a number using exponential format.

The string may begin with a '+' or '-', for example:

```
number% =  VAL ("-8")
```

assigns the value –8 to 'number%'.

If, however, the characters of the string (ignoring spaces) do not start with a digit or a plus or minus sign, then VAL returns 0.

There are three functions for converting a number into a string. The most commonly used is STR$. This takes a decimal number as its argument and returns the string containing the digits of the number. For example:

```
num1% = 42
num2% = 37
string1$ = STR$(num1%)
string2$ = STR$(num2%)
PRINT string1$ + string2$
```

produces the string ' 42 37'.

Note that the strings start with a space. All positive numbers are converted to strings with a leading space whereas negative ones have a leading minus sign.

The other two functions are HEX$ and OCT$. These also take a decimal number as their argument. However, the strings they produce

represent the hexadecimal (base 16), and octal (base eight), values of this number respectively.

Octal numbers contain eight digits zero to seven. A one in a particular column represents a power of eight, ie:

... 64 8 1

With hexadecimal numbers there is a slight problem. They require 16 digits to represent the decimal values 0 to 15. We can use the digits 0 to 9 as usual, but have to use the letters 'A' to 'F' to represent the values ten to fifteen. Therefore 4AC is a valid hexadecimal number. Its decimal equivalent is 4*256 + 10*16 + 12, ie 1196. Therefore:

```
PRINT HEX$(1196)
```

will print the string '4AC'.

Finding the Length of a String

Given a string, the first thing you may wish to know about it is its length. This can be determined as follows:

```
INPUT "Please give me a string :",A$
PRINT LEN(A$)
```

LEN returns the number of characters in a string including spaces, tab characters etc. This will be an integer value between 0 (for the null string) and 255 (which is the maximum number of characters allowed in a string).

Finding Strings within Strings

AmigaBASIC provides a function, INSTR, to check if one string occurs within another. This function returns a number giving the position, within the longer string, at which it found the start of the shorter one. If the longer string does not contain the shorter one at all, then INSTR returns 0. For example:

```
INPUT "Please type an upper-case letter :",letter$
IF LEN(letter$) < 1 THEN
  PRINT "You gave me a null string"
ELSEIF LEN(letter$) > 1 THEN
  PRINT "You typed too many characters"
```

```
ELSE
  pos.in.string% =
INSTR("ABCDEFGHIJKLMNOPQRSTUVWXYZ",letter$)
  IF pos.in.string% <> 0 THEN
   PRINT letter$ " is at position "
pos.in.string%,PRINT" in the alphabet."
  ELSE
    PRINT "The character was not an upper-case letter"
  END IF
END IF
```

This program inputs a string from the user and checks that it contains a single character. If so, it uses INSTR to find if this string occurs within the sequence 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. If it does then INSTR returns the position at which it found the string, ie 'A' gives the result one, 'B' gives two and so on. If the string was not found, for example if the user typed a lower-case letter or a digit, then INSTR returns 0 and the program tells the user that they didn't type an upper-case letter.

You can also tell INSTR the position in the longer string at which you wish it to start looking for the shorter one. For example:

```
PRINT INSTR(1,"The cat sat on the mat.","at")
```

will start the search for 'at' at position one, ie the start of the sentence. It will therefore give the result six since it will find the 'at' of the word 'cat'. This is the same as:

```
PRINT INSTR("The cat sat on the mat.","at")
```

since the position defaults to one if it is not given explicitly.

In contrast:

```
PRINT INSTR(11,"The cat sat on the mat.","at")
```

will start the search at the the 11th letter of the sentence, ie the 't' of 'sat'. This will give the result 21 since it will find the 'at' of the word 'mat'.

Using this feature, you can find all occurrences of one string inside another, for example:

```
res% = -1
WHILE res% <> 0
  res% = INSTR(res%+2,"The cat sat on the mat.","at")
  IF res% <> 0 THEN PRINT res%
WEND
PRINT "All found"
```

This program starts searching at position one in the sentence. Each time it finds an occurrence of 'at', it prints out the position and starts searching again, starting at the next character of the sentence. The following illustrates what is happening. The '*' symbol denotes the position at which each search starts, and the '%' symbol shows the positions of the 'at' strings which are found.

```
The cat sat on the mat.
*    % * % *        % *
```

Splitting Strings

BASIC provides three functions for taking a large string and extracting a smaller one from within it. The simplest are LEFT$ and RIGHT$. These return a string made up of a particular number of characters from the left-hand or right-hand end of another string. For example:

```
PRINT LEFT$("HELLO",2)
```

will print:

```
HE
```

ie the first two characters of the string 'HELLO'. Similarly

```
PRINT RIGHT$("HELLO",2)
```

will print:

```
LO
```

which are the last two characters of the string.

The third function is MID$. This returns a number of characters starting from a given position within the string. For example:

```
PRINT MID$("HELLO",2,3)
```

will print:

```
ELL
```

That is three characters from the string 'HELLO' starting from the second character.

You can use MID$ to extract each character in turn from a string as follows:

```
INPUT "Enter a string :",string1$
FOR start% = 1 TO LEN(string1$)
  PRINT MID$(string1$,start%,1)
NEXT start%
```

MID$ is being used to find a single character at a time. The position of this character within the string is determined by 'start%', which starts at the value one and increases by one each time round the loop. The number of characters printed is dictated by the length of the string.

MID$ is particularly useful when used in conjunction with INSTR, which we met above. For example:

```
INPUT "Enter a sentence",string1$
start% = 1
space% = INSTR(string1$," ")
WHILE space% <> 0
  num% = space% - start%
  PRINT MID$(string1$,start%,num%)
  start% = space%+1
  space% = INSTR(start%,string1$," ")
WEND
PRINT MID$(string1$,start%,LEN(string1$)+1)
```

Try running this program and typing in the sentence:

```
The quick brown fox jumps over the lazy dog
```

You should find that each word is printed out on a separate line as follows:

```
The
quick
brown
fox
jumps
over
the
lazy
dog
```

What the program is doing is extracting each word of the sentence in turn and printing it out. It begins by initialising the value of 'start%' to one, ie the position of the first character of the sentence. Next, INSTR is used to find the position of the first space in the sentence, and this value is assigned to 'space%'.

Then the loop starts. The first line within the loop sets 'num%' to be the difference between 'space%' and 'start%'. This gives the number of characters in the first word. Using MID$, this number of characters is extracted from the sentence starting at position 'start%', and is printed out. Then the program prepares to find the next word. 'start%' is updated to the value 'space% + 1'. Since 'space%' contains the position of the first space, 'space% + 1' will give the position of the start of the second word. (This is making the assumption that words are separated by just a single space – the program won't work properly if this is not the case.) INSTR is used again to find the position of a space within the sentence. This time the sentence is searched from the start of the second word, rather than from the beginning, so that the second space is found. Then the process is repeated.

This continues until no more spaces are found. At this stage, start% holds the position of the start of the final word which has not yet been printed. Therefore the final PRINT is needed to complete the process and print the remaining characters of the string.

Replacing Part of a String

MID$ can also be used to replace part of a string. For example:

```
a$ = "AAAAAAAA"
MID$(a$,3,2) = "BCDE"
PRINT a$
```

This program takes the variable 'a$' which contains the string 'AAAAAAAA' and replaces two of the characters in it, starting at position three, with the first two characters of the string 'BCDE'. The result is the string 'AABCAAAA'.

If the final argument is omitted, then the whole of the replacing string is used:

```
a$ = "AAAAAAAA"
MID$(a$,3) = "BCDE"
PRINT a$
```

This gives the result 'AABCDEAA'.

Note that the length of the string being replaced is never altered:

```
a$ = "AAAAAAAA"
MID$(a$,3) = "BCDEFGHI"
PRINT a$
```

This gives the result 'AABCDEFG'. MID$ starts at position three and carries on replacing the characters in 'a$' with those in the right-hand string, until it runs out of characters in 'a$' to replace.

Altering the Width of a Line

What does BASIC do when it is asked to print a line which is wider than the width of the screen? To find out, try running the following program:

```
test$ = "The quick brown fox jumps over the lazy dog"
PRINT "The phrase '" test$ "' contains all the letters
of the alphabet."
```

You should find that the last few words of the sentence disappear off the right-hand side of the screen. AmigaBASIC has the concept of a 'line width'. This is a limit on the number of characters which can be printed on a line. If this number is reached, then BASIC inserts a carriage return and continues printing on the next screen line. The default line width is infinite, ie BASIC will never insert a carriage

return for you however many characters you try to print. To change this, use the WIDTH statement, for example:

```
WIDTH 62
```

sets the line width to 62 characters. This is a good number to choose when using the default font, since it is the maximum number of characters which can be fitted onto one line. Giving WIDTH an argument between one and 254 sets the line width to a particular number of characters, the value 255 is the default, which is taken as meaning infinite.

To see the effect of WIDTH try the following program:

```
a$ = "aaaaaaaaaa"
b$ = "bbbbbbbbbb"
FOR chars% = 20 TO 80 STEP 20
  WIDTH chars%
  PRINT a$;b$;a$;b$;a$;b$;a$;b$
  PRINT
NEXT
```

Character Positions

Whenever you print a character, it appears at the current 'pen position'. Just like a graphics coordinate, this has a position which is measured relative to the left-hand side and top of the screen. However, instead of being measured in pixels, it is measured in terms of lines and columns. A character which is in the top left-hand corner is in column one on line one. Moving down a row increases the line number and moving right by one character increases the column number.

With the default font, all the characters are equal. Therefore if you start at the left-hand side of a line and print 50 characters, they will appear in columns 1–50 exactly. However, this is not always the case. Some fonts are 'proportionally spaced', which means that different characters have different widths. For example 'm' and 'w' are wider than 'i' and 'l'. In this case, what do we mean by the width of a character? It is actually determined by the character '0' of the font being used. So if you print the number '0' fifty times, the last one is

defined to be in column 50. Whereas the fiftieth letter 'm' will probably be in a higher numbered column and the fiftieth letter 'i' in a lower numbered one.

For the sake of clarity, the examples in this chapter assume that the default font is being used.

Tabulating Output

Another use of WIDTH is to alter the 'zone widths'. For example:

```
WIDTH 62,10
```

sets the line width to 62 and the zone width to 10 columns. Therefore, if you use a comma to separate items to be printed, these will appear either in column 1, 11, 21, 31, 41, 51 or 61. For example, try the following:

```
max.string$ = "1234567890"
FOR loop% = 1 TO 10
  min.str$ = LEFT$(max.string$,loop%)
  WIDTH 62,loop%
 PRINT
min.str$;min.str$;min.str$;min.str$;min.str$;min.str$:
  PRINT "-","-","-","-","-","-"
NEXT
```

Each time round the loop, the loop variable is used to determine the zone width and the number of characters of the sequence '1234567890' which are placed in 'max.string$'. Then six copies of 'max.string$ are printed consecutively, followed on the line beneath by six '–' characters, each at the start of a zone. Since the zone width is equal to the number of characters in 'max.string$', this has the effect of underlining the start of each copy of 'max.string$'. Therefore the '–' always lies beneath the number 1 as shown below:

```
111111
------
121212121212
- - - - - -
123123123123123123
-  -  -  -  -  -
123412341234123412341234
-   -   -   -   -   -
123451234512345123451234512345
```

```
-    -    -    -    -    -
12345612345612345612345612345612345
-    -    -    -    -    -
12345671234567123456712345671234567
-    -    -    -    -    -
12345678123456781234567812345678123456781234567
-    -    -    -    -    -
12345678912345678912345678912345678912345678912345678
-    -    -    -    -    -
12345678901234567890123456789012345678901234567890123456789
-    -    -    -    -    -
```

A more flexible way of outputting text in columns is provided by the TAB statement. This is used as part of the PRINT statement, as follows:

```
INPUT "Please input three strings :",a$,b$,c$
PRINT a$;TAB(10);b$;TAB(15);c$
```

The argument which TAB takes determines the column at which the following string will be printed. If the current pen position lies beyond this column, then the string will be printed in the correct column of the line below. For example, entering the strings:

```
Fred, Jonathon, Steve
```

produces:

```
Fred  Jonathon
            Steve
```

This means that you can determine exactly the columns in which items will be printed.

Outputting text more precisely can be achieved in a similar manner using PTAB. This is the exception to the rule which says that text is positioned in terms of rows and columns. It allows you to move to a particular pixel along a line and print at that position. This means that, even with the default font, characters can be 'out of line' with each other. For example:

```
FOR loop% = 1 TO 16
  start.pos% = 315 - 5*(loop%-1)
```

```
  PRINT PTAB(start.pos%);ST
RING$(loop%,"*")
NEXT
```

This uses the fact that the width of each character in the default font is ten pixels to build a symmetrical tree of stars.

Note that the semi-colon following the TAB and PTAB statements is optional, ie:

```
PRINT TAB(10);a$
```

is equivalent to:

```
PRINT TAB(10)a$
```

Positioning Text

The next step on from positioning text at a particular column is locating it by giving both the column and line. You can do this using the statement LOCATE, for example:

```
LOCATE 10,20
```

moves the pen position to line 10, column 20. Subsequent PRINT statements will then start at that position. If you wish to change only the line or only the column, then you can give it just one value and it takes the other from the current position. For example:

```
LOCATE ,15
```

moves the pen to column 15 of the current line and:

```
LOCATE 5
```

moves the pen to line five but keeps it in the same column.

To read the current position yourself, use the following functions:

CSRLIN	Returns the line number
POS	Returns the column number

For example:

```
curlin% = CSRLIN
curcol% = POS(0)
```

Note that you must supply an argument to POS, however; this can be any value since it is ignored.

Note also that, if proportional fonts are being used or the pen has been aligned to pixel positions, then these values are only approximate.

These statements for positioning text and the other string handling routines introduced in this chapter are demonstrated in the following program. It is a small adventure game, as described in the introduction. The aim is to find the treasure. To do this you have to explore the different locations, looking for items to help you find it and avoid getting killed by the monsters which are out to get you.

A Final Example

The following program demonstrates the use of strings in an adventure game. It is only a small implementation, containing 20 rooms and recognising just a few one or two word instructions. However, it does show the principles involved. In addition, it provides the full framework on which you could create a full size game, should you wish to do so.

Try playing it. The object of the game is to find the treasure. To do so you will have to overcome a few obstacles on the way. You can move around using instructions such as 'GO NORTH' (this can actually be abbreviated to 'NORTH' or simply 'N'). 'INV' will give you an inventory of what you are carrying and 'LOOK' will repeat the description of your current location. Read the descriptions carefully: some of them contain clues as to what you should do. You'll have to work out the rest of the instructions for yourself. If you have problems solving it, then take a look at how the program works.

```
MapInit:
locs% = 20 : verbs% = 15
nouns% = 21 : items% = 9
OPTION BASE 1
DIM SHARED des$(locs%)
```

```
DIM SHARED N%(locs%),S%(locs%),E%(locs%),W%(locs%), U%(locs%),D%(locs%)
RESTORE roomdata
:

FOR room% = 1 TO locs%
 READ des$(room%),N%(room%),S%(room%),E%(room%),W%(room%), U%(room%)
,D%(room%) NEXT
DIM SHARED itemloc%(items%),itemdes$(items%),itemadj$(items%)
RESTORE itemdata:
FOR item% = 1 TO items%
  READ itemadj$(item%),itemdes$(item%),itemloc%(item%)
NEXT

MainBody:
dead%    = 0 : curloc% = 1
torchon% = 0 : baton%  = 0
bought%  = 0 : lock%   = 0
awake%   = 0
CALL look("")
WHILE dead% = 0
  PRINT
  LINE INPUT ":";sen$
  sen$ = UCASE$(sen$)
  CALL parse(sen$,words%,word1$, word2$)
  IF words% > 2 THEN
    CALL pp("Sorry, the sentence is too complex.")
    CALL pp("Please enter just one or two words.")
  ELSE
    CALL ident(word1$,word2$,words%,noun$,verb$,state%)
    IF state% = 1 THEN
    CALL act(noun$,verb$)
    END IF
  END IF
WEND
IF dead% = 1 THEN
  CALL pp("Sorry, you didn't make it!")
ELSE
  CALL pp("Well done.")
END IF
END

parse:
SUB parse(sen$,words%,word1$,word2$) STATIC
  CALL tspcrem(sen$)
  CALL lspcrem(sen$)
  spacepos% = INSTR(sen$," ")
  IF spacepos% = 0 THEN
    word1$ = sen$
    word2$ = ""
```

```
    words% = 1
  ELSE
    word1$ = MID$(sen$,1,spacepos%-1)
    sen$ = MID$(sen$,spacepos%+1)
    CALL lspcrem(sen$)
    spacepos% = INSTR(sen$," ")
    IF spacepos% = 0 THEN
      word2$ = sen$
      words% = 2
    ELSE
      words% = 3
    END IF
  END IF
END SUB

ident:
SUB ident(word1$,word2$,words%,noun$,verb$,state%) STATIC
SHARED nouns%, verbs%
  noun1% = 0 : noun2% = 0 : verb1% = 0 : verb2% = 0
  state% = 0 : noun$ = "" : verb$ = ""
  RESTORE nounlist:
  FOR loop% = 1 TO nouns%
    READ word$ : word$ = UCASE$(word$)
    IF word$ = word1$ THEN noun1% = 1
    IF word$ = word2$ THEN noun2% = 1
  NEXT
  RESTORE verblist:
  FOR loop% = 1 TO verbs%
    READ word$ : word$ = UCASE$(word$)
    IF word$ = word1$ THEN verb1% = 1
    IF word$ = word2$ THEN verb2% = 1
  NEXT
  IF noun1% = 1 AND noun2% = 1 THEN
    CALL pp("I can't cope with more than 1 noun")
  ELSEIF verb1% = 1 AND verb2% = 1 THEN
    CALL pp("I can't cope with more than 1 verb")
  ELSEIF noun1% = 0 AND noun2% = 0 AND verb1% = 0 AND verb2% = 0 THEN
    IF words% = 1 THEN
      CALL pp("I don't know that word")
    ELSE
      CALL pp("I don't know those words")
    END IF
  ELSEIF noun1% = 0 AND noun2% = 0 AND words% = 2 THEN
    IF verb1% = 1 THEN
      CALL pp("I don't know the noun '"+word2$+"'")
    ELSE
      CALL pp("I don't know the noun '"+word1$+"'")
    END IF
  ELSEIF verb1% = 0 AND verb2% = 0 AND words% = 2 THEN
```

```
    IF noun1% = 1 THEN
      CALL pp("I don't know the verb '"+word2$+"'")
    ELSE
      CALL pp("I don't know the verb '"+word1$+"'")
    END IF
  ELSE
    IF noun1% = 1 THEN noun$ = word1$
    IF noun2% = 1 THEN noun$ = word2$
    IF verb1% = 1 THEN verb$ = word1$
    IF verb2% = 1 THEN verb$ = word2$
    state% = 1
  END IF
END SUB

act:
SUB act(noun$,verb$) STATIC
  IF verb$ = "" THEN
    CALL direction(noun$,status%)
    IF status% = 1 THEN
      verb$ = "GO"
    ELSE
      CALL iown(noun$,status%)
      IF status% = 1 THEN
        verb$ = "DROP"
      ELSE
        verb$ = "GET"
      END IF
    END IF
  END IF

  IF     verb$ = "INV" THEN
    CALL inv(noun$)
  ELSEIF verb$ = "LOOK" THEN
    CALL look(noun$)
  ELSEIF verb$ = "IN" OR verb$ = "ENTER" THEN
    CALL enter(noun$)
  ELSEIF verb$ = "OUT" OR verb$ = "EXIT" THEN
    CALL leave(noun$)
  ELSEIF verb$ = "GO" THEN
    CALL go(noun$)
  ELSEIF verb$ = "GET" OR verb$ = "TAKE" THEN
    CALL take(noun$)
  ELSEIF verb$ = "DROP" THEN
    CALL drop(noun$)
  ELSEIF verb$ = "BUY" THEN
    CALL buy(noun$)
  ELSEIF verb$ = "EAT" THEN
    CALL eat(noun$)
  ELSEIF verb$ = "UNLOCK" THEN
```

```
    CALL unlock(noun$)
  ELSEIF verb$ = "ON" THEN
    CALL onv(noun$)
  ELSEIF verb$ = "OFF" THEN
    CALL offv(noun$)
  END IF
END SUB

direction:
SUB direction(noun$,status%) STATIC
  status% = 0
  RESTORE dirlist
  FOR loop% = 1 TO 12
    READ dir$
    IF UCASE$(dir$) = noun$ THEN status% = 1
  NEXT
END SUB

iown:
SUB iown(noun$,status%)STATIC
  status% = 0
  CALL itemid(noun$,item%)
  IF item% <> 0 THEN
    IF itemloc%(item%) = 0 THEN status% = 1
  END IF
END SUB

inv:
SUB inv(noun$) STATIC
SHARED items%
  IF noun$ = "" THEN
    count% = 0 : c% = 0
    FOR loop% = 1 TO items%
      IF itemloc%(loop%) = 0 THEN count% = count% + 1
    NEXT
    IF count% > 0 THEN
      CALL pp("You are holding")
      FOR loop% = 1 TO items%
        IF itemloc%(loop%) = 0 THEN
          IF c% = count% - 1 AND count% <> 1 THEN
            CALL pp("and")
          END IF
          CALL pp(itemadj$(loop%))
          IF c% = count% - 1 THEN
            CALL pp(itemdes$(loop%)+".")
          ELSEIF c% = count% - 2 THEN
            CALL pp(itemdes$(loop%))
          ELSE
            CALL pp(itemdes$(loop%)+",")
```

```
          END IF
          c% = c% + 1
        END IF
      NEXT
    ELSE
      CALL pp("You are not holding anything.")
    END IF
  ELSE
    CALL pp("I don't understand that instruction")
  END IF
END SUB

look:
SUB look(noun$) STATIC
SHARED curloc%,torchon%,baton%,bought%,lock%,dead%,awake%
  IF noun$ = "" THEN
    CALL pp("You are "+des$(curloc%))
    IF curloc% = 20 THEN
      dead% = -1
    ELSE
      IF torchon% = 1 AND baton% > 10 THEN
        CALL pp("Your torch has faded.")
        torchon% = 0
        IF (curloc%>=8 AND curloc%<=16) OR curloc%>=19 THEN
          CALL pp("It's dark in here.")
          CALL pp("If you move, you may fall into a pit.")
        END IF
      ELSEIF torchon% = 0 AND (curloc% = 8 OR curloc% = 16 OR curloc% =
      19) THEN
        CALL pp("It's dark in here.")
        CALL pp("If you move, you may fall into a pit.")
      END IF
      CALL exits(curloc%)
      CALL contents(curloc%)
    END IF
  ELSE
    CALL pp("I don't understand that instruction")
  END IF
END SUB

go:
SUB go(noun$) STATIC
SHARED curloc%,torchon%,baton%,bought%,lock%,dead%,awake%
  CALL direction(noun$,status%)
  IF status% = 1 THEN
    oldloc% = curloc%
    CALL iown("TORCH",status%)
    IF torchon% = 0 THEN
```

```
      IF (curloc% >= 8 AND curloc% <= 16) OR curloc% = 19 OR curloc% = 20
      THEN
        CALL pp("Oops, you've fallen into a pit and broken your neck.")
        dead% = 1
      END IF
    END IF

    IF dead% = 0 THEN
      moved% = 1
      IF noun$ = "NORTH" OR noun$= "N" THEN
        CALL checkgo(N%(),moved%)
      ELSEIF noun$ = "SOUTH" OR noun$= "S" THEN
        CALL checkgo(S%(),moved%)
      ELSEIF noun$ = "EAST" OR noun$= "E" THEN
        CALL checkgo(E%(),moved%)
      ELSEIF noun$ = "WEST" OR noun$= "W" THEN
        CALL checkgo(W%(),moved%)
      ELSEIF noun$ = "UP" OR noun$= "U" THEN
        CALL checkgo(U%(),moved%)
      ELSEIF noun$ = "DOWN" OR noun$= "D" THEN
        IF lock% = 0 AND curloc% = 18 THEN
          CALL pp("You can't - the trapdoor is locked.")
          moved% = 0
        ELSE
          CALL checkgo(D%(),moved%)
        END IF
      END IF
      IF moved% = 1 THEN
        IF torchon% = 1 THEN baton% = baton% + 1
        CALL look("")
        CALL giant(oldloc%)
      END IF
    END IF
  ELSE
    CALL pp("I don't understand that instruction")
  END IF
END SUB
SUB checkgo(dir%(),moved%) STATIC
SHARED curloc%
  IF dir%(curloc%) = 0 THEN
    CALL pp ("There is no way to go in that direction.")
    moved% = 0
  ELSE
    curloc% = dir%(curloc%)
  END IF
END SUB
SUB giant(oldloc%) STATIC
SHARED curloc%, awake%, dead%
  CALL itemid("GIANT",gi%)
```

```
  IF itemloc%(gi%) = curloc% THEN
    IF awake% = 1 THEN
      CALL pp("The giant sees you, licks his lips and eats you.")
      dead% = 1
    ELSE
      CALL itemid("FOOD",fo%)
      IF itemloc%(fo%) = 0 THEN
        CALL pp("The giant's nose starts twitching.")
        CALL pp("Suddenly he wakes up, sees you and the food and eats you
        both.")
        dead% = 1
      ELSE
        CALL pp("Although you tiptoe about very quietly, the giant starts
         to wake up.")
        awake% = 1
      END IF
    END IF
  ELSE
    IF awake% = 1 THEN
      itemloc%(gi%) = oldloc%
      CALL itemid("FOOD",fo%)
      IF itemloc%(gi%) = itemloc%(fo%) THEN
        awake% = 0
        itemloc%(fo%) = -1
        SAY(TRANSLATE$("Yummy, yummy"))
      END IF
    END IF
  END IF
END SUB

take:
SUB take(noun$) STATIC
SHARED curloc%,torchon%,baton%,bought%,lock%,dead%,awake%
  CALL itemid(noun$,item%)
  IF item% <> 0 THEN
    IF itemloc%(item%) = curloc% THEN
      IF     noun$ = "GIANT" THEN
        CALL pp("The giant wakes up as you struggle to lift him.")
        CALL pp("He is not amused and kills you with a single blow")
        dead% = 1
      ELSEIF noun$ = "BATTERY" AND bought% = 0 THEN
        CALL pp("The shop keeper isn't happy about you taking the battery
        without paying for it.")
        CALL pp("He calls the police to escort you away.")
        dead% = 1
      ELSE
        CALL pp("You get the "+itemdes$(item%)+".")
        itemloc%(item%) = 0
      END IF
```

```
    ELSE
      CALL pp("I see no "+itemdes$(item%)+" here.")
    END IF
  ELSE
  CALL pp("I don't understand that instruction")
  END IF
END SUB

drop:
SUB drop(noun$) STATIC
SHARED curloc%,torchon%,baton%,bought%,lock%,dead%,awake%
  CALL itemid(noun$,item%)
  IF item% <> 0 THEN
    IF itemloc%(item%) = 0 THEN
      CALL pp("You drop the "+itemdes$(item%)+".")
      itemloc%(item%) = curloc%
      IF (noun$ = "BATTERY" OR noun$ = "TORCH") THEN
        IF noun$ = "BATTERY" AND torchon% = 1 THEN
          CALL pp("Your torch has now gone out.")
          torchon% = 0
          IF (curloc%>=8 AND curloc%<=16) OR curloc%>=19 THEN
            CALL pp("It's dark in here.")
            CALL pp("If you move you may fall into a pit.")
          END IF
        ELSEIF noun$ = "TORCH" THEN
          CALL itemid("BATTERY",ba%)
          IF itemloc%(ba%) = 0 THEN
            itemloc%(ba%) = curloc%
            IF torchon% = 1 THEN
              CALL pp("The battery falls out of it and the light goes
              out.")
              torchon% = 0
              IF curloc%>=8 OR curloc%<=16 OR curloc%>=19 THEN
                CALL pp("It's dark in here.")
                CALL pp("If you move you may fall into a pit.")
              END IF
            ELSE
              CALL pp("The battery falls out of it.")
            END IF
          END IF
        END IF
      END IF
    ELSE
      CALL pp("You are not holding the "+itemdes$(item%)+".")
    END IF
  ELSE
   CALL pp("I don't understand that instruction")
  END IF
END SUB
```

```
enter:
SUB enter(noun$) STATIC
SHARED curloc%,torchon%,baton%,bought%,lock%,dead%,awake%
  IF     noun$ = "" THEN
    IF curloc% = 2 THEN
      curloc%= 3
      CALL look("")
    ELSEIF curloc% = 4 THEN
      curloc% = 5
      CALL look("")
    ELSE
      CALL pp("There is nothing here I can enter.")
    END IF
  ELSEIF noun$ = "HUT" THEN
    IF curloc% = 2 THEN
      curloc% = 3
      CALL look("")
    ELSE
      CALL pp("There isn't a hut here to enter.")
    END IF
  ELSEIF noun$ = "SHOP" THEN
    IF curloc% = 4 THEN
      curloc% = 5
      CALL look("")
    ELSE
      CALL pp("There isn't a shop here to enter.")
    END IF
  ELSE
    CALL pp("I don't understand that instruction")
  END IF
END SUB

leave:
SUB leave(noun$) STATIC
SHARED curloc%,torchon%,baton%,bought%,lock%,dead%,awake%
  IF     noun$ = "" THEN
    IF curloc% = 3 THEN
      curloc%= 2
      CALL look("")
    ELSEIF curloc% = 5 THEN
      curloc% = 4
      CALL look("")
    ELSE
      CALL pp("There is nothing here I can exit.")
    END IF
  ELSEIF noun$ = "HUT" THEN
    IF curloc% = 3 THEN
      curloc% = 2
```

```
      CALL look("")
    ELSE
      CALL pp("You're not inside a hut.")
    END IF
  ELSEIF noun$ = "SHOP" THEN
    IF curloc% = 5 THEN
      curloc% = 4
      CALL look("")
    ELSE
      CALL pp("You're not inside a shop.")
    END IF
  ELSE
    CALL pp("I don't understand that instruction")
  END IF
END SUB

buy:
SUB buy(noun$) STATIC
SHARED curloc%,torchon%,baton%,bought%,lock%,dead%,awake%
  CALL itemid(noun$,item%)
  IF item% <> 0 THEN
    IF itemloc%(item%) = curloc% THEN
      IF noun$ = "BATTERY" THEN
        IF bought% = 0 THEN
          CALL itemid("SOVEREIGN",sov%)
          IF itemloc%(sov%) = 0 THEN
            CALL pp("The shopkeeper takes your sovereign in exchange for
             the battery.")
            itemloc%(item%) = 0
            itemloc%(sov%) = -1
            bought% = 1
          ELSE
            CALL pp("You have nothing to pay for it with.")
          END IF
        ELSE
          CALL pp("You've already paid for it.")
        END IF
      ELSE
        CALL pp("It appears not to belong to anyone.")
        CALL pp("I should just take it if I were you.")
      END IF
    ELSE
      CALL pp("I see no "+itemdes$(item%)+" here.")
    END IF
  ELSE
    CALL pp("I can't apply that without a noun")
  END IF
END SUB
```

```
eat:
SUB eat(noun$) STATIC
SHARED curloc%,torchon%,baton%,bought%,lock%,dead%,awake%
  IF noun$ <> "" THEN
    IF noun$ = "FOOD" THEN
      CALL itemid("FOOD",fo%)
      IF itemloc%(fo%) = 0 THEN
        CALL pp("You eat the food and immediately fall asleep.")
        CALL pp("Some time later you wake up again, feeling very sick.")
        itemloc%(fo%) = -1
        IF torchon% = 1 THEN baton% = baton% + 5
      ELSE
        CALL pp("You haven't got any food.")
      END IF
    ELSE
      CALL pp("Don't be silly!")
    END IF
  ELSE
    CALL pp("I can't apply that without a noun.")
  END IF
END SUB

unlock:
SUB unlock(noun$) STATIC
SHARED curloc%,torchon%,baton%,bought%,lock%,dead%,awake%
  IF noun$ <> "" THEN
    IF noun$ = "DOOR" THEN
      IF curloc% = 18 THEN
        CALL itemid("KEY",ke%)
        IF itemloc%(ke%) = 0 THEN
          IF lock% = 0 THEN
            CALL pp("You unlock the door with your key.")
            lock% = 1
          ELSE
            CALL pp("The door is already unlocked.")
          END IF
        ELSE
          CALL pp("You haven't got the key.")
        END IF
      ELSE
        CALL pp("There isn't a door here.")
      END IF
    ELSE
      CALL pp("I don't understand that instruction.")
    END IF
  ELSE
    CALL pp("I can't apply that without a noun.")
  END IF
END SUB
```

```
onv:
SUB onv(noun$) STATIC
SHARED curloc%,torchon%,baton%,bought%,lock%,dead%,awake%
  IF noun$ <> "" THEN
    IF noun$ = "TORCH" THEN
      CALL itemid("TORCH",tor%)
      IF itemloc%(tor%) = 0 THEN
        CALL itemid("BATTERY",bat%)
        IF itemloc%(bat%) = 0 THEN
          IF torchon% <> 1 THEN
            IF baton% <= 10 THEN
              CALL pp("Your torch is now on.")
              torchon% = 1
            ELSE
              CALL pp("Sorry, the battery is dead.")
            END IF
          ELSE
            CALL pp("The torch is already on.")
          END IF
        ELSE
          CALL pp("You don't have a battery for it.")
        END IF
      ELSE
        CALL pp("You aren't carrying a torch.")
      END IF
    ELSE
      CALL pp("I don't understand that instruction.")
    END IF
  ELSE
    CALL pp("I can't apply that without a noun.")
  END IF
END SUB

offv:
SUB offv(noun$) STATIC
SHARED curloc%,torchon%,baton%,bought%,lock%,dead%,awake%
  IF noun$ <> "" THEN
    IF noun$ = "TORCH" THEN
      CALL itemid("TORCH",tor%)
      IF itemloc%(tor%) = 0 THEN
        IF torchon% = 1 THEN
          CALL pp("Your torch is now off.")
            torchon% = 0
            IF (curloc%>=8 AND curloc%<=16) OR curloc%>=19 THEN
              CALL pp("It's dark in here.")
              CALL pp("If you move you may fall into a pit.")
            END IF
          ELSE
```

```
            CALL pp("The torch is already off.")
          END IF
        ELSE
        CALL pp("You aren't carrying a torch.")
      END IF
    ELSE
      CALL pp("I don't understand that instruction.")
    END IF
  ELSE
    CALL pp("I can't apply that without a noun.")
  END IF
END SUB

exits:
SUB exits(room%) STATIC
  count% = 0 : c% = 0
  IF N%(room%) <> 0 THEN count% = count% + 1
  IF S%(room%) <> 0 THEN count% = count% + 1
  IF E%(room%) <> 0 THEN count% = count% + 1
  IF W%(room%) <> 0 THEN count% = count% + 1
  IF U%(room%) <> 0 THEN count% = count% + 1
  IF D%(room%) <> 0 THEN count% = count% + 1
  IF count% = 1 THEN
    CALL pp("There is an exit")
  ELSEIF count% > 1 THEN
    CALL pp("There are exits")
  END IF
  IF N%(room%) <> 0 THEN CALL eachexit("north",c%,count%)
  IF S%(room%) <> 0 THEN CALL eachexit("south",c%,count%)
  IF E%(room%) <> 0 THEN CALL eachexit("east",c%,count%)
  IF W%(room%) <> 0 THEN CALL eachexit("west",c%,count%)
  IF U%(room%) <> 0 THEN CALL eachexit("up",c%,count%)
  IF D%(room%) <> 0 THEN CALL eachexit("down",c%,count%)
END SUB

eachexit:
SUB eachexit(dir$,c%,count%) STATIC
  IF count% = 1 THEN
    CALL pp(dir$+".")
  ELSEIF c% = count% - 1 THEN
    CALL pp("and "+dir$+".")
  ELSEIF c% = count% - 2 THEN
    CALL pp(dir$)
  ELSE
    CALL pp(dir$+",")
  END IF
  c% = c% + 1
END SUB
```

```
contents:
SUB contents(room%) STATIC
SHARED curloc%,items%,awake%
  count% = 0 : c% = 0
  FOR loop% = 1 TO items%
    IF itemloc%(loop%) = curloc% THEN count% = count% + 1
  NEXT
  IF count% = 1 THEN
    CALL pp("There is")
  ELSEIF count% > 1 THEN
    CALL pp("There are")
  END IF
  IF count% > 0 THEN
    FOR loop% = 1 TO items%
      IF itemloc%(loop%) = curloc% THEN
        IF c% = count% - 1 AND count% <> 1 THEN
          CALL pp("and")
        END IF
        IF itemdes$(loop%) = "giant" AND awake% = 1 THEN
          CALL pp(" an angry looking")
        ELSE
          CALL pp(itemadj$(loop%))
        END IF
        IF c% = count% - 1 THEN
          CALL pp(itemdes$(loop%)+" here.")
        ELSEIF c% = count% - 2 THEN
          CALL pp(itemdes$(loop%))
        ELSE
          CALL pp(itemdes$(loop%)+",")
        END IF
        c% = c% + 1
      END IF
    NEXT
  END IF
END SUB

lspcrem:
SUB lspcrem(sen$) STATIC
  ch$ = " "
  WHILE ch$ = " "
    ch$ = LEFT$(sen$,1)
    IF ch$ = " " THEN sen$ = MID$(sen$,2)
  WEND
END SUB

tspcrem:
SUB tspcrem(sen$) STATIC
  ch$ = " "
  WHILE ch$ = " "
```

```
    ch$ = RIGHT$(sen$,1)
    IF ch$ = " " THEN sen$ = MID$(sen$,1,LEN(sen$)-1)
  WEND
END SUB

pp:
SUB pp(sen$) STATIC
  spos% = 0 : done% = 0 : chars% = 60
  WHILE done% = 0
    oldspos% = spos% + 1
    spos% = INSTR(oldspos%,sen$," ")
    IF spos% > 0 THEN
      word$ = MID$(sen$,oldspos%,spos%-oldspos%)
      IF POS(0) > chars% - (LEN(word$) + 1) THEN PRINT
      PRINT " ";word$;
    ELSE
      done% = 1
    END IF
  WEND
  word$ = MID$(sen$,oldspos%)
  IF POS(0) > chars% - (LEN(word$) + 1) THEN PRINT
  PRINT " ";word$;
END SUB

itemid:
SUB itemid(noun$,item%) STATIC
SHARED items%
  item% = 0
  FOR loop% = 1 TO items%
    IF UCASE$(itemdes$(loop%))= UCASE$(noun$) THEN
      item% = loop%
    END IF
  NEXT
END SUB

itemdata:
DATA "a shiny gold","sovereign",0
DATA "a sturdy grey","torch",3
DATA "a torch","battery",5
DATA "a tasty morsel of","food",11
DATA "a loudly snoring","giant",16
DATA "a large metal","key",13
DATA "","hut",-1
DATA "","shop",-1
DATA "","door",-1

nounlist:
dirlist:
DATA north, n, south, s, east, e, west, w, up, u, down, d
```

```
DATA sovereign, torch, battery, food, giant, key
DATA hut, shop, door
verblist:
DATA look, inv, go, in, out, enter, exit
DATA get, take, drop, buy, on, off, eat, unlock roomdata:
DATA "in clearing in the forest.",0,2,0,0,0,0
DATA "at a junction between well-worn paths. A small hut stands at the
roadside.",1,6,4,0,0,0
DATA "inside the hut.",0,0,0,0,0,0
DATA "at the end of the path in front of a strange looking shop.",0,0,
0,2,0,0
DATA "inside the shop. All kinds of strange objects stock the shelves.
 In the centre is a table containing this week's special offers."
,0,0,0,0,0,0
DATA "at the edge of a small lake.",2,7,0,0,0,0
DATA "following a small, stream which abruptly disappears into the ground
.",6,0,0,0,0,8
DATA "in a musty smelling corridor.",0,0,9,0,7,0
DATA "in a windy passage way.",0,10,0,8,0,0
DATA "inside a giant hall.",9,16,11,0,0,0
DATA "in a huge kitchen. Cooking implements are littered around the place
 and the aroma of baking fills the air.",0,12,0,10,0,0
DATA "in a dining hall. The room is dominated by a massive table in the
centre with a single chair at its head.",11,14,13,0,0,0
DATA "in a study. Row upon row of leather bound volumes deck the walls."
,0,0,0,12,0,0
DATA "in a bedroom. A huge single bed is placed against one wall and an
equally huge wardrobe against another.",12,0,0,15,0,0
DATA "in a corridor.",16,0,14,0,0,0
DATA "in the sitting room. There is an armchair in one corner but very
little other furniture.",10,15,0,17,0,0
DATA "outside in a fantastic garden. Beautiful flowers are growing
everywhere and gently singing to each other. The air is heavy with their
scent.",0,18,16,0,0,0
DATA "at the edge of the garden. There is a trap door in the ground."
,17,0,0,0,0,19
DATA "in a small, damp corridor.",0,20,0,0,18,0
DATA "in the treasure room. You've reached your destination.",0,0,0,0,0,0
```

The program works as follows:

MapInit:
It starts by setting up the following variables:

locs%	= number of rooms
verbs%	= number of verbs it knows
nouns%	= number of nouns it knows (items% + the 12 directions)

items% = number of items it knows

Then it sets up various arrays, reading the data for them from the data statements at the bottom of the program:

des$	holds the descriptions of the rooms
N%	holds the number of the location to the north of each room
S%	holds the number of the location to the south of each room
E%	holds the number of the location to the east of each room
W%	holds the number of the location to the west of each room
U%	holds the number of the location up from each room
D%	holds the number of the location down from each room
itemloc%	holds the number of the location of each item
itemdes$	holds the definition of each item
itemadj$	holds the adjective describing each item

Note that, if an item has a location number of 0, then it is being carried by the player. If the number is –1, then the object no longer exists or is being treated as a special object.

Note also that the arrays are all SHARED, so that the contents of them can be accessed by the subprograms as well as the main program.

Main Body:
This starts by setting up various flags:

curloc%	= number of the current room
dead%	= 0 if the player is still alive = 1 if the player is dead = –1 if the player has successfully completed the game
torchon%	= 0 if the torch is off and 1 otherwise
baton%	= number of moves the battery has been on for
bought%	= 0 if the battery has not yet been bought = 1 otherwise
lock%	= 0 if the trapdoor is locked and 1 otherwise
awake%	= 0 if the giant is asleep and 1 otherwise

These flags are SHARED by the subprograms which occur later on in the program to do the work. This means that they get updated automatically as the state of one of the items changes.

Then the program calls the subprogram 'look' to print the description for the current room.

The main loop continues while the player is still alive but has not yet completed the game. This loop reads in an instruction from the user (LINE INPUT assigns the whole line to the variable, including characters which INPUT on its own treats as separators). This is then parsed to obtain the individual words in it by the subprogram 'parse'. Provided that just one or two words were given, these are then analysed to see if the program recognises them by the subprogram 'ident'. This subprogram prints out messages if the individual words don't make sense and returns 'state%' as 0. Otherwise, it assigns the appropriate words to:

verb$ = verb input or the null string if no verb was given
noun$ = noun input or the null string if no noun was given

and returns 'state%' as one. Finally, the subprogram 'act' is called to act upon the instruction.

parse:
The parser starts by calling other subprograms to remove the leading and trailing spaces from the instruction. Then it searches for a space. If one is not found, the instruction must consist of just a single word. This is assigned to 'word1$' and a null string is placed in 'word2$'.

If a space is found, then the letters up to the space are placed in 'word1$' and 'sen$' is altered by these letters and the space being removed from the start of it. Note that, since 'sen$' is being passed from 'act' without being enclosed in round brackets, then the contents of the variable being passed are being altered by this action. This doesn't matter since it is not going to be used again. If you wished, you could use the brackets to ensure that it is not changed.

Then any leading spaces are removed again. This is so that two words which are typed in with more than one space separating them are dealt with correctly. Again a space is looked for. If one is found then the sentence must have contained at least three words and so

'words%' is set to three and the main body rejects it. Otherwise, the remaining word is placed in 'word2$'.

ident:
This subprogram tries to identify the words entered. It loops round all the nouns and verbs which is knows. For each one, if it matches one of the words, a flag is set:

noun1% = 0 if word1$ is not a noun and 1 otherwise
noun2% = 0 if word2$ is not a noun and 1 otherwise
verb1% = 0 if word1$ is not a verb and 1 otherwise
verb2% = 0 if word2$ is not a verb and 1 otherwise

It then checks for the combinations which are disallowed. Two verbs or two nouns are rejected. Similarly, if it doesn't recognise any of them, it complains. The other checks ensure that, if two words were typed and only one is recognised, then the program states that it doesn't know about the other one. Finally, 'noun$' and 'verb$' are assigned the values of 'word1$' and 'word2$' according to the results of the search.

act:
This is the subprogram which deals with individual verbs. It starts by trying to guess which verb was meant when only a noun was given. If the noun is a direction, then the verb is assumed to be 'GO'. Otherwise it is assumed to be either 'DROP' or 'GET', depending on whether or not the player is currently holding the noun.

Having now ensured that it has a verb, it calls the appropriate subprogram to deal with it.

direction:
This is used by 'act' to determine whether the noun is a direction. It loops round, reading all the directions and checking to see if they match the noun which is passed as its first parameter. If the noun is a direction, the second parameter, 'status%', is returned as one; otherwise, it is returned as zero.

iown:
This again is used by 'act'. It checks to see if the noun given is the name of an item which the player is currently holding. 'itemid%' gives the number of a particular item. The location of this is then checked to

see if it is zero (ie belongs to the player). Again the second parameter is returned as one if the routine successfully found that the item was owned and zero otherwise.

inv:
This is the first of the subprograms for handling a verb. It checks that the 'INV' instruction has not been followed by a noun and, if not, it counts the number of items which the player is holding. If the number is zero then a message is printed saying that nothing is being held. Otherwise the beginning of the sentence 'You are holding' is printed, and a list of the items is output.

Note the way in which the list is given. If the current item is the last one (ie the count of items already printed is one less than the total number of items) then a full-stop is placed after it. If, in addition, there have been others in the list, then its description is preceded by the word 'and'. If it is the second to last (the count is two less than the total) then it is printed on its own. Finally, any other item is printed followed by a comma to separate it from the ones which are still to come. This allows you to obtain the following types of lists (the items are not ones which actually occur in the game):

```
You are holding a red ball.
```

The red ball is the last item so it is printed with a full stop after it. Since there were no others, nothing is printed before it.

```
You are holding a blue bag and a red ball.
```

The blue bag is the second to last item so it is printed normally. This time, the red ball is not the only item so it is preceded by the word 'and'.

```
You are holding a green lollipop, a blue bag and a red
ball.
```

The green lollipop is not the last nor the second to last item, so it is printed followed by a comma. The others are given as above.

look:
Provided that the instruction was given on its own, this subprogram prints out the description of the current room. If this room is the last one then it sets 'dead' to –1 to indicate that the player has succeeded.

It then goes on to deal with the state of the torch. If the battery has run out then 'torchon%' is set to 0 and a message is printed. If, in addition, the player is currently in an underground room, a warning message is given.

If 'torchon%' was already 0, and the player is in a room which is one move away from the outside world, then the same warning is issued. This ensures that when he goes indoors, he is warned about the possibility of falling into a pit and given the opportunity of preventing it from happening.

Finally, subprograms are called to list the exits available and any items which there may be in the current location.

go:
This subprogram deals with moving the player about. It checks to make sure that the noun is a valid location and, if so, starts by saving the number of the current location.

It then checks to see if the torch is on. If it isn't and the current location is one of the underground rooms, then this means that the player is moving about in the dark. It therefore tells him that he has fallen into a pit and sets 'dead%' to one.

The next block deals with each direction in turn. It starts by assuming that the move will take place and sets the local variable 'moved%' to one. Then it calls a subprogram to check if there is any way to go in that direction. If there is, then 'curloc%' is updated. If not, the subprogram prints a message and sets 'moved%' to 0. The only special case is when the player is trying to go down through the trap door in location 18. He is prevented from doing this if the door is locked.

Provided that the move succeeded, the number of moves that the battery has been on for is updated if the torch is on. Then the description of the new location is printed. Finally, a separate subprogram is called to handle the special cases involving the giant.

checkgo:
This is a subprogram used by 'go'. It is passed an array which contains the locations lying in a particular direction from each room. Then for the current location, it checks to see if there is any way to go in that direction and acts as described above.

giant:
Again this is used only by 'go'. It starts by finding where the giant is and, if he is in the current location, then it makes further checks. If he is awake then he eats the player. If he is asleep but the player is carrying the food he wakes up in a hurry and eats the player and the food. If he is asleep and the player hasn't got the food with him then he wakes up but takes no immediate aggressive action.

While the giant is awake, he is moved to the location which the player has just left. This means that, if the player backtracks, he will meet a hungry, wide awake giant and will be killed. The final twist is that if the giant comes across a room containing the food, he eats it and falls asleep again. The player is given a clue about this happening by the words 'Yummy, yummy' being spoken. (Speech will be dealt with properly in a later chapter.) This means that the player, on returning to this location, will find him slumbering peacefully.

take:
This deals with the verbs 'GET' and 'TAKE'. The first check is that these have been followed by a noun to indicate what is to be taken. Next a check is made to ensure that the item wanted is actually in the current location. Then normally a message is given saying that the item has been obtained and the location of the item is altered to indicate that it now is being carried. One exception to this is when the player tries to get the giant. In this case the giant kills him. The other is when he tries to get the battery for the first time. This is classed as shop lifting and the player is removed from the game.

drop:
This is the opposite to 'take'. It checks to make sure that the item specified is currently being carried and, if so, transfers its location to the current room. The special cases are when dropping the torch or battery. If the battery is dropped whilst the torch is on, then the torch automatically goes out. If the torch is dropped whilst the player is also

holding the battery, then the battery falls with it. Again the torch, if it was on, is extinguished. In either case, if the torch goes out whilst the player is underground, the warning about falling into a pit is given.

enter:
This deals with 'ENTER' and 'IN'. It can be used on its own or with the nouns 'SHOP' or 'HUT'. If it is used on its own, it enters whichever of these two the player is next to. All it does is to alter the current location to be inside either of these and to call 'look' to print out the description of this room. However, it does check beforehand that the player is actually standing next to the building he wants to enter.

leave:
This deals with 'EXIT' and 'OUT' in a similar manner to 'enter' above.

buy:
This is provided to allow the battery to be obtained. Trying to buy any other object in the room just produces a message telling the player to take it. The battery can only be bought if: it is in the current location, it hasn't been bought already and the player is holding the sovereign to pay for it with.

eat:
Trying to eat anything other than the food gives an appropriate message! Eating the food effectively puts the player to sleep for five game moves. If the torch is on, the number of moves the battery has been on for is increased by this number. The only other effect is to destroy the food – its location is set to –1.

unlock:
The only item which can be unlocked is the door. The checks made are that: the door is in the current location, it is currently locked and that the player has the key. If so, 'lock%' is set to one so that it is possible to move down through it later.

onv:
This deals with turning the torch on. For this to be possible, the player must be carrying both the torch and the battery, and the battery must not have already been on for more than 10 game moves. If this is the

case, then 'torchon%' is set to one. Trying to turn the torch on when it is already on has no effect other than producing a message.

offv:
This turns the torch off again provided it is being carried and that it is currently on. Turning it off sets 'torchon%' back to 0. In addition, if the player is currently underground, the warning about pits is given.

exits:
This is the subprogram which prints out all the exits available from the current location. It starts by counting them and then goes through them again, printing those available. The method used to do this is the same as that used by 'inv' to deal with items.

eachexit:
This subprogram is called from 'exits' above. It handles one direction at a time.

contents:
This is similar to 'inv', except that it lists the items in a particular location, rather than those being carried by the player. It starts by counting the number of items and then printing 'There is' or 'There are' depending on whether or not more than one has been found. Then the items and their adjectives are printed. The method of deciding when to print commas or the word 'and' is the same as in 'inv'. The list is terminated by the word 'here.'

There is one special case which this subprogram deals with. If the giant is awake, then the description stored in the array, 'a loudly snoring', is replaced by 'an angry looking'.

lspcrem:
This removes leading spaces from a string. If checks whether or not the left-hand character is a space. If so, it converts the string into itself minus the first character and repeats the check.

tspcrem:
This removes trailing spaces by repeatedly checking the right-hand character and removing it when necessary.

pp:
This subprogram is the 'pretty printer'. All printing performed by the program is carried out by passing the string to be printed to this routine. It acts by locating the individual words in the sentence and then printing them out, preceded by a space. If the word won't fit on the current line, then a new one is started.

The individual words are extracted by finding the successive spaces in the string and then taking the letters between the previous space and the current one. Note that it is assumed that the string does not start or end with spaces and that words are separated by just a single string. Since this routine is dealing only with text contained in the program and not with text which the player inputs, this is a legitimate simplification. Note also that, when no further space is found, the last word still needs to be printed. The number of words is always one greater than the number of spaces.

The width of the screen has been set to 60 characters. To determine whether or not a word will fit on the current line, POS(0) is called to return the column that the text cursor is in. The maximum width of the window is set to be 60 (it is held in chars%). If the maximum width is less than the current position plus the length of the word plus one (for the space in front of it), then a PRINT command is given to move to the next line. A space and the word is then printed and the text cursor is left at the end of the word.

Note that a space is printed even when a new line is being used. This means that all text starts in column two. Hence the prompts for user input which are printed in the first column are more distinctive.

itemid:
This is passed a noun and it returns a number stating which element it is in the item arrays. If the noun is not an item, then the number returned is 0.

itemdata:
These data statements contain the adjectives, names and initial locations of all the items which are read into arrays at the start of the program. Note that, since the hut, shop and door aren't to be included in the list of items which can be picked up and dropped, their location

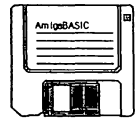
is set to –1. This means that they are never found and consequently that no adjective is required for them.

noundata:
This marks the start of the data statements containing the list of nouns which the program knows about.

dirlist:
Similarly, this marks the start of the list of directions.

verblist:
And this, the list of verbs.

roomdata:
Finally comes the data for each of the rooms in the form of its description and then six integers. Each integer gives the number of the location which is north, south, east, west, up or down respectively from the room described.



6 : More On Graphics

'A picture is worth a thousand words' is how the saying goes. To help you to cut down the number of words you need, this chapter returns to the subject of graphics. It completes the list of commands which BASIC provides for creating different outlines and solid shapes. Then, by moving on to show you how to create your own screens and windows, it makes available to you up to 32 colours to experiment with.

The use of mouse and menus is heavily involved in all screen activities. Therefore, these two topics are also dealt with fully. Finally, to prevent you from losing your masterpiece once you have created it, the secrets of how to save a copy of all or part of a window to disc will be revealed.

Painting In Areas

Most of the graphics commands we have met so far have only drawn the outline of shapes, the exception being the LINE command which can be used to create solid rectangles. However, without solid shapes, graphics look very drab. We have already 'improvised' in order to create solid shapes in the traffic lights example in Chapter Four, which required solid circles to be plotted. We created these by drawing a series of circles, centred at the same position, each with a radius one greater than the previous one. This achieved the effect we wanted but not very efficiently.

So why doesn't AmigaBASIC provide commands to create solid circles, sectors, ellipses etc? The answer lies in the PAINT command. This command is a general purpose one which can be used to fill any enclosed area with a given colour. Therefore to obtain a solid circle, all you have to do is draw its outline and then PAINT it.

In its simplest form, PAINT takes the follow format:

```
PAINT (50,50)
```

The co-ordinates of any position within the region to be filled can be used. PAINT will start at the point given and fill in all directions until it meets the border of the shape. As with all the other graphics commands, the position can be given relative to the previous location by using the STEP option:

```
PAINT STEP (0,0)
```

There are two further arguments which it can be given. The first is the colour with which the region is to be filled. For example:

```
PAINT (50,50), 2
```

will fill a region with colour two. If no value is specified, the current foreground colour is used.

The second is the colour of the border. PAINT can only recognise borders of a single colour. Consider the example of a small circle of colour one inside a larger circle of colour two. If you start filling from the centre using a border colour of one, just the small circle will be coloured. However, if you use a border colour of two, the whole of the large one will be painted. In the second case, PAINT doesn't recognise the line in colour one as anything special, so doesn't stop when it reaches it – only lines of colour two limit its action. These examples are given below for you to try out:

```
PALETTE 0,0,0,0
PALETTE 1,1,0,0
PALETTE 2,0,1,0
PALETTE 3,0,0,1
CLS
CIRCLE (320,100),40,1
CIRCLE (320,100),80,2
PAINT (320,100),3,1
PALETTE 0,0,0,0
PALETTE 1,1,0,0
PALETTE 2,0,1,0
PALETTE 3,0,0,1
CLS
CIRCLE (320,100),40,1
CIRCLE (320,100),80,2
```

```
PAINT (320,100),3,2
```

By default, the border colour is the same as the paint colour. Try altering the example above yet again to give:

```
PALETTE 0,0,0,0
PALETTE 1,1,0,0
PALETTE 2,0,1,0
PALETTE 3,0,0,1
CLS
CIRCLE (320,100),40,1
CIRCLE (320,100),80,2
PAINT (320,100),3
```

Now PAINT uses the paint colour, ie colour three, as the border colour. Since there are no lines drawn in colour three to limit the area being filled, the whole screen is painted.

Further examples of PAINT will be given later on in this chapter.

Polygons and Patterns

We can now produce both the outline and solid versions of the standard shapes; rectangles, circles and ellipses. But there are many other shapes which we might want to draw, such as triangles, hexagons, etc. We could produce these with the LINE command, for example:

```
LINE (100, 80) - (500, 80)
LINE (500, 80) - (250,150)
LINE (250,150) - (100, 80)
```

draws a triangle which we could then fill in using PAINT:

```
PAINT (250,120)
```

However, we have had to supply a lot of redundant information using this method. The co-ordinates for each corner had to be given twice: once as the end of one line and once as the start of the next line. Since polygons are such common things to want to create, AmigaBASIC provides us with a simpler method of making them, in the form of AREA and AREAFILL:

```
AREA (100, 80)
AREA (500, 80)
AREA (250,150)
AREAFILL
```

Each AREA statement defines a point of the polygon to be drawn, and the AREAFILL statement creates this polygon. Like the other graphics statements, AREA can take co-ordinates which are given relative to the previous one by using STEP. Having seen how the other statements work, you might also expect AREAFILL to take an argument which specifies a colour to use. However, this is not the case. It is actually much more versatile than that. Instead of just letting you fill the area with a single colour, AREAFILL allows you to fill it with a 'colour pattern'. The pattern to use has to be defined, prior to the AREAFILL statement, using the PATTERN command. To see this in action, try the following:

```
COLOR 1,3
DIM pat%(7)
pat%(0) = &HFFFF : REM The &H means that
pat%(1) = &H8080 : REM the numbers are
pat%(2) = &H8080 : REM being given in
pat%(3) = &H8080 : REM hexadecimal format
pat%(4) = &HFFFF
pat%(5) = &H808
pat%(6) = &H808
pat%(7) = &H808
PATTERN,pat%
AREA (100, 80)
AREA (500, 80)
AREA (250,150)
AREAFILL
```

This fills the triangle with a brickwork pattern. PATTERN is quite a complicated command to use. It can take up to two arguments. The first (which has been omitted in the example above) sets up a pattern to be used for drawing lines – we will move onto this a little later. The second argument is the one we are interested in at the moment, since this specifies a pattern for filling polygons. This argument must be an integer array containing at least two elements. Each element of the array defines a block which is 16 pixels across, the individual elements

define subsequent rows of the block. This is illustrated below in Figure 6.1.

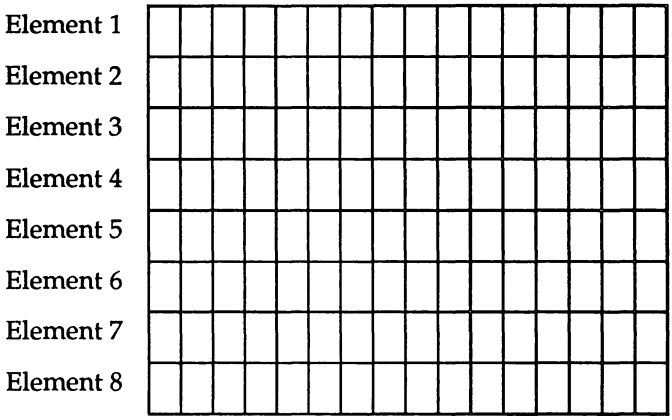


Figure 6.1. Each element of the array defines a block which is 16 pixels across, the individual elements define subsequent rows of the block.

The individual elements are best thought of as bit patterns. Each bit of the number corresponds to one pixel. For example, the hexadecimal numbers we used above have the following bit patterns:

```
&HFFFF      1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
&H8080      1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
&H8080      1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
&H8080      1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
&HFFFF      1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
&H0808      0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0
&H0808      0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0
&H0808      0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0
```

If you study the bit patterns above, you can see how the brick pattern was formed. The AREAFILL command coloured all the pixels corresponding to bits which were set in the current foreground colour, and all the pixels corresponding to bits which were unset in the current background colour. Therefore the 1s created the white 'mortar' and the 0s the red 'bricks'.

The block defined using PATTERN is repeated throughout the whole area to be filled. The width of the pattern is limited to 16 pixels,

therefore patterns will always repeat every 16 pixels. However, the number of rows defined can be altered by using larger arrays. The one limitation is that the number used must be a power of two, ie, 2, 4, 8, 16 etc.

The pattern defined will be used by all AREAFILL commands until the PATTERN command is used again to create a new one. Altering the pattern will not affect any existing areas already coloured in. The following program draws different regular polygons randomly positioned on the screen and fills each in an appropriate fill pattern:

```
REM Set up three different patterns
patterns:
DIM tri%(7)
DIM squ%(7)
DIM hex%(7)
FOR loop% = 0 TO 7
READ tri%(loop%)
NEXT
FOR loop% = 0 TO 7
READ squ%(loop%)
NEXT

FOR loop% = 0 TO 7
READ hex%(loop%)
NEXT
RANDOMIZE TIMER

REM Choose shape at random
mainloop:
FOR loop% = 1 TO 50
IF RND < .33 THEN
  CALL shape(3,1,tri%())
ELSEIF RND < .66 THEN
  CALL shape(4,2,squ%())
ELSE
  CALL shape(6,3,hex%())
END IF
NEXT loop%

shape:
SUB shape(edges%, col%, pat%(1)) STATIC
  xsize% = 60
  ysize% = 30
  xcen% = xsize% + INT(RND*(600-2*xsize%))
  ycen% = ysize% + INT(RND*(180-2*ysize%))
  pi = 3.14159
```

```
  FOR side% = 1 TO edges%
    ang = (2*pi/edges%)*side%  - pi/2
    AREA (xcen%+xsize%*COS(ang),ycen%+ysize%*SIN(ang))
  NEXT
  COLOR col%, col% MOD 3 + 1
  PATTERN ,pat%
  AREAFILL
END SUB

tri:
DATA &H0180, &H03C0, &H07E0. &H0FF0
DATA &H1FF8, &H3FFC, &H0000, &H0000
square:
DATA &H0180, &H07E0, &H1FF8, &H7FFE
DATA &H1FF8, &H07E0, &H0180, &H0000
hex:
DATA &H0180, &H07E0, &H1FF8, &H1FF8
DATA &H1FF8, &H07E0, &H0180, &H0000
```

The program starts by setting up three patterns, each eight rows deep. The values used are read from the data statements at the end of the programs; their bit patterns are shown below:

```
tri%              squ%              hex%
0000000110000000  0000000110000000  0000000110000000
0000001111000000  0000011111100000  0000011111100000
0000011111100000  0001111111111000  0001111111111000
0000111111110000  0111111111111110  0001111111111000
0001111111111000  0001111111111000  0001111111111000
0011111111111100  0000011111100000  0000011111100000
0000000000000000  0000000110000000  0000000110000000
0000000000000000  0000000000000000  0000000000000000
```

Then it chooses at random whether to draw a triangle, square or hexagon. All the shapes are drawn by the general purpose subprogram 'shape'. It takes three arguments: the number of sides which the polygon has and the colour and definition for the pattern used to fill it.

Shape starts by assigning values to the variables 'xsize%' and 'ysize%'. These determine the maximum dimensions of the polygon. Note that ysize% is double xsize%. This is to help counteract the fact that the pixels are roughly twice as high as they are wide. Then the subprogram selects the centre of the polygon at random, ensuring that

it is at least xsize% from the sides of the screen and ysize% from the top and bottom. This means that all the corners of the polygon will lie totally within the window, provided that the window is the full size of the screen. It is important to do this since AREA will produce an 'Illegal function call' error message if the co-ordinates it is given do not lie within the window.

The loop inside shape goes from one to the number of edges, and hence corners, that the polygon has. For each corner, shape calculates the position the point is at. The diagrams below in Figure 6.2 show the maths used to do this.

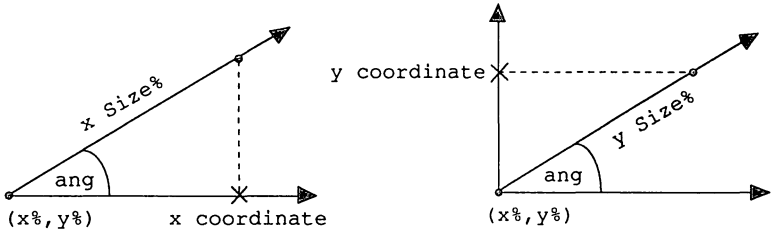


Figure 6.2.

Then the subprogram sets the current foreground and background colours. The foreground colour is determined by the value passed to shape as a parameter. The background colour is set according to the foreground colour as follows:

Foreground	Background
1	2
2	3
3	1

Finally, 'shape' sets up the pattern to be used, and fills in the polygon.

There is an alternative way of filling in an area using AREAFILL. In each case so far, AREAFILL has coloured the pixels corresponding to bits which are set in the foreground colour, and those corresponding to bits which are unset in the background colour. However, instead, we can choose to have it act in a different manner by giving it the argument one. In this mode, AREAFILL ignores the current

foreground and background colours. Instead, it 'inverts' the colour numbers of the pixels corresponding to the set bits, and leaves unchanged those corresponding to unset bits. Inverting the colour number means that it changes as follows:

colour number = colour number EOR maximum colour number

Under default conditions, in which the maximum colour number is three, the colours change as follows:

Colour	Inverted Colour
0	0 EOR 3 = 3
1	1 EOR 3 = 2
2	2 EOR 3 = 1
3	3 EOR 3 = 0

When using AREAFILL in the default mode to fill in a polygon, all the pixels inside the polygon are altered. Some change to the foreground colour, the rest to the background colour. Therefore, filling in a polygon with a pattern totally removes all trace of the original polygon contents. However, when using the invert mode, the pixels corresponding to unset bits of the pattern remain unchanged. This means that some of the original contents of the polygon remain visible. The following program shows how this feature can be used to create a multi-coloured pattern.

```
PALETTE 0,0,0,0
PALETTE 1,1,0,0
PALETTE 2,0,0,0
PALETTE 3,0,1,0
CLS
COLOR 1,3

DIM pat%(15)

mainloop:
FOR mode% = 0 TO 1
  AREA ( 40, 23)
  AREA ( 40,159)
  AREA (600,159)
  AREA (600, 23)
  FOR loop% = 0 TO 15
```

```
    READ pat%(loop%)
  NEXT
  PATTERN ,pat%
  AREAFILL mode%
NEXT

RedCards:
DATA &H0008, &H001C, &H003E, &H007F
DATA &H003E, &H001C, &H0008, &H0000
DATA &H3600, &H7F00, &H7F00, &H3E00
DATA &H3E00, &H1C00, &H0800, &H0000

BlackCards:
DATA &H0800, &H1C00, &H2A00, &H7F00
DATA &H2A00, &H0800, &H0800, &H0000
DATA &H0008, &H001C, &H003E, &H007F
DATA &H002A, &H0008, &H0008, &H0000
```

The program starts by setting up the palette, clearing the screen to change the background to black, and setting up new foreground and background colours which are one (red) and three (green) respectively. It then dimensions a 16 element integer array which will be used for holding the patterns. The main loop is performed twice. Both times round, a rectangular polygon is defined and a second loop is used to read in a pattern from the data statements at the bottom. This pattern is then made current by calling PATTERN. The bit patterns of the numbers used are displayed below to show the two different patterns which are used:

```
0000000000001000  0000100000000000
0000000000011100  0001110000000000
0000000000111110  0010101000000000
0000000001111111  0111111100000000
0000000000111110  0010101000000000
0000000000011100  0001000000000000
0000000000001000  0001000000000000
0000000000000000  0000000000000000
0011011000000000  0000000000001000
0111111100000000  0000000000011100
0111111100000000  0000000000111110
0011111000000000  0000000001111111
0011111000000000  0000000000101010
0001110000000000  0000000000001000
0000100000000000  0000000000001000
0000000000000000  0000000000000000
```

These represent the different suits on playing cards. The first pattern contains a diamond in the top right and a heart in the bottom left. The second has a club in the top left and spade in the bottom right. Note that care has been taken to make sure that the different symbols do not overlap.

Finally, AREAFILL is called to apply the pattern to the polygon. The first time it is called with an argument of 0 which is the default value. This means that current foreground and background colours are used. At this stage the window will contain a green rectangle with red hearts and diamonds in it. The second time, the argument is one so the invert mode is used. Therefore the only change made to the contents of the rectangle is to invert the pixels which correspond to the bits in the second pattern which are set. These pixels are all currently drawn in colour three. Therefore inverting them changes them to colour 0 which is black. Therefore the green rectangle ends up displaying the bits set in the first pattern (the hearts and diamonds) in red, and those in the second pattern (the clubs and spades) in black.

Line Patterns

We've seen that the second argument of PATTERN affects how polygons are coloured in when using AREAFILL. The first argument doesn't affect solid shapes: instead it alters the way in which lines and rectangles are drawn using the LINE command.

The first argument to pattern is a single 16 bit number. Each of the bits represents a pixel in the line. A few examples are given below:

Value	Line
&HCCCC	-- -- -- --
&H1111	- - - -
&HEAEA	--- - - --- - -

The following gives a brief demonstration:

```
RANDOMIZE TIMER

FOR y% = 20 TO 180 STEP 10
  PATTERN INT(RND*&H7FFF)
  LINE (40,y%) - (600,y%)
NEXT
```

Creating Screens and Windows

The display area of the Amiga can contain a number of 'windows' which appear within 'screens'. For example when you enter BASIC there are two windows, the Output window and Edit window which are contained within the Workbench screen.

A screen has different attributes associated with it which control its appearance. The first of these is the number of pixels it has per line. The higher the number of pixels, the smaller each pixel is and hence the higher the resolution. A screen can either contain 320 or 640 pixels on every line. Screens which contain 320 pixels horizontally are referred to as 'low resolution' and those which contain 640 pixels as 'high resolution'.

In addition, a screen can contain a maximum of either 256 or 512 lines (200 and 400 respectively on American machines). A 512 line display is obtained by a technique called 'interlacing'. The display on a TV or monitor screen is refreshed 50 times a second. When the screen is interlaced, two different images are held: one containing the odd rows and the other containing the even ones, and these are refreshed alternately. This has the effect of doubling the vertical resolution but it also makes the screen flicker. Many applications avoid using interlace because of this.

These two attributes combine to produce four different screen modes:

Mode	Description
1	320 pixels across and 256 lines high
2	640 pixels across and 256 lines high
3	320 pixels across and 512 lines high
4	640 pixels across and 512 lines high

Another important attribute which screens have is the maximum number of colours which can be displayed in them at any one time. This is determined by the 'depth' of the screen or number of 'bits per pixel'. There are five different values:

Depth / Bits per pixel	Number of colours
1	2
2	4
3	8
4	16
5	32

You can think of the bits associated with each pixel as switches. For example, if you have only one switch there are just two different states it can be in: either on or off. On represents one colour and off a second one. Therefore there is a choice of just two colours for each pixel. If you double the number of switches to two, then there are four states: on & on, on & off, off & on and off & off. This means that the number of different colours which can be represented increases to four. Similarly three switches doubles the number of states and hence colours to eight etc.

A screen can contain several windows. Each of these windows shares the attributes of the screen it is in, and so they all have the same resolution and can display the same number of colours. Therefore, to enable us to use more than the four colours that we have been limited to so far, we are going to have to create our own screen. Try the following program:

```
SCREEN 1,640,256,4,2
WINDOW 2,"My own output window",,31,1
```

```
xpos1% = 0
ypos1% = 0
xpos2% = 639
ypos2% = 249
FOR loop% = 1 TO 15
  xpos1% = xpos1% + 20
  ypos1% = ypos1% +  8
  xpos2% = xpos2% - 20
  ypos2% = ypos2% -  8
  LINE (xpos1%,ypos1%) - (xpos2%,ypos2%),loop%,bf
NEXT
```

The SCREEN statement creates a new screen. The first argument is the identity number which is associated with the screen. AmigaBASIC can handle up to four different screens at once, each of which has a different identity number: one, two, three, or four. This number is used by other commands to tell them which screen is being acted on.

We'll leave the second and third arguments for now and move onto the fourth one. This is the depth of the screen. It takes a number in the range 1-5 as listed in the table above. Our example uses the number four, which allows a maximum of 16 colours to be used.

The last argument is the mode. This is a number 1-4 which specifies the resolution and whether the screen is interlaced or not. The definitions of the different modes are given above. The value two means high resolution and non-interlaced. This is the mode we have been using previously.

Finally, we'll return to the second and third arguments. These are respectively the width and height of the screen in pixels. However, these values do not affect the screen resolution – this is determined purely by the last argument. Instead, they limit the area of the screen which can contain windows and hence text and graphics. In our example, the size of the screen has been set up to the maximum values allowed for the mode we have selected. Therefore we will be able to use the whole area of the display. If we had chosen smaller numbers instead, the screen would still have appeared to occupy the whole display but part of the area would have been inaccessible, so would always have remained empty.

The WINDOW statement creates an output window. The first argument is the identity number of the window. This can be any positive integer value. The AmigaBASIC Output window has the number one, therefore the program has chosen to use the number two so that it creates an additional window, rather than re-defining the existing one.

All the other arguments are optional. The second is a string which will be used as the title of the window. The third (which has been omitted in our example) defines the rectangular area of the window in the form:

(x1,y1) – (x2,y2)

Since we have omitted this argument, our window will be created at the default size for the screen, which is the full screen size.

The fourth argument determines the features which the window will contain. Each feature has a different value:

Value	Description
1	Sizing gadget provided to allow mouse control of window size.
2	Window can be moved about using Title Bar.
4	Back gadget provided to allow window to be moved to front/back.
8	Close gadget provided to allow window to be closed using mouse.
16	Window is redrawn if a window in front of it is moved.

The number used is the sum of all the values of the features required. In our case we have been greedy and asked for all of them (31=1+2+4+8+16).

The final argument is the identity number of the screen which is to contain the window. We have used the value 1 to add it to the screen which we have just defined. We could alternatively have used the value –1, which is the default, to add it to the Workbench screen. If we had done this, the window would have picked up the attributes of the Workbench screen and acted just like the default BASIC Output

window, giving us just four colours. Adding the window to our own screen, which has four bits per pixel, means that we are now ready to create the 16 colour display we were aiming at.

When the window is created, it is automatically selected as the current output window. Therefore the remainder of the program sends all its graphics output to this new window. This output consists of a number of rectangles, which show the default colours assigned to colour numbers 0 to 15.

Memory Usage

Creating windows is one of the easiest ways of using up the memory available in your machine. Compared to the earliest microcomputers, which came with just 1Kbyte of memory (1k for short), you may think that the 256, 512 or 1024k that you have is a vast amount. However, each window can potentially use up to 160k of that memory.

The actual amount depends on several things. One of these is whether or not you select feature 16 to specify that the contents of a window are to be redrawn if another window, which was in front of it, is moved. Doing so means that BASIC has to reserve enough space to keep a copy of the contents of the window. If you also specify that you want feature one as well, which allows the window size to be changed, then it has to reserve enough memory to allow a copy of the whole screen to be stored, since your window can potentially become that large.

The space required to hold the contents of a window of the maximum size depends on the screen resolution, interlace mode and depth. The least amount of space is used by windows in a low-resolution, non-interlace screen which can display just two colours. In this case, the whole screen display requires just 10k. Changing the resolution or turning on interlace or increasing the number of colours alters the memory needed, as shown in the table below:

Resolution	Colours	Memory required
320 x 256	2	10 Kbytes
320 x 512	2	20 Kbytes
640 x 256	2	20 Kbytes
640 x 512	2	40 Kbytes
320 x 256	4	20 Kbytes
320 x 512	4	40 Kbytes
640 x 256	4	40 Kbytes
640 x 512	4	80 Kbytes
320 x 256	8	30 Kbytes
320 x 512	8	60 Kbytes
640 x 256	8	60 Kbytes
640 x 512	8	120 Kbytes
320 x 256	16	40 Kbytes
320 x 512	16	80 Kbytes
640 x 256	16	80 Kbytes
640 x 512	16	160 Kbytes
320 x 256	32	50 Kbytes
320 x 512	32	100 Kbytes

Note that the full 32 colours are only available in the low resolution modes.

Using Multiple Windows

We mentioned above that, when an output window is created, it is automatically selected as the current output window. However, you can change the output window when you like. The only restriction is that you can have just one window selected at once. Therefore to make the same line of text appear in two windows you must PRINT it twice, swapping the output window in between the two PRINT commands. The following program demonstrates how a number of windows can be created and used apparently 'simultaneously'.

```
SCREEN 1,320,256,5,1
WINDOW 2,"Rectangles",( 20, 20) - (140,120),22,1
WINDOW 3,"Circles",   (180, 20) - (300,120),22,1
WINDOW 4,"Ellipses",  ( 20,140) - (140,240),22,1
WINDOW 5,"Triangles", (180,140) - (300,240),22,1
```

```
DIM pat%(1)
pat%(0) = &HFFFF
pat%(1) = &HFFFF
PATTERN,pat%

RANDOMIZE TIMER
FOR col% = 31 TO 1 STEP -1

  rectangles:
  WINDOW OUTPUT 2
  xpos1% = INT(RND*120)
  ypos1% = INT(RND*100)
  xpos2% = INT(RND*120)
  ypos2% = INT(RND*100)
  LINE (xpos1%,ypos1%) - (xpos2%,ypos2%),col%,bf

  circles:
  WINDOW OUTPUT 3
  xpos1% = INT(RND*120)
  ypos1% = INT(RND*100)
  rad%   = INT(RND*30)
  CIRCLE (xpos1%,ypos1%),rad%,col%,,,1.12
  PAINT  (xpos1%,ypos1%),col%

  ellipses:
  WINDOW OUTPUT 4
  xpos1% = INT(RND*120)
  ypos1% = INT(RND*60)
  rad%   = INT(RND*30)
  asp    = 1.12*RND*2
  CIRCLE (xpos1%,ypos1%),rad%,col%,,,asp
  PAINT  (xpos1%,ypos1%),col%

  triangles:
  WINDOW OUTPUT 5
  COLOR col%
  xpos1% = INT(RND*120)
  ypos1% = INT(RND*100)
  xpos2% = INT(RND*120)
  ypos2% = INT(RND*100)
  xpos3% = INT(RND*120)
  ypos3% = INT(RND*100)
  AREA (xpos1%,ypos1%)
  AREA (xpos2%,ypos2%)
  AREA (xpos3%,ypos3%)
  AREAFILL
NEXT
```

The program starts by creating a new screen of the maximum size in mode five. This is a low resolution screen and so allows us to use the full 32 colours (depth five). Four windows are created in this screen, each being 120 pixels by 100 pixels. They are placed initially in the four corners of the screen. Their feature value is 22 (16 + 4 + 2). This means that they will be redrawn when any window in front of them is moved; they can be moved to the front or back using the Back gadget or moved about the screen using the Title Bar. However, they cannot be closed and so do not have a Close gadget. In addition, the size of them cannot be changed, thus saving memory.

Next, the preparation is completed by defining a simple, solid pattern ready for use by the AREAFILL command and re-seeding the random number generator.

Then the main body of the program starts. This consists of a loop repeated 31 times. Within the loop, each of the windows is selected in turn and a different solid shape is drawn in it. The colour of the shape is determined by the loop variable. The loop variable starts at the highest colour number and decreases by one each time, in order to get the lowest numbered colours used last. These are the brightest, so they make the final picture more colourful.

The shapes drawn correspond with the titles of the windows: rectangles, circles, ellipses and triangles. Rectangles are produced using the LINE command with the 'bf' option. Solid circles and ellipses are created by using the CIRCLE command and then PAINTing the outline drawn. Note that, because the resolution of the screen is 320x200 pixels, the aspect ratio needed to obtain circles is double the value required when we were creating them on a 640x200 pixel screen. Finally, the triangles are plotted using AREA to mark the three corners, and then AREAFILL to create the shape defined.

Menus

Mice and menus are the 'in thing' in computing at the moment. Their introduction, just a few years ago, revolutionised user interfaces. Gone are the days when you had no option but to press magic combinations of keys and type mystic runes to get the computer to do something for

you. Now, you can just pull down a menu or point at the screen to select what you want to do.

You should by now have used the mouse and menus fairly extensively while using both AmigaBASIC and the Workbench. No doubt many of you will have found them to be invaluable. Now we are going to look at how you can create menus and handle the mouse in your own program, and so make life easier for the people running them.

The MENU command is used to create menus, add items to them and to enable/disable either the whole menu or individual items. This takes four arguments which are described below:

The first argument is the identity number of the menu. This can be any number between one and 10. AmigaBASIC uses one to four for its menus. You can replace these if you wish or add extra ones.

The second argument is the identity number of the menu item. A menu can contain 19 items: therefore each item must have a number in the range 1 - 19. The alternative value this can take is 0, which means that the command refers to the whole menu.

The third argument defines the state of the item selected (or whole menu if 0 was used). This can take any of the following values:

0	Disable the menu item
1	Enable the menu item
2	Enable the menu item and place a check mark by it

The final argument, which is optional, is the title string of the item or menu.

For example, run the following program:

```
MENU 5,0,1,"My menu"
MENU 5,1,1,"  Item 1"
MENU 5,2,2,"  Item 2"
MENU 5,3,0,"  Item 3"
WHILE 1 = 1
WEND
```

While the program is running, press the right mouse button and you should see an extra item on the menu bar called 'My menu'. Now

point at this and its list of contents will be displayed. There should be three: Item 1, Item 2 and Item 3. Item 3 is disabled so that it cannot be selected. Its title will be 'ghosted out' to indicate this. Items 1 and 2 are both enabled, in addition, Item 2 has had a check mark placed by it which should be visible as a tick. This check mark takes up two character positions, which is why you had to leave two spaces before the titles of the items.

Stop this program by selecting the Stop item from the Run menu or by pressing Amiga-fullstop. Normally, if you run a program which creates menus, these menus will disappear when the program ends. Because the program above did not finish, the new menu will still be there. However, you can return to the default AmigaBASIC Menu Bar by typing:

```
MENU RESET
```

in the Output window.

Now that you have a program to create a menu, you need to know how to make it respond when a menu item has been selected. Occurrences such as menu items being selected or mouse buttons being pressed are called 'events'. One of the best features of the Amiga is that it can watch out for these events happening (ie 'trap' them) as a background activity whilst getting on with its main task of running your program. All you have to do is tell BASIC that you are interested in knowing about a particular event and set up a subroutine to handle it, should it occur. You never have to call this subroutine yourself: BASIC will do it for you whenever it is necessary.

You can turn menu event trapping on by using MENU ON. This should follow the command ON MENU GOSUB, which tells BASIC how to deal with menu items being selected. ON MENU GOSUB takes the name of the subroutine you have written to handle menu events. Then, if a menu item is selected whilst your program is running, BASIC will execute the named subroutine.

In order to be able to act appropriately, the subroutine will need to find out which item the user selected. You can find this out by using

the MENU functions. MENU(0) returns the number of the last menu selected and MENU(1) the number of the last item selected.

The following program should help to illustrate this:

```
init:
PALETTE 0,0,0,0
PALETTE 1,1,1,1
PALETTE 2,1,0,0
PALETTE 3,0,0,1
col% = 1
CLS

menuinit:
MENU 5,0,1,"My menu"
MENU 5,1,2,"  White"
MENU 5,2,1,"  Red"
MENU 5,3,1,"  Blue"
MENU 5,4,1,"  Clear screen"
ON MENU GOSUB menuhandler
MENU ON

mainloop:
WHILE INKEY$ = ""
  xpos1% = RND*620
  ypos1% = RND*180
  xpos2% = RND*620
  ypos2% = RND*180
  LINE (xpos1%,ypos1%) - (xpos2%,ypos2%),col%,b
WEND
END

menuhandler:
  REM Check correct menu
  IF MENU(0) = 5 THEN
    REM If so then act according to item selected
    IF     MENU(1) = 1 THEN
      col% = 1
      MENU 5,1,2
      MENU 5,2,1
      MENU 5,3,1
    ELSEIF MENU(1) = 2 THEN
      col% = 2
      MENU 5,2,2
      MENU 5,1,1
      MENU 5,3,1
    ELSEIF MENU(1) = 3 THEN
      col% = 3
      MENU 5,3,2
```

```
      MENU 5,1,1
      MENU 5,2,1
    ELSEIF MENU(1) = 4 THEN
      CLS
    END IF
  END IF
RETURN
```

This starts by setting up the palette, clearing the screen and initialising the value of 'col%' to one. Then it creates a new menu with four items, sets up the subroutine to deal with menu events and turns menu trapping on. The main loop of program draws rectangles, in the colour determined by col%, until a key is pressed.

The final part of the program is the subroutine for handling menu selections. This checks initially to see that it is menu number five which has been selected. If so, then it goes on to check which item was chosen. For the first three items, it assigns an appropriate value to col%, thus selecting the colour subsequent rectangles will be plotted in. Then it places a checkmark by the item in the menu and clears any checkmarks from the other two. For the fourth item it clears the screen.

You can turn the trapping of menu events off for part of a program by using the MENU OFF and MENU STOP commands. MENU OFF turns the trapping off completely, so that any information about menu items selected, whilst it is off, is lost. MENU STOP still traps menu events but does not react to them. When menu trapping is turned on again, information about these events will be given in response to MENU(0) and MENU(1) function calls. To turn menu event trapping on again, use MENU ON.

Mice

Dealing with mouse input is very similar to handling menu selections. The subroutine for dealing with a mouse event (ie the left mouse button being pressed) is specified using ON MOUSE GOSUB. Information about what has happened can be obtained using calls of the function MOUSE with the appropriate arguments. Finally, mouse event trapping can be disabled using MOUSE OFF or MOUSE STOP, and re-enabled again using MOUSE ON.

The MOUSE function returns various different pieces of information, depending on the argument it is given. These are summarised in the table below:

Argument	Return
0	Left button position
1	Current x co-ordinate
2	Current y co-ordinate
3	Start x co-ordinate
4	Start y co-ordinate
5	End x co-ordinate
6	End y co-ordinate

MOUSE(0) gives the status of the left mouse button as follows:

0 Button is not pressed, and has not been pressed since MOUSE(0) was last called

n Button is not pressed, but has been pressed n times since MOUSE(0) was last called. MOUSE(3) – MOUSE(7) can be used to give the positions of the mouse at the time the button was pressed and released.

-n Mouse button is currently pressed after being clicked n times.

Calling this function sets up the information necessary for MOUSE(1) - MOUSE(6).

MOUSE(1) and MOUSE(2) give the position of the mouse at the time that MOUSE(0) was called.

MOUSE(3) and MOUSE(4) give the position of the mouse at the time the button was last pressed before MOUSE(0) was called.

If the mouse button was pressed when MOUSE(0) was last called, MOUSE(5) and MOUSE(6) give the position of the mouse at that time, thus acting like MOUSE(1) and MOUSE(2). If the mouse button wasn't pressed, MOUSE(5) and MOUSE(6) give the position of the cursor at the time the mouse button was released.

For example:

```
mouseinit:
```

```
ON MOUSE GOSUB mousehandler
MOUSE ON

mainloop:
WHILE 1 = 1
SOUND 523.25,18.2
SOUND 659.26,18.2
SOUND 783.99,18.2
SOUND 1046.50,18.2
WEND
END

mousehandler:
  WHILE MOUSE(0) < 0
  WEND
  LINE (MOUSE(3),MOUSE(4)) - (MOUSE(5),MOUSE(6))
RETURN
```

This program starts by initialising the mouse event handler and activating mouse event trapping. The body of the program repeatedly plays four notes. This is just to show you that the program is still being executed while any mouse events are dealt with.

When the mouse button is pressed, the mousehandler subroutine is called. This waits until the mouse button has been released and then draws a line from the position of the mouse when the button was pressed, to its position at the time the button was released.

Storing Graphic Images

You have now encountered all of the commands for creating graphics which AmigaBASIC provides. However, before you rush off to start creating masterpieces, there is one last thing you need to know – how to save a picture once you have created it. The first stage of this is done using GET. For example:

```
DIM rect%(626)
GET (120,90) - (200,140),rect%
```

This saves in the array rect%, the details of a rectangular area of the screen defined by the corners (120,90) and (200,140).

You can then place a copy of the image stored in the array onto the screen at any position using PUT, for example:

```
PUT (220,90),rect%
```

The amount of space, needed in the array, depends on three things: the width and height of the rectangle and the number of bits per pixel of the screen. These are combined to give the number of bytes of storage space, as follows:

bytes needed = height*2 * INT((width+15)/16) * bits per pixel + 6

Since an integer is two bytes long, the number of array elements required is half the number of bytes.

The extra six bytes, which appear at the end of the formula above, are placed in the array as the first three elements. These contain the values of the width, height and number of bits per pixel. These pieces of information are vital for PUT.

In our example the height is 51 (140– 90+1) and the width is 81 (200–120+1). Therefore applying the formula gives:

bytes needed = 51*2 * INT((81+15)/16) * 2 + 6
= 104 * 6 * 2 + 6
= 1254

Therefore we needed 627 integer elements.

Having turned the picture on the screen into elements of an array, we are now ready to store the picture onto disc. The following program shows how to do this:

```
DIM rect%(626)
DIM brick%(7)
DIM tile%(1)
DIM glass%(1)
DIM solid%(1)

patterns:
brick%(0) = &HFFFF : brick%(1) = &H8080
brick%(2) = &H8080 : brick%(3) = &H8080
brick%(4) = &HFFFF : brick%(5) = &H0808
brick%(6) = &H0808 : brick%(7) = &H0808
tile%(0)  = &HFFFF : tile%(1)  = &H5555
glass%(0) = &H5555 : glass%(1) = &HAAAA
solid%(0) = &HFFFF : solid%(1) = &HFFFF
```

```
house:
AREA(120,100) : AREA(120,140)
AREA(200,140) : AREA(200,100)
COLOR 1,3 : PATTERN ,brick% : AREAFILL

roof:
AREA(120,100) : AREA(130, 90)
AREA(190,90)  : AREA(200,100)
COLOR 2,3 : PATTERN ,tile% : AREAFILL

door:
AREA(152,140) : AREA(152,124)
AREA(168,124) : AREA(168,140)
COLOR 1 : PATTERN,solid% : AREAFILL

windows:
COLOR 1,2 : PATTERN,glass%
AREA(130,132) : AREA(130,124)
AREA(145,124) : AREA(145,132)
AREAFILL
AREA(175,132) : AREA(175,124)
AREA(190,124) : AREA(190,132)
AREAFILL
AREA(130,116) : AREA(130,108)
AREA(145,108) : AREA(145,116)
AREAFILL
AREA(175,116) : AREA(175,108)
AREA(190,108) : AREA(190,116)
AREAFILL

store:
GET(120,90) - (200,140),rect%

save.to.disc:
OPEN "house" FOR OUTPUT AS 1
FOR loop% = 0 TO 626
PRINT #1,MKI$(rect%(loop%));
NEXT
CLOSE 1
```

The program starts by dimensioning some arrays. The first will be used to hold details of part of the screen. The others are for patterns to be used to colour the picture. The next stage is to set up these patterns. Then a simple house is drawn in stages: first the brickwork, then the roof, then the door and finally four windows.

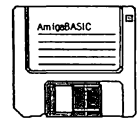
Then, the rectangular area which encloses the house is stored in the array 'rect%'. A file called 'house' is opened and each element of the array is converted into a 2-byte string and output to this file. When the complete contents of the array have been sent to the file, the file is closed. (More details about file handling can be found in Chapter 10.)

Now try the following program, which reads the contents of the file into an array and places four copies of the house on the screen:

```
DIM rect%(626)

load.from.disc:
OPEN "house" FOR INPUT AS 1
FOR loop% = 0 TO 626
rect%(loop%) = CVI(INPUT$(2,1))
NEXT
CLOSE 1

draw:
PUT(120,90),rect%
PUT(220,90),rect%
PUT(320,90),rect%
PUT(420,90),rect%
```

This program dimensions an array to hold a screen image and then performs the opposite sequence of events to the previous program. It starts by opening a file to read the data from. Then it reads each pair of bytes in turn, converts them into an integer and places this integer into the array. When all the array elements have been assigned to, it closes the file and places four copies of the contents of the array on the screen.

7 : Number Crunching



Number crunching is what computers are good at. They are orders of magnitude better at performing additions, multiplications etc than us mere humans. This has been exploited in many different ways, from finding prime numbers to analysing the stresses on different parts of a bridge.

This chapter looks at all aspects of numbers. It starts by examining the different types of numbers which AmigaBASIC supports and the operators which are available to act on them. Then it moves to demonstrate the use of arrays for solving geometrical problems and finally it deals with how to format numbers when outputting them.

Types of Numeric Variables

Numeric variables can be subdivided into four categories:

Short integers
Long integers
Single precision floating point numbers
Double precision floating point numbers

Integer variables can represent only whole numbers, for example 1, 107, -99 etc. The difference between short integer variables and long integer variables is the size of number they can hold. Short integer variables are restricted to values between -32768 and +32767. Long integer ones can hold values between -2147483648 and 2147483647. Because long integer variables have to be capable of holding larger values, they require more storage space and operations on them are slower.

Floating point variables are sometimes referred to as 'reals'. They can represent both whole numbers and decimal fractions, for example 1.5, 0.4589, etc. The difference between single precision variables and double precision variables is the size of number they can hold and the accuracy with which they hold the fractional part of the number. Single precision variables can hold values that lie between

approximately $2x10^{-38}$ and $3x10^{38}$ with up to seven digits of precision. Double precision variables are capable of holding numbers between $3x10^{-308}$ and $1x10^{308}$ with up to 16 digits of precision.

There are two ways of identifying a variable as being of a particular type. The first is to use a 'trailing declaration character'. This is a particular punctuation character which is placed at the end of the variable name. The characters allowed and the types they declare are as follows:

$	String
%	Short integer
&	Long integer
!	Single precision real
#	Double precision real

For example:

ypos%	is a short integer
ypos#	is a double precision real

The second method is to use definition statements. These declare all variables beginning with a particular letter or group of letters to be of a certain type. The definition statements available are as follows:

DEFSTR	String
DEFINT	Short integer
DEFLNG	Long integer
DEFSNG	Single precision
DEFDBL	Double precision

For example:

```
DEFINT n
```

means that any variable beginning with the letter 'n' will be treated as a short integer. Similarly:

```
DEFDBL a-d, x-z
```

means that any variable beginning with a letter in one of the given ranges ie 'a', 'b', 'c', 'd', 'x', 'y' or 'z' will be treated as a double precision floating point number.

There are two situations which have not yet been dealt with. The first is how to interpret variable names which begin with a letter not covered by a definition statement and have no trailing declaration character. These take the default type which is single precision. For example:

ypos is a single precision real

The second is how to interpret variable names whose first letter is covered by a definition statement declaring them to have one particular type and which have a trailing declaration character declaring them to be of a different type. In this case the trailing declaration character takes precedence. For example after a:

```
DEFINT y
```

statement, variable names will be interpreted as follows:

ypos is a short integer due to the definition statement

ypos# is a double precision real, the definition statement being overruled by the trailing #

Both methods have their advantages. The definition statements allow the types of a group of variables to be refined very easily. For example you may start out using single precision arithmetic for all calculations and find later that this is not accurate enough. If you are using only the definition statements, then you need only add a statement such as:

```
DEFDBL a - h , q - z
```

at the beginning of the program and the precision is changed throughout.

However, the trailing declaration characters make it clear which type each variable is. You can be certain that any variable ending with a '$' is going to be a string variable. There is no need to keep checking with the definition statements to find out the type. In addition they give

more flexibility for naming variables. Variables beginning with the same letter are not restricted to being of the same type.

Converting Between Different Numeric Types

There are several functions available which take a number and convert it to a particular type. These are:

CSNG	Converts to single precision
CDBL	Converts to double precision
CLNG	Converts to long integer
CINT	Converts to short integer
INT	Converts to short integer
FIX	Converts to short integer

The problem with converting a floating point number to an integer is how to deal with the fractional part of the number. The three routines, CINT, INT and FIX are provided to give a choice of how to treat it.

CINT rounds to the closest integer. Therefore if, the fractional part is greater than 0.5, the number is rounded up. If the fractional part is less than 0.5, the number is rounded down. If the fractional part is 0.5 exactly, then CINT rounds to the nearest even integer.

INT always rounds the number down

FIX removes the fractional part

The following table shows the difference between the three methods:

Number	Results from		
	CINT	INT	FIX
1.3	1	1	1
1.7	2	1	1
–1.3	–1	–2	–1
–1.7	–2	–2	–1

Conversion between types is performed for you automatically in most cases. When assigning a constant to a variable, the constant is stored as the type dictated by the variable name.

For example the assignments:

```
LET number! = 4/3
LET number% = 4/3
```

leave the variables with the following values:

number! is 1.333333
number% is 1

Hence the fractional part has been lost in the case of the integer variable.

When evaluating an expression, BASIC converts all the numbers so they are of the same type as the most precise operand. Any operations are performed to this precision. Then the result is converted to the type of the variable being assigned to. For example:

answer% = x! * y#

The expression contains a single precision real 'x!' and a double precision real 'y#'. When evaluating this, the following happens:

1) The value of 'x!' is converted so the operands are both double precision.
2) The multiplication is performed giving a result which has double precision accuracy.
3) This result is converted to a short integer so that it can be assigned to the variable answer%. This conversion is performed in the same manner as CINT, ie the number is rounded to the closest integer.

Numeric Expressions

Numeric expressions consist of a series of constants or variables which are acted upon by 'operators'. These operators fall into different categories as given below:

Arithmetic Operators

The most common kind of operators are 'arithmetic operators'. These produce a numerical result. For example:

(1.5 + 2) * (x%-130)

contains three arithmetic operators, '+', '*' and '–'.

The full list of arithmetic operators is given below:

Operator	Operation	Example
+	Unary plus	+x
–	Unary minus	–x
+	Addition	x + y
–	Subtraction	x – y
*	Multiplication	x * y
^	Raise to the power	x ^ y
MOD	Integer remainder	x MOD y
\	Integer division	x \ y

'MOD' and '\' are integer arithmetic operators. They act only on integer operands. Any floating point operands are rounded to the nearest integer before the operation is carried out. '\' returns the integer part of a division. MOD returns the integer remainder of a division. For example

```
13 \ 5      produces the result 2
13 MOD 5    produces the result 3
```

Relational Operators

Another type of operator that we have already met is called a 'relational operator'. This takes two operands and returns one of two alternate results: either true or false. For example:

```
b > 2
```

Relational operators tend to be used mainly as tests in IF or WHILE statements. The full list is given below:

Operator	Relation	Example
=	Equal to	x = y
<>	Not equal to	x <> y
<	Less than	x < y
>	Greater than	x > y
<=	Less than or equal to	x <= y
>=	Greater than or equal to	x >= y

Logical Operators

The final type of operator we are going to look at is the 'logical' or 'boolean' operator. This can be looked at in two ways. The simplest is to regard it as acting on operands which have the value true or false. It then produces true or false as its result. For example:

```
(a < 2) OR (a > 5)
```

is true if 'a' is less than two or greater than five and false otherwise.

The six logical operators are:

Operator	Returns true if
NOT	operand is false
EQV	operands are the same
AND	operands are both true
OR	either of the operands is true
XOR	either (but not both) of the operands is true
IMP	the first operand is false or both operands are true

The second way of looking at how these operators work is to regard the operands as a series of 'bits' or 'BInary digiTS'.

Normally, we deal in decimal numbers, ie numbers in base ten. Decimal numbers contain ten separate digits: zero to nine. These digits can be treated as being in columns, with a one in a column representing a power of ten, ie:

```
... 10000 1000 100 10 1
```

For example 123 represents 1*100 + 2*10 + 3*1.

Binary numbers are numbers in base two. They consist entirely of the digits '0' and '1'. A one in a particular column represents a power of two, ie:

```
... 128 64 32 16 8 4 2 1
```

For example the binary number 100101 = 1*32+0*16+0*8+1*4+0*2+1*1

This is equivalent to the decimal number 37.

The logical operators act on each of the bits of the operands and produce results as follows:

NOT If a bit of the operand is zero, a one is placed in the corresponding bit of the result.

EQV If the bits of the operands are both zero or both one, then a one is placed in the corresponding bit of the result.

AND If the bits of the operands are both one, then a one is placed in the corresponding bit of the result.

OR If either or both of the bits in the operands is one, then a one is placed in the corresponding bit of the result.

XOR If either but not both of the bits in the operands is one, then a one is placed in the corresponding bit of the result.

IMP If the bit in the first operand is zero, or the bit of the second operand is one, then a one is placed in the corresponding bit of the result.

For example, consider the two integers:

a% which has the value 53 (0000000000110101 in binary notation)
b% which has the value 12 (0000000000001100 in binary notation)

Applying the logical operators to these values give the following results:

Function	Decimal	Binary Representation
NOT a%	is –54	(1111111111001010)
NOT b%	is –13	(1111111111110011)
a% EQV b%	is –58	(1111111111000110)
a% AND b%	is 4	(0000000000000100)
a% OR b%	is 61	(0000000000111101)
a% XOR b%	is 57	(0000000000111001)
a% IMP b%	is –50	(1111111111001110)
b% IMP a%	is –9	(1111111111110111)

Note that the action of the operators provide the binary notation of the answer. However, the relationship between the binary notation of a number and its decimal value may not be immediately clear in the case of negative numbers. Negative numbers are those whose top bit contains a one. The way to find the size of the number is to invert all

the bits in the answer and to add one. For example, the first operation, NOT a% returned the bit pattern:

1111111111001010

When inverted this becomes:

0000000000110101

and when one is added it changes to:

0000000000110110

whose decimal equivalent is 54. Therefore NOT a% gives the result 54.

The logical operators actually always work by acting on the individual bits of the numbers. 'True' and 'false' are just two special 'numbers' which take the values –1 and 0 respectively. In binary notation –1 has all the bits set to one and 0 has all the bits cleared to zero. It just so happens that the operators, when acting on a combination of 0 and –1, will always give a result which is either 0 or –1. The following illustrates this:

a% has the value true (1111111111111111)
b% has the value false (0000000000000000)

NOT a%	is false	(0000000000000000)
NOT b%	is true	(1111111111111111)
a% EQV b%	is false	(0000000000000000)
a% AND b%	is false	(0000000000000000)
a% OR b%	is true	(1111111111111111)
a% XOR b%	is true	(1111111111111111)
a% IMP b%	is false	(0000000000000000)
b% IMP a%	is true	(1111111111111111)

Operator Priority

Each operator is assigned a 'priority' and, when an expression is being evaluated this priority determines the order in which the operators are executed. Priority one operators are those acted upon first, and priority twelve last.

Priority	Operator	Action
1	^	Raise to the power
2	–	Unary minus
	+	Unary plus
3	*	Multiplication
	/	Division
4	\	Integer division
5	MOD	Integer remainder
6	+	Addition
	–	Subtraction
7	=	Equal to
	<>	Not equal to
	<	Less than
	>	Greater than
	<=	Less than or equal to
	>=	Greater than or equal to
8	NOT	Logical and bitwise complement
9	AND	Logical and bitwise conjunction
10	OR	Logical and bitwise disjunction
	XOR	Logical and bitwise exclusive or
11	EQV	Logical and bitwise equivalence
12	IMP	Logical and bitwise implication

For example, '12+3*4^2' is evaluated as '12+(3*(4^2))' and so produces the result 60.

Operators with the same priority are executed left to right, as they appear in the expression. For example, '24/4/2' is evaluated as '(24/4)/2'.

Mathematical Functions

AmigaBASIC supports all the standard mathematical functions, which are grouped below:

Function	Result	Argument	Result range
SIN(x)	sine of x	angle in radians	-1 to +1
COS(x)	cosine of x	angle in radians	-1 to +1
TAN(x)	tangent of x	angle in radians	any real
ATN(x)	arc-tangent of x	any real	$-\pi/2$ to $+\pi/2$
LOG(x)	log to base e of x	any +ve real	-38 to +38
EXP(x)	exponential of x	any real < 89 for any +ve real single precision, or < 710 for double precision numbers	
SQR(x)	square root of x	any +ve real	any +ve real
ABS(x)	absolute value of x	any real	any +ve real

Advanced Use of Arrays

In a simple program which handles arrays, it is usual for subprograms to make direct access to the arrays by use of the SHARED command. However, this means that the subprogram cannot readily be extracted and used as part of a new, and perhaps more complicated, program. Fortunately, a more general mechanism is available which allows libraries of subprograms to be constructed in such a way that they can act upon arrays of any shape and size.

To achieve this, AmigaBASIC allows an entire array to be passed as a single parameter to a subprogram. In addition, BASIC provides two functions: LBOUND and UBOUND which return the lower and upper bounds, respectively, of the various subscripts of an array. These allow a subprogram to handle arrays of different sizes. For example:

```
OPTION BASE 1
DIM arr%(10,20)
PRINT LBOUND(arr%)
PRINT UBOUND(arr%)
PRINT LBOUND(arr%,2)
PRINT UBOUND(arr%,2)
```

will produce:

```
1
10
1
20
```

If LBOUND and UBOUND are passed only the name of the array, then they return the lower and upper bounds of the first subscript. To investigate other subscripts, these functions can take a second argument which is the number of the subscript in question.

Note that the lower bound will always be zero or one, since these are the only values which can be given to OPTION BASE.

The following subprograms give methods of performing 'matrix' arithmetic. Matrices are two dimensional arrays, where the first dimension gives the number of rows in the matrix and the second gives the number of columns. They are drawn in the following form:

$$\begin{pmatrix} 11 \\ 11 \\ 11 \end{pmatrix} \qquad \text{or:} \qquad (1\ 1\ 1\ 1\ 1)$$

(3x2) (1x5)

The diagrams above show a 3x2 matrix and a 1x5 matrix respectively.

Matrices can be added, subtracted and multiplied together according to certain rules.

Only matrices with the same number of rows and columns can be added or subtracted. The resulting matrix is also the same size. For each element in the resulting matrix, its value is given by adding or subtracting the pair of elements in the corresponding place in the matrices being acted upon. For example:

$$\begin{pmatrix} 2 & -4 \\ 1 & 5 \\ -6 & 3 \end{pmatrix} + \begin{pmatrix} 1 & -2 \\ -2 & 3 \\ 4 & 0 \end{pmatrix} = \begin{pmatrix} 3 & -6 \\ -1 & 8 \\ -2 & 3 \end{pmatrix}$$

(a1) (a2) (a3)

Therefore matrices can be added using the subprogram in the following program. Although the program deals with the example above, the subprogram will work with matrices of any size:

```
OPTION BASE 1
DIM a1%(3,2)
DIM a2%(3,2)
DIM a3%(3,2)


a1%(1,1) =  2 : a1%(1,2) = -4
a1%(2,1) =  1 : a1%(2,2) =  5
a1%(3,1) = -6 : a1%(3,2) =  3
a2%(1,1) =  1 : a2%(1,2) = -2
a2%(2,1) = -2 : a2%(2,2) =  3
a2%(3,1) =  4 : a2%(3,2) =  0

CALL add.array(a1%(),a2%(),a3%())
PRINT a3%(1,1) a3%(1,2)
PRINT a3%(2,1) a3%(2,2)
PRINT a3%(3,1) a3%(3,2)

add.array:
SUB add.array(a1%(),a2%(),a3%()) STATIC
lb%       = LBOUND(a1%)
rows.ub% = UBOUND(a1%)
cols.ub% = UBOUND(a2%,2)

FOR row% = lb% TO rows.ub%
  FOR col% = lb% TO cols.ub%
    a3%(row%,col%) = a1%(row%,col%) + a2%(row%,col%)
  NEXT
NEXT
END SUB
```

Note that this subprogram assumes that the lower bounds of all three matrices are the same. In addition, it does not check that the three matrices are the same size, ie that the upper bounds are equivalent for both dimensions.

Matrix multiplication is more complex. A matrix with 'n' rows and 'm' columns can only be multiplied by a matrix with 'm' rows and 'p'

columns. The result is a matrix with 'n' rows and 'p' columns. The element in the ith row and jth column of the resulting matrix is given by:

ai1.b1j + ai2.b2j + ... + aim.bmj

For example:

$$(1 \quad 2 \quad 3) \quad x \quad \begin{pmatrix} 0 & 1 \\ -1 & 0 \\ 2 & 1 \end{pmatrix} = (1x0+2x-1+3x2 \qquad 1x1+2x0+3x1)$$

$$= \qquad (4 \qquad\qquad\qquad 4)$$

Therefore a subprogram to perform matrix multiplication is as follows:

```
OPTION BASE 1
DIM a1%(1,3)
DIM a2%(3,2)
DIM a3%(1,2)


a1%(1,1) =  1 : a1%(1,2) =  2 : a1%(1,3) =  3
a2%(1,1) =  0 : a2%(1,2) =  1
a2%(2,1) = -1 : a2%(2,2) =  0
a2%(3,1) =  2 : a2%(3,2) =  1


CALL mult.array(a1%(),a2%(),a3%())
PRINT a3%(1,1) a3%(1,2)
SUB mult.array(a1%(),a2%(),a3%()) STATIC
lb%       = LBOUND(a1%)
rows1.ub% = UBOUND(a1%)
cols1.ub% = UBOUND(a1%,2)
cols2.ub% = UBOUND(a2%,2)

FOR row% = lb% TO rows1.ub%
  FOR col% = lb% TO cols2.ub%
    a3%(row%,col%) = 0
    FOR n% = lb% TO cols1.ub%
      a3%(row%,col%) = a3%(row%,col%)+

a1%(row%,n%)*a2%(n%,col%)
```

```
    NEXT
  NEXT
NEXT
END SUB
```

Again the subprogram assumes that the lower bounds of all three arrays are the same and that their dimensions are valid. These have been set up to be so in the main program.

Array Space

When you DIM an array, BASIC reserves the maximum amount of space the elements in the array may require. For example, an innocent looking array such as :

```
OPTION BASE 1
DIM col%(620,180)
```

tries to claim enough space for 116000 integers, each of which is two bytes long. Therefore this single array requires over 100k of your computer's memory! Therefore you should be careful to keep arrays to the minimum size. In addition, you should release the memory used by an array as soon as you have finished with it. If you are only intending to use an array within one particular subprogram, you should DIM the array at the start of the subprogram and then ERASE it again before the end. For example:

```
SUB xxx  STATIC
OPTION BASE 1
DIM temp%(10,10)
.................
ERASE temp%
END SUB
```

Note that the ERASE statement in the above example is vital if the subprogram is to be called more than once. Without it, the second time the subprogram was called, BASIC would try to DIM the array temp%, find that it already existed and so give the error 'Duplicate definition'.

Formatting Numbers on Output

If you wish to output a series of numbers, the statement to use is PRINT USING. This allows you to dictate the format that you wish numbers to appear in, by using special characters. The most useful of these are '#' which is used to indicate the position of a digit, '.' which inserts a decimal point and '+' which prefixes numbers with a plus or minus sign. For example:

```
FOR loop% = 1 TO 10
  READ n
  PRINT USING "+##.#";n
NEXT
DATA 28,13.5,-27.99,-16,89
DATA 25.4,0,23.34,3.14,191.4
```

produces:

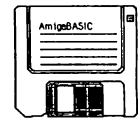
```
+28.0
+13.5
-28.0
-16.0
+89.0
+25.4
 +0.0
+23.3
 +3.1
%+191.4
```

Note the last number printed. This was too large to be fitted into the format supplied since it requires three digits in front of the decimal point. Therefore it has been preceded by a '%' character to indicate that it overflowed the field width. If you need to print numbers whose size you don't know, then it is safer to use exponential format. For example try the following:

```
FOR loop% = 1 TO 5
  INPUT "Please give me a number :",n(loop%)
NEXT
FOR loop% = 1 TO 5
  PRINT USING "+#.##^^^^";n(loop%)
```

```
NEXT
```

The '^^^^' is used to indicate exponential format. This is replaced by an 'E', a plus or minus and one or two digits of exponent.



8 : Sounds and Voices

We've already seen how to use BASIC to make the Amiga play a one-part tune. The next stage is to build on this to produce multi-part tunes. Then we will investigate how to make each of the parts produce a different type of sound, from soft melodious notes to harsh buzzing noises. In addition, we'll be taking a look at speech to investigate the many different personalities the Amiga can adopt.

Synchronisation

The Amiga supports four sound channels numbered '0', '1', '2' and '3'. When you play a note using the SOUND statement, this note is played on just one channel. By default this is channel 0. However you can choose one of the others by adding a fourth argument to SOUND. For example:

```
SOUND  523.25, 18.2, 127, 1
```

plays a middle C lasting for one second at the default volume on channel one. Sounds created on channels 0 and three are sent to the left speaker and sounds on channel one and two to the right speaker. You will only be able to tell the difference, though, if you link your Amiga up to a stereo system. Therefore when producing notes on the native machine all channels are equal.

If you use several SOUND commands, each is queued up until the previous one on its channel has finished. The obvious way to try to play two notes at once is to use two different channels. For example:

```
SOUND  523.25, 9.1, 127, 0
SOUND 1046.50, 9.1, 127, 1
```

There is a potential problem with this, in that the second note will be slightly behind the first because of the time it takes AmigaBASIC to analyse the second statement and act upon it. This delay will be

unnoticeable in the example given but, if BASIC had to do more work between the SOUND statements, it could become unacceptable.

Proper synchronisation of the channels is possible by using two statements: SOUND WAIT and SOUND RESUME. After a SOUND WAIT, all SOUND commands are stored rather than being played. The notes are only played after a SOUND RESUME command is given. Then, any notes on the same channel are played in order and notes on different channels are played simultaneously. For example:

```
SOUND WAIT
SOUND  523.25, 9.1, 127, 0
SOUND  659.26, 9.1, 127, 0
SOUND  783.99, 9.1, 127, 0
SOUND 1046.50, 9.1, 127, 0
SOUND RESUME
```

produces the four notes middle C, E, G and top C played one after the other producing an 'arpeggio' or 'broken chord'. This is because all of the notes are being played on channel 0, so they are played in sequence. Whereas:

```
SOUND WAIT
SOUND  523.25, 9.1, 127, 0
SOUND  659.26, 9.1, 127, 1
SOUND  783.99, 9.1, 127, 2
SOUND 1046.50, 9.1, 127, 3
SOUND RESUME
```

plays the same notes, but this time they all sound together producing a proper 'chord'. This is because they are on different channels so sound simultaneously in response to the SOUND RESUME.

We can use these statements to play a multi-part tune. The following program plays the same extract of the Can-can from Orpheus in the Underworld by Offenbach as the program we wrote in Chapter Two. However, this version plays three parts rather than just one.

```
REM Set up variables

dur      = 8.0
```

```
duration = 0
ending%  = 1

REM Produce the sounds
mainloop:
SOUND WAIT
WHILE channel% <> -4
  READ note$,duration,channel%
  IF channel% >= 0 THEN
    CALL note.value(note$,freq)
    SOUND freq,duration*dur,127,channel%
  ELSEIF channel% = -1 THEN
    SOUND RESUME
    SOUND WAIT
  ELSEIF channel% = -2 THEN
    SOUND RESUME
    SOUND WAIT
    IF ending% = 1 THEN
      RESTORE end1
    ELSE
      RESTORE end2
    END IF
    ending% = ending% + 1
  ELSEIF channel% = -3 THEN
    SOUND RESUME
    SOUND WAIT
    RESTORE main
  ELSEIF channel% = -4 THEN
    SOUND RESUME
  END IF
WEND

noteval:
SUB note.value(note$,freq) STATIC
IF     note$ = "a2n" THEN
  freq =  220.00
ELSEIF note$ = "b2n" THEN
  freq =  246.94
ELSEIF note$ = "c2n" THEN
  freq =  261.64
ELSEIF note$ = "d2n" THEN
  freq =  293.66
ELSEIF note$ = "e2n" THEN
```

```
    freq =  329.63
  ELSEIF note$ = "f2s" THEN
    freq =  370.00
  ELSEIF note$ = "g2n" THEN
    freq =  392.00
  ELSEIF note$ = "a3n" THEN
    freq =  440.00
  ELSEIF note$ = "b3n" THEN
    freq =  493.88
  ELSEIF note$ = "c3n" THEN
    freq =  523.25
  ELSEIF note$ = "d3n" THEN
    freq =  587.33
  ELSEIF note$ = "e3n" THEN
    freq =  659.26
  ELSEIF note$ = "f3s" THEN
    freq =  740.00
  ELSEIF note$ = "g3n" THEN
    freq =  783.99
  ELSEIF note$ = "a4n" THEN
    freq =  880.00
  ELSEIF note$ = "b4n" THEN
    freq =  993.00
  ELSEIF note$ = "c4n" THEN
    freq = 1046.50
  ELSEIF note$ = "d4n" THEN
    freq = 1174.70
  ELSEIF note$ = "e4n" THEN
    freq = 1318.50
  ELSEIF note$ = "f4s" THEN
    freq = 1480.00
  ELSEIF note$ = "g4n" THEN
    freq = 1568.00
  ELSE
    BEEP:STOP
  END IF
  END SUB

  REM The Data
  main:
  DATA g3n,2.0,0
  DATA g3n,0.5,1,d3n,0.5,1,g3n,0.5,1,d3n,0.5,1
  DATA g2n,0.5,2,b3n,0.5,2,d3n,0.5,2,b3n,0.5,2
```

```
DATA "",0.0,-1
DATA a4n,0.5,0,c4n,0.5,0,b4n,0.5,0,a4n,0.5,0
DATA a3n,0.5,1,c3n,0.5,1,b4n,0.5,1,c3n,0.5,1
DATA f2s,0.5,2,a3n,0.5,2,d2n,0.5,2,f2s,0.5,2
DATA "",0.0,-1
DATA d4n,1.0,0,d4n,1.0,0
DATA d4n,0.5,1,d3n,0.5,1,d4n,0.5,1,d3n,0.5,1
DATA g2n,0.5,2,b3n,0.5,2,d2n,0.5,2,b3n,0.5,2
DATA "",0.0,-1
DATA d4n,0.5,0,e4n,0.5,0,b4n,0.5,0,c4n,0.5,0
DATA d4n,0.5,1,b3n,0.5,1,b4n,0.5,1,b3n,0.5,1
DATA g2n,0.5,2,g2n,0.5,2,d2n,0.5,2,g2n,0.5,2
DATA "",0.0,-1
DATA a4n,1.0,0,a4n,1.0,0
DATA a4n,0.5,1,c3n,0.5,1,a4n,0.5,1,c3n,0.5,1
DATA f2s,0.5,2,a3n,0.5,2,d2n,0.5,2,f2s,0.5,2
DATA "",0.0,-1
DATA a4n,0.5,0,c4n,0.5,0,b4n,0.5,0,a4n,0.5,0
DATA a4n,0.5,1,c3n,0.5,1,b4n,0.5,1,c3n,0.5,1
DATA f2s,0.5,2,a3n,0.5,2,d2n,0.5,2,f2s,0.5,2
DATA "",0.0,-2
end1:
DATA g3n,0.5,0,g4n,0.5,0,f4s,0.5,0,e4n,0.5,0
DATA g3n,0.5,1,d3n,0.5,1,f4s,0.5,1,d3n,0.5,1
DATA g2n,0.5,2,b3n,0.5,2,d2n,0.5,2,b3n,0.5,2
DATA "",0.0,-1
DATA d4n,0.5,0,c4n,0.5,0,b4n,0.5,0,a4n,0.5,0
DATA d4n,0.5,1,c3n,0.5,1,b4n,0.5,1,c3n,0.5,1
DATA d3n,0.5,2,f3s,0.5,2,d3n,0.5,2,f3s,0.5,2
DATA "",0.0,-3

end2:
DATA g3n,0.5,0,d4n,0.5,0,a4n,0.5,0,b4n,0.5,0
DATA g3n,0.5,1,d3n,0.5,1,a4n,0.5,1,c3n,0.5,1
DATA g2n,0.5,2,b3n,0.5,2,d2n,0.5,2,f2s,0.5,2
DATA "",0.0,-1
DATA g3n,1.0,0,g4n,1.0,0
DATA b3n,1.0,1,b2n,1.0,1
DATA g2n,1.0,2,d2n,1.0,2
DATA "",0.0,-4
```

This program stores the details about the notes to be played in data statements. There are three statements for each bar, one for each of the different parts being played. The statements are divided up like this to help you keep track of what is happening. Each group of three statements is followed by a set of special data to mark the end of the bar. The program starts by issuing a SOUND WAIT statement. It then loops round, reading in the details of notes and queueing up SOUND statements for each of them, until it comes across the special data. This data causes the program to execute a SOUND RESUME statement to play the previous bar, followed by a SOUND WAIT to start the process off again for the next bar. Only one bar's worth of notes are queued at a time because the sound buffer is limited in size and if it gets full an error is generated.

The piece we are playing is made up of 16 bars of music. However, the first six bars are repeated twice. To avoid having to repeat the data statements for these, the special data has different channel values so that it can give instructions to the main program about which bars to perform next. These are interpreted as follows:

–1	end of an ordinary bar, do nothing special
–2	end of the main section, start either end1 or end2
–3	end of the first half, repeat main section
–4	end of the second half so stop

DATA statements can only hold constant numbers or strings; they cannot contain variables or expressions. This prevents us from using variables like 'b4n' to hold the values of the notes. Since having a descriptive form for each note, rather than just a number, is so useful, the program has found a solution to the problem. It represents each note as a constant string such as "b4n". When the string is read, it is passed to a subprogram 'note.value'. This subprogram uses a large IF statement to match it against each of the notes it knows, and places the frequency for it in a second parameter whose value is returned to the main program.

Waves

All sounds are made up of waves. A pure sound would consist of a perfect sine wave like the one shown in Figure 8.1:

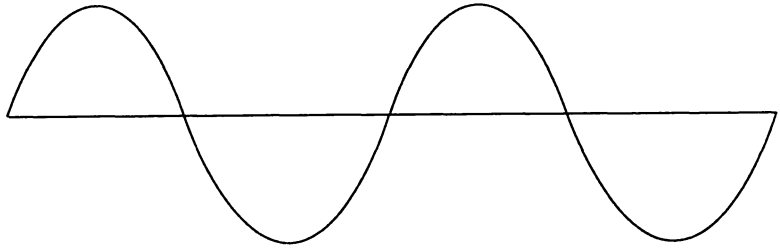


Figure 8.1. A pure sound wave.

The frequency of the wave dictates its pitch and the 'amplitude' or height dictates its volume. Unfortunately no instruments are capable of producing pure sounds. Instead they produce a mixture of waves which combine to form complicated waveforms.

The waveform is important in determining the characteristics of the sound produced. Each instrument has a different waveform associated with it, so to imitate an instrument we have to try to replicate its waveform. Normally, AmigaBASIC uses the sine wave, but you can override this using the WAVE statement. This requires two arguments. The first is simply the channel which you want to assign the waveform to. The second is either the word 'SIN' to select the default wave form, or an array containing at least 256 integer elements. Each of these represents a point of the waveform. The maximum value of an element is 127, which means that the waveform is at the maximum height above the middle line. The minimum value is –128, which means that the waveform is at the greatest distance below the line.

Obtaining the right waveform is a process of trial and error. The following program attempts to let you see and hear different waveforms in action. It allows you to select one of 10 available waveforms by pressing a key one to zero. This waveform will be illustrated graphically to show its shape. Then if you press certain keys on the keyboard the program will play a note using the current waveform. These keys are as follows:

Key	A	S	D	F	G	H	J	K
Note	C	D	E	F	G	A	B	C

You can continue this process over and over again. The program only ends when a non-special key is pressed.

```
array.initialisation:
OPTION BASE 0
DIM wf%(255)

variables:
pi  = 3.14159
con = 2*pi/256    : REM converts 0 - 256 into 0 - 2*PI
dur = 9.1         : REM Duration of each note

screen.colours:
PALETTE 0,0,0,0   : REM Black background
PALETTE 1,1,1,1   : REM White axes
PALETTE 2,0,1,0   : REM Green graph
CLS               : REM Clear the screen

main:
WHILE 1 = 1
  key$ = INKEY$
  IF key$ <> "" THEN
    key$ = UCASE$(key$)

numbers:
    IF      key$ = "1" THEN
      CALL wave1 : WAVE 0,wf% : CALL draw
    ELSEIF key$ = "2" THEN
      CALL wave2 : WAVE 0,wf% : CALL draw
```

```
    ELSEIF key$ = "3" THEN
      CALL wave3 : WAVE 0,wf% : CALL draw
    ELSEIF key$ = "4" THEN
      CALL wave4 : WAVE 0,wf% : CALL draw
    ELSEIF key$ = "5" THEN
      CALL wave5 : WAVE 0,wf% : CALL draw
    ELSEIF key$ = "6" THEN
      CALL wave6 : WAVE 0,wf% : CALL draw
    ELSEIF key$ = "7" THEN
      CALL wave7 : WAVE 0,wf% : CALL draw
    ELSEIF key$ = "8" THEN
      CALL wave8 : WAVE 0,wf% : CALL draw
    ELSEIF key$ = "9" THEN
      CALL wave9 : WAVE 0,wf% : CALL draw
    ELSEIF key$ = "0" THEN
      CALL wave0 : WAVE 0,wf% : CALL draw

letters:
    ELSEIF key$ = "A" THEN
      SOUND  523.25,dur,127,0
    ELSEIF key$ = "S" THEN
      SOUND  587.33,dur,127,0
    ELSEIF key$ = "D" THEN
      SOUND  659.26,dur,127,0
    ELSEIF key$ = "F" THEN
      SOUND  701.00,dur,127,0
    ELSEIF key$ = "G" THEN
      SOUND  783.99,dur,127,0
    ELSEIF key$ = "H" THEN
      SOUND  880.00,dur,127,0
    ELSEIF key$ = "J" THEN
      SOUND  993.00,dur,127,0
    ELSEIF key$ = "K" THEN
      SOUND 1046.50,dur,127,0

 illegal.key:
    ELSE
      END
    END IF
  END IF
WEND

draw:
```

```
SUBdraw STATIC
SHARED wf%()
CLS
LINE (50,100) - STEP(255*2, 0)
LINE (50,100) - STEP(0    , 128/2)
LINE (50,100) - STEP(0    ,-128/2)
PSET (50,wf%(0)/2),2
FOR loop% = 1 TO 255
  LINE STEP(0,0) - (50+loop%*2,100-wf%(loop%)/2),2
NEXT
END SUB

wave1:
SUB wave1 STATIC
SHARED wf%(), con
FOR loop% = 0 TO 255
  wf%(loop%) = 127*SIN(loop%*con)
NEXT
END SUB

wave2:
SUB wave2 STATIC
SHARED wf%(), con
FOR loop% = 0 TO 255
  wf%(loop%) = 63*(SIN(loop%*con) + COS(loop%*con*2))
NEXT
END SUB

wave3:
SUB wave3 STATIC
SHARED wf%(), con
FOR loop% = 0 TO 255
  wf%(loop%) = 40*(2*SIN(loop%*con)*COS(loop%*con*6))
NEXT
END SUB

wave4:
SUB wave4 STATIC
SHARED wf%(), con
FOR loop% = 0 TO 255
  IF loop% < 128 THEN
    wf%(loop%) = 127
  ELSE
```

```
    wf%(loop%) = -127
  END IF
NEXT
END SUB

wave5:
SUB wave5 STATIC
SHARED wf%(), con
FOR loop% = 0 TO 255
  IF loop% < 128 THEN
    wf%(loop%) = 127 - 2*loop%
  ELSE
    wf%(loop%) = 2*(loop%-128) - 127
  END IF
NEXT
END SUB

wave6:
SUB wave6 STATIC
SHARED wf%(), con
FOR loop% = 0 TO 255
  wf%(loop%) = 127*SIN(loop%*con*5)*EXP(-loop%/50)
NEXT
END SUB

wave7:
SUB wave7 STATIC
SHARED wf%(), con
FOR loop% = 0 TO 255
  wf%(loop%) = 127*EXP(-loop%/40)
NEXT
END SUB

wave8:
SUB wave8 STATIC
SHARED wf%(), con
FOR loop% = 0 TO 255
  wf%(loop%) =
40*(SIN(loop%*con)+SIN(loop%*con*2)+SIN(loop%*con*4)
NEXT
END SUB

wave9:
```

```
SUB wave9 STATIC
SHARED wf%(), con
FOR loop% = 0 TO 255
  wf%(loop%) = 63*(SIN(loop%*con)+SIN(loop%*con*12)
NEXT
END SUB

wave0:
SUB wave0 STATIC
SHARED wf%(), con
FOR loop% = 0 TO 255
  wf%(loop%) = RND*255 - 127
NEXT
END SUB
```

The main loop is a WHILE loop whose conditional expression is '1 = 1'. This always gives the result TRUE so the loop never ends. Inside this loop is a call of INKEY$ to test the keyboard to see if a key has been pressed. If so, the string returned is converted to upper-case by UCASE$. This means, for example, that we don't have to test the string against both "A" and "a": just one test is sufficient. Then the string is compared with all the keys we are interested in. If it is a number, then the program calls a subprogram to set up the values for the waveform in the array wf%, assigns this waveform to channel 0 and calls the subprogram 'draw' to display the shape of the waveform on the screen. If it is a letter corresponding to a note, the program plays the relevant note for half a second using the current waveform. Otherwise the key is invalid and the program terminates.

The subprogram 'draw' clears the screen, draws the axes and plots the point corresponding to the first element of wf%. It then loops round the rest of the points, linking each to the previous one with a straight line.

Subprograms 'wave1' to 'wave0' perform the task of assigning values to the array wf% which is shared with the main program. To alter a waveform all you have to do is to change the action of one of these subprograms. Note that you must make sure that the each value in the array is in the range –128 to 127, otherwise an:

```
Illegal function call
```

error message is generated by the WAVE statement.

Speech

All that you need to make the computer speak are the two keywords TRANSLATE$ and SAY. For example:

```
FOR loop% = 1 TO 5
  PRINT
  PRINT "Please give me a sentence "
  INPUT ":",sen$
  sen$ = TRANSLATE$(sen$)
  SAY sen$
NEXT
```

Note that, when you run this program, you will be prompted to place the Workbench disc in a disc drive if it is not in one currently. This is because the software BASIC requires to perform TRANSLATE$ and SAY is supplied as part of the Workbench. Since these routines are fairly large, BASIC doesn't keep them in memory. If they are present when a program is run, then BASIC deletes them before starting to execute the program. This means that, if you run this program several times, the routines will be reloaded into memory each time.

TRANSLATE$ converts a text string into 'phonemes' or 'units of significant sound'. Roughly speaking, (no pun intended), TRANSLATE$ takes a string of characters and converts them into a string of sounds. SAY then takes this string of sounds and 'speaks' it.

Unfortunately TRANSLATE$ is not perfect. English is an extremely difficult language to handle in this way. Consider the following words: cough, bough, through; in each the string 'ough' is pronounced in a different way. TRANSLATE$ has a set of rules to work to about how different combinations of letters should be spoken. Unless a computer has vast amounts of memory to spare, teaching it all the special cases is not possible. Even if it were, having a phonetic version of every word would not be enough. For example 'I am going to read that book' and 'I have read that book' both contain the word 'read' but the context it is in affects its pronunciation.

Phonemes

If words aren't spoken as you would like them to be, then the easiest way to solve the problem is to cheat. For example, if you want the word 'data' to be pronounced the English way rather than the American, then spell it 'dayter'. A much harder, but more precise, way is to construct the phonemes yourself rather than letting TRANSLATE$ do it for you. This is a very complex area of AmigaBASIC which could take up a whole book in itself. The following provides only a brief introduction, which should allow you to experiment for yourself if you wish.

Each phoneme is represented by one or two upper-case characters. These are listed in categories below, together with words which give examples of their sound.

Vowels

AA	cot
AE	cat
AH	fun
AO	walk
AX	about
EH	let
ER	bird
IH	fit
IX	solid
IY	feet
OH	cord
UH	book

The vowels are the continuous sounds which connect the shorter consonants together.

At first sight, AE seems similar to AX and IH similar to IX. The difference is that AX and IX are used for vowels which tend to get buried in the surrounding sounds due to the lazy way in which we speak. If we pronounced all words clearly and correctly there would hardly be any use for these phonemes.

Any of the vowels other than AX and IX can be followed by a digit one to nine. The existence of a digit causes the syllable to be lengthened to place stress on it. The value of the digit indicates the intonation to be used: a higher value creates a greater rise in pitch at that point.

Diphthongs:

AW	cow
AY	tide
EY	fade
OW	row
OY	foil
UW	flew

Diphthongs are like vowels except that they change their sound as they are spoken. They too can be emphasised.

Consonants:

B	bat	K	cot	T	tap
/C	block	L	lot	TH	this
CH	chat	M	mat	V	very
D	day	N	next	W	wide
DH	the	NX	king	Y	yacht
F	fat	P	pat	Z	zoo
G	get	R	rat	ZH	leisure
/H	hot	S	sat		
J	jam	SH	show		

Punctuation

Punctuation marks can be used to help the intonation of sentences. Sentences should be ended with either a '.' or a '?' to determine the final rise or fall of the voice. In addition a '–' or ',' can be used in the middle of a sentence to cause a short pause; using the ',' also causes a slight rise in the voice to indicate that the sentence has not yet finished. The final punctuation characters are '(' and ')'. These can be put round phrases to group the words together for intonation purposes.

To try out all this theory, adapt the program above to read as follows:

```
FOR loop% = 1 TO 5
```

```
    PRINT
    PRINT "Please give me a phonetic sentence "
    INPUT;":",sen$
    SAY sen$
NEXT
```

Now you have to input the phonetic string which is fed directly into SAY. Try the following for starters:

```
AY LAYK UW3ZIXNX AEMIY5GAE BEY3SIXK.
```

If you need help at any stage, have a look at what TRANSLATE$ produces and try editing its output.

Altering the Voice

Besides deciding what the Amiga says to you, you can also choose how it says it. The SAY statement takes a second argument which is an integer array containing at least nine elements. The first nine elements in it are used to control various characteristics of the voice which the computer uses:

0 – Pitch
This is the base pitch or frequency of the voice. The value should be in the range 65 to 320, which allows voices ranging from very deep to high and squeaky to be used. The default value is 110, which is a typical male voice, for a female the value should be higher.

1 – Inflection
You can either have inflections in the speech or not. Zero, the default value, causes the voice to have modulations in it like normal speech. The value one produces a monotonic, artificial sounding voice.

2 – Rate
The speaking rate is specified in words per minute. Permitted values are between 40 and 400, with the default value being 150. For anyone wanting to check how good their shorthand is, this is the ideal tool!

3 – Voice
Either a male or female voice can be used by providing the value zero (which is the default) or one.

4 – Tuning
This determines the quality of the sound produced by altering the 'sampling frequency'. The permitted values are in the range 5000 to 28000, and these generate voices which are low and vibrant to high and squeaky. The default value is 22200. Again, this should be raised for a female voice.

5 – Volume
Values between 0 (silent) and 64 (loud) are allowed. The default is 64.

6 – Channel
Selecting a channel is not quite as simple as with the SOUND command. This is because voices can be sent to more than one channel at once. Therefore, if you connect your Amiga to a stereo system, you can have the voice coming from either the left or right speaker or both. The combinations which are possible are:

0 Channel 0
1 Channel 1
2 Channel 2
3 Channel 3
4 Channels 0 and 1
5 Channels 0 and 2
6 Channels 1 and 3
7 Channels 2 and 3
8 Any available left channel (0 and/or 3)
9 Any available right channel (1 and/or 2)
10 Either available pair of channels (0 & 3 or 1 & 2)
11 Any available single channel

A channel is 'available' when there is no sound currently being produced on it. Therefore, if your program is using channels zero and one to play a tune, it has channels two and three available for speech.

The default value is 10, which selects any available pair of channels.

7 – Mode
The mode determines how BASIC handles the other commands in the program following the SAY command. A value of zero makes it wait until SAY has finished outputting the entire phonetic message it was given, before starting on the next command. This is the default mode.

The other value allowed is one, which causes it to continue executing the other commands while the speech is being produced.

8 – Control

If a value of one is given to the mode option, then the control option is used to determine how multiple SAY statements are dealt with:

0 Start the second when the first has finished
1 Stop all speech processing
2 Terminate the first and start the second immediately

The following program lets you try out the various voice options:

```
init:
PALETTE 0,0,0,0
PALETTE 1,1,1,1
CLS

menuinit:
MENU 5,0,1,"Speech"
MENU 5,1,1,"Speak"
MENU 5,2,1,"Quit"
MENU 5,3,2,"  Male"
MENU 5,4,1,"  Female"
MENU 5,5,2,"  Inflections"
MENU 5,6,1,"  Monotone"
ON MENU GOSUB menuhandler
MENU ON

mouseinit:
ON MOUSE GOSUB mousehandler
MOUSE ON

speechinit:
DIM mode%(8)
mode%(0) = 110 : mode%(1) =     0 : mode%(2) = 150
mode%(3) =   0 : mode%(4) = 22200 : mode%(5) =  64
mode%(6) =  10 : mode%(7) =     0 : mode%(8) =   0
DIM min%(8), max%(8), row%(8)
min%(0) =   65 : max%(0) =   320 : row%(0) =  2
min%(2) =   40 : max%(2) =   400 : row%(2) =  8
min%(4) = 5000 : max%(4) = 28000 : row%(4) =  5
```

```
min%(5) =    0 : max%(5) =    64 : row%(5) = 11
sentence$ = "The quick brown fox jumps over the lazy
dog"
sen$ = TRANSLATE$(sentence$)

screeninit:
LOCATE  3,1 : PRINT "Pitch"
LOCATE  6,1 : PRINT "Tuning"
LOCATE  9,1 : PRINT "Rate"
LOCATE 12,1 : PRINT "Volume"
LOCATE 15,1 : PRINT "Settings"
cy% = 9 : minb% = 100 : maxb% = 600
LINE (minb%-1, 2*cy%-1) - (maxb%+1, 3*cy%+1),1,b
LINE (minb%-1, 5*cy%-1) - (maxb%+1, 6*cy%+1),1,b
LINE (minb%-1, 8*cy%-1) - (maxb%+1, 9*cy%+1),1,b
LINE (minb%-1,11*cy%-1) - (maxb%+1,12*cy%+1),1,b
FOR loop% = 0 TO 8
  CALL update(loop%)
NEXT
mainloop:
WHILE 1 = 1
WEND

update:
SUB update(entry%) STATIC
  SHARED min%(),max%(),mode%(),row%(),cy%,minb%.maxb%
  IF row%(entry%) <> 0 THEN
    ypos1% = row%(entry%)*cy%
      ypos2% = (row%(entry%)+1)*cy%
    range% = max%(entry%) - min%(entry%)
    fract  = (mode%(entry%) - min%(entry%))/range%
    xpos2% = minb% + fract*500
    LINE (minb%,ypos1%) - (maxb%, ypos2%),0,bf
    LINE (minb%,ypos1%) - (xpos2%,ypos2%),1,bf
  END IF
  LOCATE 15,10 : PRINT TAB(10+6*entry%);SPC(6)
  LOCATE 15,10 : PRINT TAB(10+6*entry%);mode%(entry%)
END SUB

menuhandler:
  IF MENU(0) = 5 THEN
    IF     MENU(1) = 1 THEN
      SAY sen$, mode%
```

```
    ELSEIF MENU(1) = 2 THEN
      END
    ELSEIF MENU(1) = 3 THEN
      mode%(3) = 0
      MENU 5,3,2
      MENU 5,4,1
      CALL update(3)
    ELSEIF MENU(1) = 4 THEN
      mode%(3) = 1
      MENU 5,4,2
      MENU 5,3,1
      CALL update(3)
    ELSEIF MENU(1) = 5 THEN
      mode%(1) = 0
      MENU 5,5,2
      MENU 5,6,1
      CALL update(1)
    ELSEIF MENU(1) = 6 THEN
      mode%(1) = 1
      MENU 5,6,2
      MENU 5,5,1
      CALL update(1)
    END IF
  END IF
RETURN
mousehandler:
  check% = MOUSE(0)
  xpos% = MOUSE(3) : ypos% = MOUSE(4)
  entry% = -1
  IF xpos% >= minb% AND xpos% <= maxb% THEN
    IF     ypos% >  2*cy% AND ypos% <  3*cy% THEN
      entry% = 0
    ELSEIF ypos% >  5*cy% AND ypos% <  6*cy% THEN
      entry% = 4
    ELSEIF ypos% >  8*cy% AND ypos% <  9*cy% THEN
      entry% = 2
    ELSEIF ypos% > 11*cy% AND ypos% < 12*cy% THEN
      entry% = 5
    END IF

    IF entry% <> -1 THEN
```

```
        range% = max%(entry%) - min%(entry%)
        fract  = (xpos% - minb%)/(maxb% - minb%)
        mode%(entry%) = min%(entry%) + fract*range%
        CALL update(entry%)
      END IF
    END IF
  RETURN
```

The program allows you to try altering the first six of the voice characteristics. The screen displays four bars representing the current level of pitch, rate, tuning and volume. A zero length bar represents the minimum value for the characteristic. One occupying the full length represents the maximum possible value. To change any of these, simply press the mouse button, at the appropriate distance along, within the bar limits. The bar will be redrawn at its new length and the corresponding value of the argument to be given to the SAY command will be updated at the bottom of the screen.

The voice and inflection options are provided by means of a menu. To alter one of these, press the menu button and select an item from the 'Speech' menu in the usual manner. The currently selected values are marked.

The menu also provides two other items. Selecting the first of these will allow you to hear your current voice settings in action saying 'The quick brown fox jumps over the lazy dog'. The second ends the program.

The program works as follows:

init:
The program starts by setting up the palette and clearing the screen.

menuinit:
Next, it sets up an extra menu containing six items. Items three and four form a pair of options, as do items five and six. The first of each pair is initially selected and has a checkmark placed by it. Note that the titles of these items are indented to allow room for the checkmark. Then a menu event handler is set up and menu events are enabled.

mouseinit:
Similarly, a mouse event handler is set up and mouse events are enabled.

speechinit:
This block initialises many of the details describing the arguments for the SAY command:

mode%()	holds the current setting of the nine different voice characteristics which will be passed to SAY.
min%()	holds the minimum values allowed for those characteristics which can take a range of values
max%()	holds the maximum values allowed for those characteristics which can take a range of values
row%()	holds the values of the text rows on which the bars representing the elements in row%() are to be placed

Each of the elements of mode% is initialised to its default value. Only the elements of min%, max% and row% which correspond to the characteristics taking a range of possible values are assigned to. This is because these are the only ones represented by bars and so details of the other elements are not required. The other elements will be left as zero (which the DIM statement initialised them to). This value indicates that they don't correspond to a bar.

Finally, the sentence to be spoken is defined and translated into phonemes:

say$ = list of phonemes to be spoken

screeninit:
The next action is to initialise the screen display. It sets up the following variables:

cy%	= height of a character in pixels
minb%	= left-hand edge of the bars in pixels
maxb%	= maximum right-hand edge of the bars in pixels

The titles of the options are printed starting in the left-hand column, at the appropriate distances down the screen. Then the limits of the bars are outlined to the right of them. The horizontal limits of the bar are

given in pixels by minb% and maxb%. The bottom of each bar is given by the number of text rows it is to be placed at down the screen, multiplied by cy%. Each bar is one row high. Note that the outline is drawn so that it lies one pixel outside the bar limits in each direction, so that the bar fits inside it.

The bars themselves are then drawn and the current settings printed by calling the subprogram 'update' for each characteristic in turn.

mainloop:
The main body of the program consists of a loop which continues forever.

update:
The subprogram 'update' has one parameter 'entry%' which is the entry in the array 'mode%' of the item to be updated.

If the row number of the entry is non-zero, then this means that it has a bar associated with it which needs updating. The bottom and top of the bar are initialised in ypos1% and ypos2% using the row number multiplied by cy%. Then the next few lines calculate the right-hand end of the bar. The current setting is converted into a fractional value of the possible range using :

$$\text{fraction} = \frac{\text{current value - minimum value}}{\text{maximum value - minimum value}}$$

Then the length of the bar is obtained by multiplying this fraction by the maximum possible length of the bar. Finally, the position of the right-hand end of the bar is calculated by adding the length onto the left-hand position (which is always minb%).

Two solid rectangles are then drawn. The first is one of the maximum possible length which is drawn in the background colour to erase the previous bar. Then one of the length calculated is drawn to display the new bar.

The subprogram ends by updating the display of the value of the array entry shown at the bottom of the screen. It positions the text cursor on column 10 of row 15 and then moves along by the appropriate number of characters until it reaches the start of the slot

containing the value. The first time it does this, it then prints six spaces to erase the current text. The second time, it prints the current value of the entry held in mode%.

menuhandler:
The menu handler starts by checking that the correct menu has been selected. If so, it checks which item of the menu has been chosen. For item one, 'Speak', it executes the SAY command using the current values in mode% to define the voice characteristics. For item two, 'Quit', it simply ends the execution of the program. Items three to six are similar to each other. These represent options for the voice. In each case, the appropriate element of mode% is assigned a value, the option selected is checkmarked, and the opposite option is redrawn without a checkmark to show that it is no longer selected. Finally, the subprogram 'update' is called to update the screen display of the current settings.

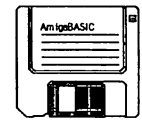
mousehandler:
The mouse handler starts by calling MOUSE(0) to set up the values for the other MOUSE calls. Then MOUSE(3) and MOUSE(4) are used to find the x- and y- co-ordinates of the mouse when the button was pressed. This subroutine then checks to make sure that the x position was between the minimum and maximum extents of the bars. Next it checks to see if the y position lies between the top and bottom of a bar. If so, entry% is set to be the number of the element in the array mode% which corresponds to the bar hit. Provided that entry% has been assigned to, a fraction is calculated giving the proportion of the distance along the bar at which the hit was made:

$$\text{fraction} = \frac{\text{pos of hit - pos of left-hand end of bar}}{\text{pos of right-hand end of bar - pos of left hand end of bar}}$$

Then the value of the entry is calculated by multiplying this fraction by the range of values allowed and adding on the minimum value it can take:

setting = min value + fraction * (max value – min value)

This result is placed in mode%(entry%), and the subprogram 'update' is called to update both the graphical and textual screen display.



9 : Animation

AmigaBASIC provides commands for controlling things called 'objects'. These are graphic shapes which can be placed on the screen and told which direction to move in. They will then continue going in that direction until they hit something, or you instruct them to do something different. One of the best features about them is that they do not affect the background. Many other computers do not have this capability and, without it, tasks which are very simple on the Amiga become very difficult.

For example, on older computers, a shape could only be moved by rubbing out its image at the old position and plotting it again at the new one. The problem was that rubbing out the old image left a blank space on the screen, instead of the scenery which should have been there in the background. To get around this, pieces of the background had to be saved before the shape was placed on top of them, and then reinstated again later. All this took time and meant that smooth animation was not possible in BASIC. Instead, programs had to be written in the machine's native machine code language which is very primitive and difficult to understand.

Bobs and Sprites

There are two different types of objects: bobs and sprites. They are handled using the same commands; however they are different in some respects. Bobs have far fewer restrictions placed on them than sprites. Firstly their size is unlimited, whereas sprites are only 16 pixels wide. In addition, they can contain up to 32 colours, depending on the screen mode in operation, whereas sprites are always limited to just three. Finally, any number of bobs can be displayed at once (they are restricted only by the amount of memory available), whereas the number of different sprites is limited to eight.

However, the disadvantage of bobs is that they move slower than sprites and have a tendency to flicker. Therefore where the limitations of sprites are no problem, they should be used in preference to bobs.

The Object Editor

Both kinds of objects can be be created using the Object Editor, which is supplied on the Extras Disk. To use this program, load your copy of the Extras Disk and open the BasicDemos drawer by double-clicking on it. This contains several different BASIC programs. The one we are interested in is called 'ObjEdit'. Start up this program, again by double-clicking on it. The Amiga will automatically load BASIC into the computer so that the program can run; hence it may take a few seconds before anything appears on the screen.

When the program starts, the screen will contain the following text:

```
Enter '1' if you want to edit sprites
Enter '0' if you want to edit bobs
```

Respond to this by entering '0' and the screen will change. The shape in the top left-hand corner is the 'canvas' on which you will be creating your object. You can change the size of this as though it were an ordinary window by pointing at the bottom right-hand corner and pressing the left mouse button. Then, as you move the mouse with the button held down, an orange outline of the window will move with it. At the same time, the two numbers at the bottom of the screen will change. These give the horizontal and vertical dimensions of the canvas in pixels.

To start creating an object, move the pointer to a position within the canvas and then press and hold down the left mouse button. As you move the mouse, the pointer will act as a pen and will leave a white trail behind it from its tip.

You can make the pointer create other shapes by selecting different tools to use. The tool currently selected is stated at the bottom of the screen. By default this is the pen. To see the others available, press the Menu button and select the Tools menu. Try selecting and experimenting with each of these in turn. Their actions are described briefly below:

Pen
The pen allows you to sketch lines and curves freehand. The outline created is determined by the path taken by the tip of the mouse pointer whilst the mouse button is held down.

Line
Line is for creating straight lines. A line is created from the position of the tip of the pointer when the mouse button is pressed to the position it is at when the button is released

Oval
This tool enables you to create circles and ellipses. Pressing the mouse button fixes one corner of a rectangle, which is displayed in orange as the mouse is moved around. Releasing the mouse button fixes the other corner and an ellipse is drawn so that it touches the centre of all four sides.

Rectangle
This is similar to the oval tool. However, instead of an ellipse being drawn when the button is released, the rectangle outline itself is created.

Eraser
The eraser allows the pointer to be used to wipe out parts of the canvas. As you move the mouse around with the button held down, the graphics underneath the tip of the pointer will disappear.

Paint
Paint allows you to fill outlines with a colour. To use it, point at the centre of an outline you would like to colour in and click the mouse button. If you think back to BASIC's PAINT command, you should recall that this paints an area which is delimited by lines drawn in the 'border colour'. The paint tool acts in a similar way. It fills in any area which is outlined in the colour you are using to do the painting.

There are four colours which can be used displayed in boxes at the bottom of the screen. To select one of these simply click on it and the word 'Color:', drawn to the left, will change to the colour selected.

It's now time for you to create a proper object which you will use in later sections of this chapter. This first thing to do is to throw away the results of your experimentation with the different tools and start

afresh with a clean canvas. To do this, select the New item from the File menu. This acts like BASIC and reminds you that you haven't saved your current file. Press N, to say that you don't want to save it, and you will be returned to the initial screen and asked if you want to edit a bob or a sprite. Again the answer is a bob.

The object you need to create is a rocket. This wants to be roughly 40 pixels wide by 50 pixels high. It can be as complicated or as simple as you like. When you have finished, select the Save as item from the File menu and in response to the prompt: 'Enter Filename >' type ':Myprogs/Rocket'. This will save the details of the object you have just created on the disc in the directory you created earlier (see Chapter One).

To leave the Object Editor, click the Close gadget in the top left-hand corner of the window. This will return you to BasicDemos drawer. Close this as well, and then enter BASIC in the normal way by double clicking on the AmigaBASIC icon.

Positioning Objects

In the next few sections we will be stepping through the process of making an object move around in a window. AmigaBASIC has a whole host of commands for dealing with objects. At each stage, one or two of these will be introduced and an example showing their syntax will be given. These examples have been chosen so that you can use them to build up a short demonstration program. Therefore, before you read any further, create a new program which contains the following two lines:

```
WHILE INKEY$ = ""
WEND
```

Then each time a section of BASIC is given, add it to the program. Unless the text states otherwise, it should be placed immediately above the WHILE statement. Then try running the whole program again to see what effect the latest command(s) have had. The WHILE loop is needed because any sprites set in motion during a program stop as soon as the program ends. This loop allows the program to run

as long as you want it to so you can see how the sprites behave. To stop it, press any key.

The first stage is to assign the shape you created in the previous section to a particular object. This involves three separate commands: one to open the file containing the shape definition, one to read the shape definition and assign it to an object, and another to close the file again. These commands will be glossed over here and discussed properly in Chapter 10 which is dedicated to file handling. The only one which requires any comment at this time is OBJECT.SHAPE. This is the first object handling command which we have met. It takes two arguments, an object number and the definition of the shape as produced by the Object Editor. For example:

```
OPEN  "Myprogs/Rocket" FOR INPUT AS 1
OBJECT.SHAPE 1, INPUT$(LOF(1),1)
CLOSE 1
```

Each shape definition that you use requires a certain amount of memory the larger the shape the greater the amount of space needed. Therefore, BASIC provides an alternative version of OBJECT.SHAPE which allows you to create a new object by copying the shape of an existing one. This allows the objects to share the same memory for holding the description of the shape. However you can still control them independently. For example:

```
OBJECT.SHAPE 2, 1
OBJECT.SHAPE 3, 1
```

This creates two further objects, both with the same shape as object number one.

Having defined what an object is to look like, the next step is to place it at its starting position. There are two commands for this: OBJECT.X and OBJECT.Y. Each takes the number of the object and a value giving either the X or Y co-ordinate. The co-ordinate refers to the top left-hand corner of the rectangle which encloses the object, and its value can be any number in the range -32768 to 32767. However, if you want the object to start within the window area, a number in the range from zero to the maximum window size should be used. For example:

```
OBJECT.X 1,220
OBJECT.Y 1,140
OBJECT.X 2,320
OBJECT.Y 2,140
OBJECT.X 3,420
OBJECT.Y 3,140
```

Even though the commands above positioned the objects within the window extent, nothing will appear when you run the program you have created so far. There is still one more step needed. This is to give the command OBJECT ON to 'turn on' the objects and so make them visible. You can make particular objects visible by following this command by a list of object numbers separated by commas. However, if you don't provide an argument, all the objects within the current output window will be affected. For example:

```
OBJECT.ON
```

The inverse of this command is OBJECT.OFF, which makes makes one or all of the objects invisible again.

At this stage your program should display the three rockets near the bottom of the screen. If you have made them so large that they don't all appear on the screen, decrease the initial Y co-ordinates in the OBJECT.Y commands above to move them higher up the screen. It is important, for the purposes of this program, that they don't start off overlapping the edge of the window.

Setting Things in Motion

To set an object in motion, you first have to decide what 'velocity' or speed you want it to move at. You can then assign this velocity to the object using the two commands OBJECT.VX and OBJECT VY. These allow the horizontal velocity and the vertical velocity to be controlled separately. As you might expect, they each take two arguments. The first is the object number and the second is the size of the velocity. For example:

```
OBJECT.VY 1,  -1
OBJECT.VY 2,  -2
```

```
OBJECT.VY 3,  -3
```

These give the objects vertical velocities of –1, –2 and –3 respectively. The horizontal velocities have not been defined therefore they will be zero, since this is the default value assigned when the objects were created. These velocities will move them in the direction of the top of the screen. A positive value for one component of the velocity will move an object so that the value of the corresponding co-ordinate increases. Conversely, a negative value will move it so that the co-ordinate decreases. This is shown in Figure 9.1.

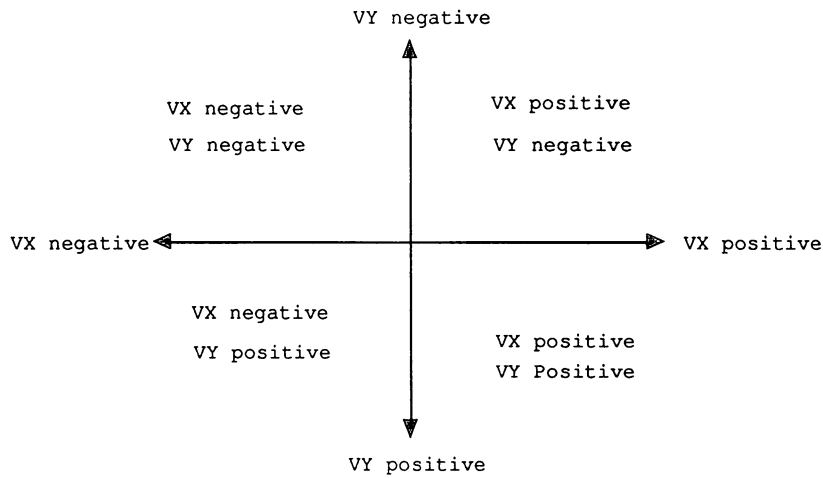


Figure 9.1. The 'quadrant' coordinates.

Velocity is measured in units of pixels per second. Remember that in the high-resolution modes, the pixels are almost twice as high as they are wide. Therefore a velocity of one in both directions will move an object almost twice as fast vertically as it does horizontally.

When you placed the objects on the screen, you found that they weren't visible straight away. An additional command was required to turn them on. Similarly, when you assign the velocities to the objects they will remain stationary until you give the command OBJECT.START to make them start moving:

```
OBJECT.START 1,2,3
```

Again, this command can take a list of objects as shown above. Alternatively we could have omitted the argument, as we did with OBJECT.ON to affect all the objects within the window, and this would have produced the same result.

The problem with the example now is that the rockets don't act in a very realistic manner. Rockets don't move with a constant velocity as soon as they leave the ground instead they accelerate at an impressive rate. Fortunately AmigaBASIC provides us with commands to assign accelerations to objects as well as velocities. These are OBJECT.AX and OBJECT.AY and they take the same syntax as their velocity counterparts:

```
OBJECT.AY 1,-1
OBJECT.AY 2,-2
OBJECT.AY 3,-3
```

Giving an object an acceleration of n means that each second its velocity will increase by 'n' pixels per second. A negative value decreases the velocity.

The Area of Action

The first of Isaac Newton's laws of motion says that a body moving with a constant velocity will continue moving in a straight line and at the same speed unless it is acted on by a force. This law, with slight modifications, describes how objects act. An object will continue moving in a straight line (and if it is moving at a constant velocity at the same speed) unless it collides with something or a command is given to stop or alter it.

You should have found that, when you ran the program created so far, the rockets all stopped when they reached the top of the window. The animation system actually has its own rectangular area, outside of which objects cannot be drawn. The default value happens to coincide with the border of the current output window. However, you can alter it by using the OBJECT.CLIP command. Add the following line to the start of your program:

```
OBJECT.CLIP (0,40) - (620,200)
```

Now when you run the program, the objects should stop well below the top of the screen.

Note that if you change the size of the window, the object boundary will not automatically change with it. You will have to use OBJECT.CLIP to change it explicitly.

To stop an object yourself, use the command OBJECT.STOP. For example place the following inside the WHILE loop.

```
IF OBJECT.Y(3) <= 20 THEN
  OBJECT.STOP 1
END IF
```

There are four object functions, which each require an object number as an argument, and return information about the object as follows:

Function	Returns
OBJECT.X	X co-ordinate of upper left-hand corner of its rectangle
OBJECT.Y	Y co-ordinate of upper left-hand corner of its rectangle
OBJECT.VX	Velocity in the X direction
OBJECT.VY	Velocity in the Y direction

The lines above test to see if object three has reached the top object boundary and, if so, they stop object one from moving any further. Therefore both object three and object one will stop at the same time.

We have seen that, by default, objects always stop when they hit the boundary. The same thing also happens if they hit each other. However, you can choose not to have that happen by using OBJECT.HIT. This is quite a complicated command which takes up to three arguments. The first is the number of the object being dealt with. The other two provide information about what happens if the object collides with another object or another object collides with it.

The least significant bit of the final argument specifies if the object is to collide with the boundary or not. If the bit is set (ie the number is odd)

then it will collide. However, if the bit is clear (ie the number is even) then it will not collide and will fly out of the visible area.

To find out about the other bits, consider three objects as follows:

Shape1	0000000000000010	0000000000000111
Shape2	0000000000000100	0000000000000111
Shape3	0000000000001000	0000000000000000

The first number provides the bit pattern for the shape itself. The second number gives the bit pattern representing the objects which it can collide with. When one object hits another, the first bit pattern of the first object is ANDed with the second bit pattern for the second object. If the result is zero then no collision takes place.

So, using the above illustration, Shape 1 can collide with Shape 2 and vice versa. However, Shape 3 cannot collide with anything since none of the objects has the fourth bit set. Similarly, nothing can collide with it since its last number is zero.

In the case of our example, the only items which the rockets can collide with are the borders therefore only the last bit of the second value is of significance. Adding the following line to the program will mean that the second rocket does not collide with the border. Instead it flies out of the window, leaving you with only two objects visible:

```
OBJECT.HIT 2,,0
```

This marks the end of the example program.

Handling Collisions

Using a function to test to see if an object has collided with a boundary is fine for our simple example above. However, a better way of handling collisions exists. Collisions are another type of 'event' which the Amiga can trap. Collision trapping can be enabled using COLLISION ON, nd a subroutine established for handling any collisions which occur using ON COLLISION GOSUB.

The COLLISION function can be used to obtain information about a collision which has occurred. If it is called with an argument of zero, then it returns the identification number of an object which has been

involved in a collision. A subsequent call with an argument of –1 returns the identification number of the window in which the collision occurred. However, if the argument is greater than zero, then the function will treat it as an object number and return the number of the object with which it collided or one of the values below to indicate a collision with a boundary:

–1 Top
–2 Left
–3 Bottom
–4 Right

AmigaBASIC is capable of storing the information about 16 collisions. Each time COLLISION is called to get information about a particular collision, this information is removed from the queue. However, if the queue is full and another collision occurs, then the information about this new collision will be thrown away and lost.

We will now finish this chapter by demonstrating collision handling along with further uses of the OBJECT commands in a brief animated scene:

```
Init:
SCREEN 1,640,256,2,2
WINDOW 2,"Traffic",,16,1
OBJECT.CLIP (0,0) - (639,255)
i% = COLLISION(0)
WHILE i% <> 0
  i2% = COLLISION(i%)
  i% = COLLISION(0)
WEND

ScreenInit:
PALETTE 3,.2,1,.2
LINE (0,255) - (639,40),3,bf
LINE (0,80) - (639,140),2,bf
LINE (0,109) - (639,111),1,bf
FOR dash% = 0 TO 620 STEP 8
  LINE (dash%,124) - (dash%+4,126),1,bf
  LINE (dash%, 94) - (dash%+4, 96),1,bf
NEXT
```

```
ReadShapes:
OPEN "df0:carl" FOR INPUT AS 1
OBJECT.SHAPE 1, INPUT$(LOF(1),1)
CLOSE 1
OPEN "df0:carr" FOR INPUT AS 1
OBJECT.SHAPE 11, INPUT$(LOF(1),1)
CLOSE 1

Collision.Init:
ON COLLISION GOSUB collision.handler
COLLISION ON

CarInit:
lanes% = 2 : cars.in.lane% = 2
space% = 640/cars.in.lane%
RANDOMIZE TIMER

CarLeft:
FOR car% = 2 TO lanes%*cars.in.lane%
  OBJECT.SHAPE car%,1
NEXT
FOR lane% = 1 TO lanes%
  FOR car% = 1 TO cars.in.lane%
    carno% = (lane%-1)*cars.in.lane% + car%
    OBJECT.X   carno%, INT(RND*(space%-32)) +
space%*(car%-1)
    OBJECT.Y   carno%,104+lane%*16
    OBJECT.VX  carno%,-40*(lanes%-lane%+1)
  NEXT
NEXT

CarRight:
FOR car% = 12 TO 10 + lanes%*cars.in.lane%
  OBJECT.SHAPE car%,11
NEXT
FOR lane% = 1 TO lanes%
  FOR car% = 11 TO 10 + cars.in.lane%
    carno% = (lane%-1)*cars.in.lane% + car%
    OBJECT.X   carno%, INT(RND*(space%-32)) +
space%*(car%-11)
    OBJECT.Y   carno%,120-lane%*16
    OBJECT.VX  carno%,40*(lanes%-lane%+1)
  NEXT
```

```
NEXT

MainBody:
OBJECT.ON
OBJECT.START
FOR loop% = 1 TO 10000 + RND*10000
NEXT
OBJECT.VY 1,-36
OBJECT.AX 1,2
OBJECT.AY 1,2
WHILE OBJECT.VY(1) < 0
WEND
OBJECT.STOP 1
WHILE 1 = 1
WEND
END

collision.handler:
  obj1% = COLLISION(0)
  WHILE obj1% <> 0
    obj2% = COLLISION(obj1%)
    IF obj2% < 0 THEN
      IF obj1% > 10 THEN
        OBJECT.X obj1%, 0
      ELSE
        OBJECT.X obj1%, 600
      END IF
      OBJECT.START obj1%
    ELSE
      PRINT SPC(28) "CRASH!"
    END IF
  obj1% = COLLISION(0)
  WEND
RETURN
```

The program requires two sprites to be pre-defined and saved as "carll" and "carr" on the current disc. These should be roughly 16 pixels high and should represent cars facing left and right respectively. Note that when defining sprites, the menu option for enlarging the canvas to four times its normal size is very useful. The disadvantage of this is that only pen mode can be used. However, it

does provide a far more accurate means of drawing the shape you want.

The program works as follows:

Init:
Firstly, a new screen is created at the maximum size, allowing four colours to be used. Then a window is drawn occupying the whole of the screen and the object area is set up to be the same size as the screen.

Then a loop is executed repeatedly until COLLISION(0) no longer returns the value zero. If a different value is returned then COLLISION is called with this number as a parameter. This removes it from the collision list. Hence the effect of this loop is to clear any collisions from the stack which may be there from previous programs.

ScreenInit:
This sets up colour three to be a shade of green and then draws the background for the scene.

ReadShapes:
This opens the two sprite files, reads the sprite definitions and assigns these shapes to objects one and 11.

CollisionInit:
This sets up the collision handler and turns collision event trapping on.

CarInit:
The following variables are set up at this stage:

lanes%	Number of lanes on each side of the road
cars.in.lane%	Number of cars per lane
space%	Amount of road per car in pixels

Then the random number generator is re-seeded.

CarLeft:
This sets up the details for the cars in the left-hand lanes. One car pointing in this direction already exists as object one. The others required are copied from this one. The number needed is given by the number of cars per lane multiplied by the number of left-hand lanes.

Then all these cars are given an initial starting position and horizontal velocity.

The vertical position is such that the car lies in the correct lane. The velocity is also determined by the lane, so that the cars in the outer lanes travel faster. Within each lane, every car is positioned randomly within the space available for it. For example, if there are two cars per lane, one car is placed at random within the left-hand half and the other at random within the right-hand half. Note that a small amount (32 pixels) is deducted from the right-hand edge of the space for each. This is so the cars are never started on top of each other. Otherwise, if one car was at the far right of its area and the next at the far left of its space then they would overlap.

CarRight:
This performs the same action for the cars travelling in the opposite direction.

MainBody:
Within the main body, the objects are made visible and are set in motion. For a random length of time, they move along as they were initialised to do. Then object number one is assigned a vertical velocity so that it swerves across the other carriageway. It is also given an acceleration which serves to decrease the absolute size of both the vertical and horizontal velocity. Therefore it starts slowing down.

The first empty WHILE loop repeats while object one is still moving up the screen (ie its horizontal velocity is negative). Although the acceleration acts originally to slow the car down, once the velocity in a particular direction has reached 0, then the acceleration would increase the velocity again but in the opposite direction. This would make the car appear to be travelling backwards at an ever increasing rate. To prevent this happening the object is stopped as soon its vertical movement has changed direction (ie when the sign of the velocity has changed from negative to positive).

Then the program just continues running until the Stop item is selected from the Run Menu, or Amiga-Fullstop is pressed.

collision.handler:
This performs most of the work of the program. It starts by reading the number of one of the objects which has collided. Then it reads the number of the object or edge which it has collided with. If the collision was with one of the edges of the screen (which have negative values) then the car is repositioned at the other side of the screen and restarted. This gives the appearance of one car driving into the picture when another has just driven out of it. Otherwise, it must have collided with another car in which case the word 'CRASH!' is printed. Note that the collision automatically stops the cars so they will be left stationary in the road.

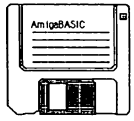
Note that the collision handling is performed in a loop. The routine is entered only when a collision occurs. Therefore, there will always be a collision pending on entry to this code. This means that the loop will always be performed once. The last statement in the loop checks to see if there is another collision in the buffer. If so, the collision handling code is automatically repeated.

This loop is necessary because of the way in which event trapping works. On entry to the collision handling routine, an ON COLLISION STOP is automatically performed. This means that BASIC buffers any further collisions but does not act on them. On exit from the routine, the event trapping is automatically turned on again. In theory, when event trapping is turned ON, any events which occurred whilst it was STOPped will immediately be registered and acted upon. Unfortunately, in practice this doesn't always occur. Therefore, without the loop, after one collision had been dealt with there could be another waiting in the buffer which didn't cause the collision handler to be called. This would mean that the collision handling would get out of step. The next collision would cause the handler to be entered, but the details of the previous collision would be read and acted upon.

To see the effect of this, comment out the WHILE and WEND statements using REMs. Then try running the program a few times. You should soon notice a very strange effect – cars sometimes appear to pause when they hit the edge of the screen. If you examine what is happening more closely you will see that they only start moving again when another collision occurs. This is because one of the collisions has

occurred whilst the previous collision was being dealt with and the trap for it has failed to occur. Therefore, its details are read when the next collision takes place. Since collisions automatically cause the objects to stop, the object pauses until the next collision occurs and its own collision details are handled.

10 : File Handling



So far we have saved and loaded both BASIC programs and graphic screen images. This chapter goes on to describe a new kind of file – the 'data file'. Data files can contain many different sorts of information. For example, documents created on a wordprocessor, names and addresses held in a card index system and exam results typed into a school statistics package are all examples of information which can be stored on disc as data files. Without data files most types of software would not be viable. Can you imagine using a wordprocessor which didn't allow you to save the letters you wrote?

Sequential Files

There are two different types of data files: sequential files and random access files. We are going to start off looking at the sequential files, since they are the easier of the two to create and use.

Sequential files get their name from the fact that the items of data written to them are stored sequentially in the order that they were written. In addition, information is read from them sequentially in the same order.

Creating and Opening Files

In order to be able to write data to a sequential file or read data from it, the file has to be open. You can open a file using the command OPEN. For example:

```
OPEN "Data1" FOR OUTPUT AS 1
```

This creates a file called 'Data1' in the current directory and opens it so that we can output information to it. The number '1' is the value which is to be associated with the file while it is open. Since it is possible to have several files open at once, the other filing system routines have to be told which file they are to act upon. This is done by giving them the number of the file. Therefore each file you open has to

have a unique file number. This can be any integer in the range one to 255.

If you already have a file called 'Data1', then the command above will delete the existing file and create a new, empty one. Therefore this command is no use for adding extra data onto the end of a file you have created previously. To do this you have to open the file for APPEND:

```
OPEN "Data2" FOR APPEND AS 2
```

This opens a data file called 'Data2' so that data can be added to the end of it. If a file of that name doesn't already exist then the command acts exactly as though the OUTPUT mode had been used and the file is created.

The third option is:

```
OPEN "Data3" FOR INPUT AS 3
```

This opens an existing file so that data can be 'input' or read from it. Note that in this case the file must already exist. If it doesn't an error message is generated.

At any given time, a file can only be open once. Therefore you cannot be writing to and reading from a sequential file at the same time.

Once you have finished accessing a file you should close it as follows:

```
CLOSE 1
```

This command closes the file whose file number is one. This file number can then be allocated to a different file.

Outputting and Inputting Data

Outputting data to a sequential file is similar to outputting it to the screen. For example:

```
PRINT "Smith"
```

writes:

```
Smith
```

to the screen, whereas:

```
PRINT# 1,"Smith"
```

writes:

```
Smith
```

to file number one.

Similarly, inputting data from a file is like inputting it from the keyboard:

```
INPUT surname$
```

reads a string from the keyboard and assigns it to the variable surname$. Whereas:

```
INPUT# 1,surname$
```

reads a string from file number one and assigns it to surname$.

The first of the following programs shows how data can be read in from the user and output to a file. The second then reads the data from the file and prints it on the screen.

```
REM Program to output data to a file
OPEN "Data1" FOR OUTPUT AS 1
ans$ = "Y"
WHILE ans$ = "Y"
  REM read in data for a student from the keyboard
  INPUT "Surname        :",surname$
  INPUT "First name     :",name1$
  INPUT "Maths result   :",maths%
  INPUT "English result :",eng%
  REM output this data to the file
  PRINT# 1,surname$
  PRINT# 1,name1$
  PRINT# 1,maths%
  PRINT# 1,eng%
```

```
    REM ask if there are any students left
    INPUT "Do you wish to continue (Y/N) :",ans$
    ans$ = UCASE$(ans$)
WEND

REM All done so close file
CLOSE 1
REM Program to read data from a file
OPEN "Data1" FOR INPUT AS 1
WHILE NOT EOF(1)
    REM read data for a student from the file
    INPUT# 1,surname$
    INPUT# 1,name1$
    INPUT# 1,maths%
    INPUT# 1,eng%

    REM Print this data on the screen
    PRINT "Surname        :";surname$
    PRINT "First name     :";name1$
    PRINT "Maths result   :";maths%
    PRINT "English result :";eng%
WEND
REM All done so close file
CLOSE 1
```

Note that the second program uses the function EOF to test to see if there is any more data left to read. This function returns TRUE if the end of the file has been reached, and FALSE otherwise.

These two programs show just how simple file handling can be. However, there are some points which you should look out for. In the first example above, each item of data was written to the file separately. This meant that a carriage return character was placed after every item of data. This acted as a 'delimiter', ie it marked where one item ended and the next began. Therefore, when the second program came to read the items, it knew exactly what to read. It is far more convenient to be able to write several items of data together, but you must be careful when doing this. For example:

```
PRINT# 1,"Smith";"Fred"
```

writes:

```
SmithFred
```

to file number one. This means that the two strings are merged so that they look like one long one. When we come to read data back in from the file, they are read together as "SmithFred" and assigned to the first string variable, and there is then nothing left to assign to the second string variable. Even separating them by spaces using:

```
PRINT# 1,"Smith","Fred"
```

which outputs:

```
Smith        Fred
```

has a similar result because the INPUT# routine treats the spaces as part of the string and so reads it as "Smith Fred".

What we need to do is to separate them ourselves using some marker which the input routine will recognise as a delimiter. We can do this as follows:

```
PRINT# 1,"Smith";",";"Fred"
```

This outputs:

```
Smith,Fred
```

Now, the two strings are separated from each other by a comma which the input routine recognises as a marker between individual items. Having to separate strings by a comma poses another problem: what happens if the string we want to output contains a comma? This is the opposite situation from the one we had above. For example:

```
PRINT# 1,"Smith,Fred"
```

outputs:

```
Smith,Fred
```

This time something which was output as a single item looks like two separate ones and so only part of the string will be read back in. The solution to this problem is to enclose the string in double quotes as follows:

```
PRINT# 1,CHR$(34);"Smith,Fred";CHR$(34)
```

This outputs:

```
"Smith,Fred"
```

The double quotes now override the comma. Anything between the first double quote and the second one will be read in in one go. This process is now getting rather complicated. To output the contents of two string variables A$ and B$ the following would be required:

```
PRINT# 1,CHR$(34);A$;CHR$(34);",";CHR$(34);B$;CHR$(34)
```

which is far too verbose. A better method of outputting strings is to use WRITE#, for example:

```
WRITE# 1, A$,B$
```

This automatically outputs commas between the individual items in the list and encloses strings in quotes, and so does the work for us. WRITE# can be used for outputting numbers as well. In fact it can take any list of expressions provided that they are separated by commas, for example:

```
WRITE# 1,"Fred",A$,B$+C$,123,26.4,5+6
```

However, when outputting numbers, you may wish to use PRINT# USING since this allows you to format the numbers as you write them to the file. For example:

```
PRINT# 1, USING "##.##,";27.4;89.123;45
```

outputs:

```
27.40,89.12,45.00
```

Note that the comma at the end of the format string causes the items to be separated by commas.

Buffers

You may have noticed, when you ran the program to output data to a file, that the disc drive was only activated intermittently. This is because AmigaBASIC doesn't write each item of data to the disc immediately. Instead it buffers them up and only writes them out when its buffer is full, or when you tell it to close the file. It does this to save time, since accessing the disc is a very slow process. Therefore, if you have a lot of data to send to a file, you can speed up the operation by increasing the size of the buffer. To specify a buffer size, add an extra argument when you open the file, for example:

```
OPEN "Data1" FOR OUTPUT AS 1 LEN = 1000
```

This defines the length of the buffer to be 1000 bytes, which is almost 1k. The default size is 128 bytes, and the maximum size you can ask for is 32767 bytes.

The statement above shows us the full syntax of the OPEN statement. However, an alternative version exists:

```
OPEN "O", 1, "Data1", 1000
```

The first argument specifies the mode, the first letter of the string determines it as follows:

O	Output
A	Append
I	Input

The second is the filenumber, the third the name of the file and the fourth, optional, argument is the buffer size.

Random Access Files

Sequential files are ideal for storing information which you want to access in the same order every time. However, if you want to be able to 'jump about' within the file accessing the data in a different order then you'll find that they can be very slow. For example, if you type in and use the pair of programs above for storing exam results then you will find that these programs are fine if you want to read all the data in to produce statistics. However, what happens if you want to find the results of individual children? The only way to do it is to start at the beginning of the file and read every item of data in turn until you come to the set about the child you are interested in. Some of the time you will be lucky and find the information you want close to the front of the file. Other times you will have a lot of reading to do before you come to the data you want.

If this is how you want to use the data in a file, then you should use 'random access' files instead of sequential ones. These allow you to access the information in any order very quickly.

Random access files are composed of 'records'. If you compare a random access file with a card indexing system then each record corresponds to a card. The records themselves are split into 'fields', which correspond to the different sections that the card is divided up into.

As another example, consider the data we stored for the children's exam results. A random access file containing this information would contain one record per child. This record would have four fields: the surname, the first name, the maths result and the English result.

When creating a random access file, you have to decide in advance what size, measured in bytes, the fields are going to be. If a field is to hold characters, then the size is equivalent to the maximum number of characters it can contain. Numbers have to be converted into strings before they can be saved in a random access file. For each type of number, functions exist which will convert the number into an appropriate string and the size of the string depends on the type of number as follows:

Short integer	2 bytes
Long integer	4 bytes
Single precision real	4 bytes
Double precision real	8 bytes

For certain string fields, deciding the length is easy. For example, if you want to store dates, then you know that each can be written in the format: DD/MM/YY, which means that the field is always going to be eight characters long. However, for other things such as names, you just have to make an educated guess at the longest name that you are likely to come across and be prepared to miss out a few letters if you have to type in one that is longer than you anticipated.

Writing to Random Access Files

We are going to see how to create a random access file by considering the example of a simple address book containing:

Surname	20 characters	= 20 bytes	eg:	Jones
First name	20 characters	= 20 bytes	eg:	Fred
Address	20 characters	= 20 bytes	eg:	10 High St
Town	20 characters	= 20 bytes	eg:	Townthorpe
County	10 characters	= 10 bytes	eg:	Cambs
Postcode	8 characters	= 8 bytes	eg:	CB41 8QX
Phone	10 characters	= 10 bytes	eg:	0123456789
Birthday	6 characters	= 6 bytes	eg:	02 Mar
Age	1 short int	= 2 bytes	eg:	6

This gives a total record length of 116 bytes

To open a file for holding data in this format use the following:

```
OPEN "RanData1" AS 1 LEN = 116
```

This opens the random access file as file number one for both input and output. Note that the length stated for the buffer size has to be the same as the length of each record.

The alternative version of this is:

```
OPEN "R",1,"RanData1",116
```

The next step is to allocate separate areas in the buffer for the different fields. For example:

```
FIELD #1, 20 AS snam$, 20 AS fnam$, 20 AS addr$, 20 AS twn$,
10 AS cnty$, 8 AS post$, 10 AS phon$, 6 AS brth$, 2 AS age$
```

For each field, you have to supply the size in bytes/characters and a variable name. These variables are different to other string variables. The position in memory which they point to and hence where strings placed in them are stored, is part of the random file buffer. It is important not to assign strings to them using statements such as LET or INKEY$, since this converts them into being 'normal' string variables pointing into the normal string space. Special routines, which we will find about shortly, are supplied to place strings in field variables.

Note that the whole record description must all be given on one line. Each FIELD statement starts assigning the variables to space in the random file buffer, starting at the beginning. Therefore giving two FIELD statements such as:

```
FIELD #2, 20 AS a$
FIELD #2, 20 AS b$
```

would just provide alternative variable names pointing to the same area of the buffer.

Now we are ready to place data in the random access buffer by assigning strings to the field variables. Numbers must first be turned into strings using the following functions:

MKI$	To convert a short integer into a string
MKL$	To convert a long integer into a string
MKS$	To convert a single-precision real into a string
MKD$	To convert a double-precision real into a string

To place strings into the buffer there are two different statements provided, LSET and RSET. If the string is shorter than the maximum length of the field then LSET places it in the field so that it is left-justified, whereas RSET enters it so it is right-justified. In both cases the extra character positions are padded with spaces. With either

statement, if the string is too long, then any excess characters are lost from the right-hand side.

For our example, this process would look something like the following:

```
LSET snam$ = surname$
LSET fnam$ = firstname$
LSET addr$ = address$
LSET twn$  = town$
LSET cnty$ = county$
LSET post$ = postcode$
LSET phon$ = phone$
LSET brth$ = birthday$
LSET age$  = MKI$(age%)
```

The variables on the left-hand side are all field variables and those on the right are normal variables which could have been assigned to using INKEY$ or READ etc. You can use constant strings or expressions but normally these won't be appropriate.

Now all that remains to be done is to write out the contents of the random file buffer to the random access file. The command to do this is PUT, for example:

```
PUT #1, 5
```

This will write the contents of the buffer to file number one, as record number five. If you omit the record number, for example:

```
PUT #1
```

then the next record number will be used (ie one greater than that of the previous PUT). The lowest record number you can use is one, the highest depends on the amount of space there is available on your disc and how large your records are. For example, since our records are 116 bytes long, you can fit just over 3800 of them onto one side of an empty disc. This means that our largest record number would be about 3800. BASIC imposes a maximum value of 16777215 which is

unlikely to limit anyone, even if they used very small records and had a hard disc drive attached to their Amiga.

The number of the last record used can be found using the function LOC. For example:

```
rec% = LOC(1)
```

where the argument is again the file number.

Similarly the total length of the file in bytes can be obtained using LOF. For example:

```
file.size% = LOF(1)
```

Reading from Random Access Files

The first two stages involved in accessing data from a random access file are identical to those for writing data to one. Firstly you must OPEN the file in random mode, remembering to specify the random file buffer size. Then you have to use FIELD to allocate blocks of space in this buffer for the variables to be read and assign field variables to them.

The next step is to read the contents of the record required into the buffer. The command for this is GET. GET does the reverse operation to PUT and has the same syntax as it, for example:

```
GET #1, 5
```

Once you have the strings in the buffer, they can be assigned to normal variables in the usual way. To convert any which were originally numbers back from their string format into the appropriate type of number, four functions are provided:

CVI	To convert a string into a short integer
CVL	To convert a string into a long integer
CVS	To convert a string into a single-precision
CVD	To convert a string into a double-precision

Therefore the relevant part of our program would look something like this:

```
surname$   = snam$
firstname$ = fnam$
address$   = addr$
town$      = twn$
county$    = cnty$
postcode$  = post$
phone$     = phon$
birthday$  = brth$
age%       = CVI(age$)
```

Putting Theory into Practice

This section has introduced several new keywords and a lot of new concepts. It's now time to apply these to a real example program. The following implements the Address Book whose format we have been using above. It allows you to enter, edit or delete records and save them to disc. In addition, it provides searching facilities so you can select particular ones that you are interested in. For example this allows you to find the record of your friend called Fred Smith to look up his address, or to look for everyone you know who has a birthday in a particular month.

```
ArrayInit:
OPTION BASE 1
DIM SHARED row%(11), col%(11), box%(11)
FOR loop% = 1 TO 9
  row%(loop%) = 1 + 2*loop%
  col%(loop%) = 1
  box%(loop%) = 12
NEXT
row%(10) =  7 : row%(11) = 12
col%(10) = 41 : col%(11) = 41
box%(10) = 41 : box%(11) = 41
DIM SHARED title$(11), max%(11)
FOR loop% = 1 TO 11
  READ title$(loop%),max%(loop%)
NEXT
DIM SHARED strg$(9), strgt$(9), search$(9), sbuf$(9)

ScreenInit:
charhgt% = 9 : search% = 0
FOR loop% = 1 TO 11
  LOCATE row%(loop%), col%(loop%)
  PRINT title$(loop%)
```

```
  CALL rect(loop%,1)
NEXT

FileInit:
  OPEN "df0:Address" AS 1 LEN = 116
  FIELD #1, 20 AS sbuf$(1), 20 AS sbuf$(2), 20 AS sbuf$(3), 20 AS
     sbuf$(4), 10 AS sbuf$(5), 8 AS sbuf$(6), 10 AS sbuf$(7), 6 AS
     sbuf$(8), 2 AS sbuf$(9)
  CALL loadrec(1,1)

MouseInit:
ON MOUSE GOSUB mousehandler
MOUSE ON

MenuInit:
MENU 5,0,1,"Address"
MENU 5,1,1,"Set search"
MENU 5,2,1,"Clear search"
MENU 5,3,1,"Quit"
ON MENU GOSUB menuhandler
MENU ON

MainBody:
WHILE 1 = 1
  ch$ = ""
  WHILE ch$ = ""
    ch$ = INKEY$
  WEND
  IF fld% <> 0 THEN
    IF ASC(ch$) = 13 THEN
      IF fld% < 9 THEN
        strg$(fld%) = cur$
        CALL rect(fld%,1)
        fld% = fld% + 1
        CALL rect(fld%,2)
        cur$ = ""
        CALL clearbox(fld%)
      ELSE
        BEEP
      END IF
    ELSEIF ASC(ch$) = 8 OR ASC(ch$) = 127 THEN
      IF LEN(cur$) > 0 THEN
        cur$ = MID$(cur$,1,LEN(cur$)-1)
        LOCATE ,POS(0)-1
        PRINT " ";
        LOCATE ,POS(0)-1
      ELSE
        BEEP
      END IF
    ELSE
      IF LEN(cur$) < max%(fld%) THEN
        PRINT ch$;
        cur$ = cur$ + ch$
      ELSE
        BEEP
      END IF
    END IF
```

```
  ELSE
    BEEP
  END IF
WEND

rect:
SUB rect(item%,col%) STATIC
  SHARED charhgt%
  x1% = (box%(item%) - 1)*10
  x2% = x1% + max%(item%)*10
  y1% = (row%(item%) - 1)*charhgt%
  y2% = y1% + charhgt%
  LINE (x1%-2,y1%-2) - (x2%+2,y2%+2),col%,b
END SUB

trailrem:
SUB trailrem(array$()) STATIC
  FOR loop% = 1 TO 9
    char$ = " "
    WHILE char$ = " "
      char$ = RIGHT$(array$(loop%),1)
      IF char$ = " " THEN
        array$(loop%) = MID$(array$(loop%),1,LEN(array$(loop%))-1)
      END IF
    WEND
  NEXT
END SUB

clearbox:
SUB clearbox(item%) STATIC
  LOCATE row%(item%),box%(item%)
  PRINT SPACE$(max%(item%))
  LOCATE row%(item%),box%(item%)
END SUB

saverec:
SUB saverec(recno%) STATIC
  IF recno% > 0 THEN
    FOR loop% = 1 TO 8
      trails$ = SPACE$(max%(loop%)-LEN(strg$(loop%)))
      LSET sbuf$(loop%) = strg$(loop%) + trails$
    NEXT
    IF strg$(9) = "" THEN
      LSET sbuf$(9) = "  "
    ELSE
      LSET sbuf$(9) = MKI$(VAL(strg$(9)))
    END IF
    PUT #1,recno%
  END IF
END SUB

loadrec:
SUB loadrec(recno%,dir%) STATIC
  SHARED fld%,cur$,rec%,search%
  IF recno% > 0 THEN
    IF dir% = 1 THEN limit% = 51 ELSE limit% = 0
    ok% = 0 : endf% = 0
```

```
    reccur% = recno%
    WHILE reccur% <> limit% AND ok% = 0 AND endf% = 0
      GET #1,reccur%
      IF NOT EOF(1) THEN
        FOR loop% = 1 TO 8
          strgt$(loop%) = sbuf$(loop%)
        NEXT
        IF sbuf$(9) = "  " THEN
          strgt$(9) = ""
        ELSE
          strgt$(9) = STR$(CVI(sbuf$(9)))
        END IF
        CALL trailrem(strgt$())
      ELSE
        FOR loop% = 1 TO 9
          strgt$(loop%) = ""
        NEXT
        endf% = 1
      END IF
    CALL check(ok%)
    IF ok% = 0 THEN reccur% = reccur% + dir%
    WEND
    IF ok% = 1 THEN
      LOCATE 1,1
      PRINT "RECORD" reccur% " "
      FOR loop% = 1 TO 9
        strg$(loop%) = strgt$(loop%)
        CALL clearbox(loop%)
        PRINT strg$(loop%)
      NEXT
      rec% = reccur%
    ELSE
      BEEP
    END IF
  ELSE
    LOCATE 1,1
    PRINT "SEARCH    "
    FOR loop% = 1 TO 9
      search$(loop%) = ""
      CALL clearbox(loop%)
    NEXT
    rec% = 0 :
  END IF
  IF fld% <> 0 THEN CALL rect(fld%,1)
  fld% = 0 : cur$=""
END SUB

check:
SUB check (ok%) STATIC
  SHARED search%
  CALL trailrem(search$())
  ok% = 1
  IF search% = 1 THEN
    FOR box% = 1 TO 8
      IF ok% = 1 THEN
        lens% = LEN(search$(box%))
        IF lens% <> 0 THEN
```

```
          posstar% = INSTR(search$(box%),"*")
          IF posstar% = 0 THEN
            lenr% = LEN(strgt$(box%))
            IF lenr% <> lens% THEN
              ok% = 0
            ELSE
              FOR char% = 1 TO lenr%
                chars$ = MID$(search$(box%),char%,1)
                charr$ = MID$(strgt$(box%),char%,1)
                IF UCASE$(chars$) <> UCASE$(charr$) THEN ok%=0
              NEXT
            END IF
          ELSE
            lenlef% = posstar% - 1
            IF lenlef% > 0 THEN
              lefts$ = LEFT$(search$(box%),lenlef%)
              leftr$ = LEFT$(strgt$(box%),lenlef%)
              IF UCASE$(lefts$) <> UCASE$(leftr$) THEN ok% = 0
            END IF
            lenrig% = LEN(search$(box%)) - posstar%
            IF lenrig% > 0 THEN
              rights$ = RIGHT$(search$(box%),lenrig%)rightr$ =
              RIGHT$(strgt$(box%),lenrig%)
            IF UCASE$(rights$) <> UCASE$(rightr$) THEN ok% = 0
            END IF
          END IF
        END IF
      END IF
    NEXT
    IF LEN(search$(9)) <> 0 THEN
      chars$ = LEFT$(search$(9),1)
      nums% = VAL(MID$(search$(9),2))
      numr% = VAL(strgt$(9))
      IF chars$ = "=" THEN
        IF numr% <> nums% THEN ok% = 0
      ELSEIF chars$ = ">" THEN
        IF numr% <= nums% THEN ok% = 0
      ELSEIF chars$ = "<" THEN
        IF numr% >= nums% THEN ok% = 0
      ELSE
        ok% = 0
      END IF
    END IF
  END IF
END SUB

menuhandler:
  IF MENU(0) = 5 THEN
    IF fld% <> 0 AND rec% <> 0 THEN
      strg$(fld%) = cur$
      CALL saverec(rec%)
    END IF
    IF MENU(1) = 1 THEN
      CALL loadrec(0,0)
      search% = 1
    ELSEIF MENU(1) = 2 THEN
      search% = 0
```

```
    ELSEIF MENU(1) = 3 THEN
      CLOSE 1
      END
    END IF
  END IF
RETURN
mousehandler:
  IF fld% <> 0 THEN
    IF rec% > 0 THEN
      strg$(fld%) = cur$
    ELSE
      search$(fld%) = cur$
    END IF
  END IF
  dum% = MOUSE(0)
  xpos% = MOUSE(3) : ypos% = MOUSE(4)
  box% = 0
  FOR loop% = 1 TO 11
    x1% = (box%(loop%) - 1)*10
    x2% = x1% + max%(loop%)*10
    y1% = (row%(loop%) - 1)*charhgt%
    y2% = y1% + charhgt%
    IF xpos% > x1%-2 THEN
      IF xpos% < x2%+2 THEN
        IF ypos% > y1%-2 THEN
          IF ypos% < y2%+2 THEN
            box% = loop%
          END IF
        END IF
      END IF
    END IF
  NEXT
  IF box% = 10 THEN
    IF rec% = 50 THEN
      BEEP
    ELSE
      IF fld% <> 0 THEN CALL saverec(rec%)
      CALL loadrec(rec%+1,1)
    END IF
  ELSEIF box% = 11 THEN
    IF rec% <= 1 THEN
      BEEP
    ELSE
      IF fld% <> 0 THEN CALL saverec(rec%)
      CALL loadrec(rec%-1,-1)
      END IF
  ELSEIF box% > 0 THEN
    IF fld% <> 0 THEN CALL rect(fld%,1)
    fld% = box%
    CALL rect(fld%,2)
    CALL clearbox(fld%)
    cur$ = ""
  END IF
RETURN

DataInit:
DATA "Surname",20
```

```
DATA "First name",20
DATA "Address",20
DATA "Town",20
DATA "County$",10
DATA "Postcode$",8
DATA "Phone no",10
DATA "Birthday",6
DATA "Age",3
DATA "Next",4
DATA "Prev",4
```

The program starts by displaying the first record in the data file called 'address'. If it is unable to find the file on the current disc, it will create a new empty one and display a blank record. To enter data into any of the fields of the record displayed, click the mouse in the appropriate box. This box will then be highlighted in black to show that it has been selected and any existing text it contained will be deleted. You can then type text into it. Pressing either of the delete keys '/' or Del, whilst you are adding data, will delete the character to the left of the cursor. You can move around the fields randomly by clicking in the next one you want to edit. Alternatively, you can move down to the next box by pressing RETURN.

Clicking on the 'Next' and 'Prev' boxes will take you to the next or previous record in the file. If you have edited the record currently displayed, then you should notice the disc being accessed when you move between records. This is because the file is updated to contain the new version of the record which you have just finished altering. The program, as it stands, can hold 50 records. However, this is an arbitrary limit and can be changed very simply.

To set up a search pattern, press the menu button and select the 'Set search' item in the Address menu. An empty record entitled 'SEARCH' will be displayed. Enter the text you wish to match against in the usual way and then click on the 'Next' box. This will move you to the next record which matches the current search pattern or will beep if none are found which match. From then on, using the 'Next' and 'Prev' boxes will move you only between records which match the search pattern. To clear this pattern, select the 'Clear search' item from the Address Menu.

There are a few points to note about how the searching works: Any boxes you leave empty in the search pattern will match anything. For example, typing 'Fred' in the 'First name' box will match against any record for a person called 'Fred', regardless of their surname, address etc.

The case of the text is irrelevant; 'Fred' will match 'Fred', 'fred', 'FRED' etc.

Any spaces typed at the end of the text will be ignored. For example, entering 'Fred ' in the 'First name' box will match against 'Fred'. However, spaces before the text or between words are significant. 'FredSteven' will not match 'Fred Steven'.

A '*' character can be used to match against any number of any characters. For example, specifying a 'First name' of 'F*' will match against anyone whose name begins with the letter 'F'. Similarly typing 'F*D' will find people whose name starts with the letter 'F' and ends with the letter 'D'. Note that only one '*' may be used per field. Any subsequent ones will be treated as actual text.

The 'Age' box is treated differently from the other boxes. The search algorithm allows you to find people whose age is less than, equal to, or greater than a given number. The search string you give should start with either a '<', '=' or '>' character respectively, followed by the number. For example '<21' will find all records where the age given is less than 21.

Finally, to end the program, select the 'Quit' option from the Address Menu. This updates the current record, if necessary, and closes down the file.

The program works as follows:

ArrayInit:
The program starts by initialising all the arrays to be used throughout the program. The first few hold information about the screen display for the field boxes and the 'Next' and 'Prev' boxes:

row%	holds the rows on which the items are drawn
col%	holds the columns at which the titles will start
box%	holds the columns at which the boxes will start

title$	holds the title texts for the items
max%	holds the maximum lengths allowed for the contents of the items

The last four apply only to the fields:

strg$	holds the current field entries in the current record
strgt$	holds a temporary copy of strg$ used when loading
search$	holds the current search patterns
sbuf$	holds the pointers to the buffer for each field

Note that all these arrays are SHARED, so they may be accessed by the subprograms later in the program.

ScreenInit:
This starts by initialising two variables:

charhgt%	= height of each text character in pixels
search%	= 0 if a search pattern in operation
	= 1 if not

Then it loops round each of the items on the screen, prints its title in the correct place and calls a subprogram 'rect' to draw its box.

FileInit:
This block opens the file 'Address' on the disc in the disc drive 0 for random access. This creates the file if it doesn't already exist. The buffer length for this file is 116 bytes, as calculated above. The FIELD statement is used to point the elements of 'sbuf$' into the buffer. Note that this statement cannot be split between lines. Finally, this block calls the subprogram 'loadrec' to load the first record and display it on the screen.

MouseInit:
Next, the mouse event handler is set up and mouse event trapping is turned on.

MenuInit:
A menu is then created containing three entries, all of which are activated. Menu events are activated in the same way as the mouse events above.

MainBody:
The main part of the program consists of a never ending loop, within which the keyboard is scanned for input. Provided that a field has been selected, the character is acted upon as follows:

RETURN (ASCII 13)

1) Checks that the bottom field has not been reached and if not:
2) Updates the current record by placing the contents of the current field buffer into the current field entry.
3) Calls 'rect' to draw the box of the current field in white.
4) Increases the field number by one.
5) Calls 'rect' to draw the box of the new field in black.
6) Initialises the current field buffer to the null string.
7) Calls 'clearbox' to delete the contents of the box and move the text position to the start of it.

Del or ASCII 8 and 127

1) Checks that the start of the box has not been reached and if not:
2) Sets the current field buffer to be itself minus its last character
3) Moves the text cursor back one space, overtypes the next character by a space, and then moves the text cursor back again so that it is in the correct place.

Anything else

1) Checks that the maximum length of the current field has not been exceeded, and if not:
2) Prints the character, without a line return.
3) Adds the character to the end of the current field buffer.

rect:
This subprogram draws a box around the item whose number is passed as an argument. The colour to draw it is also passed, so that boxes can be selected or deselected using this routine.

trailrem:
This is another utility subprogram. It takes a string array as a parameter which it assumes to contain nine elements. For each of these elements it removes any trailing spaces. It does this by looking at the right-hand character and, if this character is a space, sets the element of the array to be itself minus the last character. This process is repeated until the last character is found not to be a space.

clearbox:
This subprogram has one parameter which is the number of an item. It moves to the character position at the start of the box for this item, prints the maximum number of spaces, which the item can contain, to clear the box, and then moves back to the start again so that any text entered will be printed in the correct place.

saverec:
This is the subprogram which deals with saving a record to the file. The record number to save is passed as a parameter. If the record number is zero (which is the search record) then nothing is done. Otherwise, it takes the contents of the current record, held in 'strg$', and uses LSET to place them in the buffer. Note that each string is extended to its maximum length by adding spaces onto the end.

The final field is treated differently. Although it is being held as a string for the sake of convenience, it is really a number. To show how numbers are dealt with, the string is first converted to a number using VAL and then this is converted back to a string suitable for outputting to the file. Note that this process does save memory. Instead of three bytes being used, one for each possible digit, MKI$ converts the integer into a two byte string, so each record is one byte shorter than it would otherwise be.

The exception to the conversion process is when the string is empty. Performing the above would result in the value 0 being stored to the disc, despite the fact that no number has been given. In this case, the code outputs a string containing two spaces so that the field will be empty, as expected, next time the record is viewed.

Once the buffer has been set up, the contents of it are written to the disc using PUT.

loadrec:
This performs the reverse process of loading a record. However, it is somewhat more complicated because it takes into account the search pattern set up when necessary. Its first parameter is the record number asked for. The next gives the direction in which the user is moving through the files:

dir% = 1 when moving forwards ('Next')
= 0 when moving backwards ('Prev')

The routine can be used to load a normal record or a search pattern. For the former it starts by initialising the end points for the search. Searching stops when it reaches the 51st record when moving forwards and 0th record when moving back. It then sets up two important flags:

ok% = 1 when current record matches the current search pattern
= 0 when it doesn't
endf% = 1 when the end of the file has been reached
= 0 when it hasn't

It then starts a loop which begins by trying to load a record into the buffer. If the record exists and so can be read, then the contents of it are assigned to the temporary record array 'strgt$'. Note that the reverse process to that in 'saverec' is applied. Trailing spaces are removed by calling 'trailrem' and the final field is converted from the stored string into a number and then into a normal string, provided that it doesn't just contain spaces.

If EOF returns the value TRUE, indicating that the record doesn't exist, then 'strgt$' is assigned null strings and 'endf%' is set to one.

The subprogram 'check' is then called. This checks to see if the contents of 'strgt$' match the current search pattern. 'ok%' is returned accordingly. Provided that the record matches, the loop will not be repeated. If the record is found not to match, the 'next' record (taking into account the direction) is subjected to the same treatment until either the end of the file is reached or the record limits are hit.

If a record is found to match, the heading 'RECORD' followed by the record number is printed, the contents of the fields are copied into 'strg$' and then entered into the field boxes. Note that, when the record number is printed, it is followed by an extra space. This ensures that the previous number is totally deleted. Otherwise, when moving from record 10 to record nine, the '0' would be left on the screen. Finally, another variable is assigned to:

```
rec%  = current record number
```

The next block of code deals with loading a search record. This prints the heading 'SEARCH' (again following it with spaces to delete any record number), sets the search fields to contain the null string and clears out the contents of the field boxes. The record number in this case is set to 0.

Then in all cases the following are initialised:

fld% = current field number (0 if uninitialised)
cur$ = current field buffer – holds the text of the field being edited

check:
This performs the check of the temporary record contents against the current search record, provided that 'search%' is set to one to that the search record is to be applied. The temporary record contents had any trailing spaces removed when they were assigned, so this subprogram starts by doing the same to the search record. It then sets 'ok%' to one, on the assumption that the record will match. Any non-matching fields set it back to 0.

For each of the first eight fields, it checks to see if the search string is empty. If not, then it looks to see if it contains a star. The simple case is when it doesn't. In this case, it checks the length of the search string against that of the actual record. If they are different, then the match automatically fails and 'ok%' is set to 0. If they are the same then it works through, comparing them one character at a time. Any difference again results in 'ok%' being set to 0. Note that the characters

are converted to upper-case before being compared. This ensures that the case of the text is irrelevant.

If the search string contains a star, then the position of it is noted. Provided that it is not at the left-hand end of the string, the characters to the left of it are obtained and a similar number of characters are read from the record entry. Then these are converted to upper-case and compared. Similarly, any text to the right of it is dealt with.

The final section of text deals with the last, numeric, field. For this field, the left-hand character of the search string is read. This must be a '<', '=' or '>'. Then the remainder of the string is converted into a number. The whole of the record entry is also converted into a number and these two numbers are compared.

menuhandler:
If an item from menu five, the Address menu, is selected, then any field which was being edited is updated in the current record and this record is then saved to disc by calling 'saverec'. Then the behaviour depends on the item selected:

Setsearch
1 Loads a search menu
2 Sets search% to 1

Clearsearch
1 Sets search% to 0

Quit
1 Closes the file
2 Terminates the program

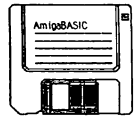
mousehandler:
Like the above subroutine, this starts by updating the current record if a field is being edited. However, this routine also performs the same action if it is the search record which is being set up. This is not necessary in the menuhandler since, whichever item is selected, the current search menu is not going to be needed. It is either going to be replaced, ignored or be made completely irrelevant.

Then the position of the mouse, at the time the button was pressed, is read and this is compared against all the boxes in turn to see which one is being selected.

In the case of the 'Next' or 'Prev' boxes, the first check made is to see if the maximum allowed record number (in either direction) has been reached. If not, the current record is saved if it has been altered and the next record (which matches the current search pattern if applicable) is loaded. Note that the check for whether or not a record has been altered is whether 'fld%' has the value 0. Whenever a record is read, fld% is set to 0. It then keeps this value until one of the field boxes is selected.

If it is one of the field boxes which has been selected, then any current field has its box redrawn in white, the value of 'fld%' is assigned the number of the field selected, then this box is highlighted and its contents cleared. Finally, the current field buffer is reset to the null string.

DataInit:
This contains the data statements holding the title and maximum length of each of the screen items.



11 : Managing Resources

This chapter looks at some of the more advanced features of AmigaBASIC which the authors of larger programs may find useful. It examines the way in which the memory of the computer is divided up and how this allocation can be altered. In addition, it suggests ways of reducing the amount of memory required by a program, in case you ever find yourself having problems squeezing one into the space available. Finally, it takes a look at how a single program can run 'background tasks' as well as its main activity, without having to think about it!

Linking Programs Together

It appears to be feature of programming that, however much memory a computer has, it is never enough. Sooner or later, you will write an application which will run out of space. Trying to reduce the size of an existing program is a time consuming activity. Therefore, if you are planning on writing a large application, it is worth thinking about designing it before you start, so that it makes the best use of the memory available.

For programs which are linear, this can be done by splitting them into separate smaller programs. Each of these smaller programs then needs to end by instructing BASIC to delete it, load the next program into memory and to start executing this new program. This technique is commonly used to create continuous demonstrations. These are made up of separate programs which show off a particular feature of the machine. When one has finished it starts the next one up, and so on in an endless loop. It could also be used for programs such as a bridge playing game. A game of bridge is split into four phases which occur sequentially: dealing the hand, bidding, playing and scoring. These phases are independent, apart from requiring certain pieces of information from the previous ones, such as the cards each hand contains.

Unfortunately, most programs cannot be split up in this way, since they usually contain a core which needs to be present at all times. However, a similar technique which involves using 'overlays' is very often possible. Overlays are sections of a program which are stored on disc and only loaded into memory when they are needed. This technique is most suited to programs which perform several separate actions. For example, with a database program the user can either be entering new information, searching through existing information, sorting records into order etc. Since these are separate activities which have little in common with each other, the code for them could be stored as overlays, so that only the one currently being used is in memory at a given time.

There are two keywords provided by AmigaBASIC which allow the processes described above to be carried out. These are CHAIN and COMMON.

In its simplest form CHAIN takes the syntax:

```
CHAIN "prog2"
```

where 'prog2' is the name of the program to be loaded and executed. In this example, prog2 will completely replace the current program. By using the MERGE option, a second program (which must have been stored as an ASCII file) can be added onto the end of the first. For example:

```
CHAIN MERGE "prog2"
```

In either case, you can specify the position at which BASIC is to start executing the program in memory after the CHAIN. The default is to start at the top. However, a numeric expression can be given and the result of this expression is used as the line number of the starting point:

```
CHAIN MERGE "prog2",start%
```

Note that alphanumeric labels cannot be used to determine the start position.

Sharing Variables Between Programs

It is possible to pass variables from the original program to the called one. However, the default is for the called program to behave as though it were a totally separate program and to not inherit the current values of any of the variables in the calling program. There are two methods of sharing variables between programs. The first is to use the ALL option in the CHAIN command:

```
CHAIN "prog2",,ALL
```

This causes all variables, other than those which are local to subprograms, to be passed to the program being called. If only a few variables are to be shared between the two, then the ALL option should be omitted, and instead, the calling program should use a COMMON statement to list the variables which are to be passed:

```
COMMON xpos%, ypos%, col%()
CHAIN "prog2"
```

Note that if variable types have been defined using DEFINT, DEFLNG etc in the calling program, then these definitions will need to be repeated in the called program, since variable types are not preserved. In addition, CHAIN without the MERGE option does not preserve the OPTION BASE setting.

For example:

```
REM Prog1
PALETTE 0,0,0,0
PALETTE 1,1,0,0
PALETTE 2,0,1,0
PALETTE 3,0,0,1
CLS
RANDOMIZE TIMER
col1% = INT(RND*2.99) + 1
FOR loop% = 1 TO 20
  xpos% = INT(RND*500) + 50
  ypos% = INT(RND*125) + 25
 LINE (xpos%,ypos%) - STEP(50,25),col1%,b
NEXT
```

```
COMMON col1%
CHAIN "Prog2"

REM Prog2
FOR loop% = 1 TO 20
  xpos% = INT(RND*500) + 50
  ypos% = INT(RND*125) + 25
  CIRCLE (xpos%,ypos%),50,col1%
NEXT
```

Enter and save each of these programs in turn. Then load 'Prog1' and run it. This draws 20 random squares in a random colour and then calls 'Prog2' and passes to it the number of the colour it has been using: 'col1%'. After a slight delay whilst 'Prog2' is being loaded from the disc, 'Prog2' is executed and produces 20 random circles in the same colour.

Overlays

Normally, when using MERGE, the second program will be an overlay. Therefore it needs to replace the current overlay. To do this a DELETE option can be added:

```
CHAIN MERGE "prog2",start%,ALL,DELETE start.overlay -
end.overlay
```

where start.overlay and end.overlay are the labels of the start and end of the section to be deleted. Therefore, it is necessary to ensure that all overlays begin and end with the same labels (or line numbers).

The following short program demonstrates how overlays may be used:

```
REM ProgMain
count% = 0

mainloop:
INPUT "Please enter two numbers :",num1,num2
op$ = ""
WHILE op$<>"+" AND op$<>"-" AND op$ <> "*" AND op$ <>
"/"
  INPUT "Enter operator (+ - * or /) :",op$
```

```
WEND
IF     op$ = "+" THEN
  ov$ = "ov1"
ELSEIF op$ = "-" THEN
  ov$ = "ov2"
ELSEIF op$ = "*" THEN
  ov$ = "ov3"
ELSEIF op$ = "/" THEN
 ov$ = "ov4"
END IF

CHAIN MERGE ov$,1,ALL,DELETE start.overlay - end.overlay

REM Start here after the CHAIN
1: CALL operate PRINT num1 " " op$ " " num2 " = " res

REM Jump to start of main body if the process
REM has been repeated fewer than 5 times
count% = count% + 1
IF count% < 5 THEN GOTO start
END
REM Dummy overlay to be replaced by relevant subprogram
start.overlay:
end.overlay:

start.overlay:
REM ov1
SUB operate STATIC
  SHARED num1, num2, res
  res = num1 + num2
END SUB
end.overlay:

start.overlay:
REM ov2
SUB operate STATIC
  SHARED num1, num2, res
  res = num1 - num2
END SUB

end.overlay:
start.overlay:
REM ov3
```

```
SUB operate STATIC
  SHARED num1, num2, res
  res = num1 * num2
END SUB
end.overlay:

start.overlay:
REM ov4
SUB operate STATIC
  SHARED num1, num2, res
  res = num1 / num2
END SUB
end.overlay:
```

To try out the example, create each of the four overlays 'ov1' to 'ov4' in turn and save each as an ASCII file. Then create, save and run the main program.

What this suite of programs is aiming to achieve is to be able to repeatedly read in two numbers and an operator and to print out the result of applying this operator to the numbers. The act of applying the operator to the numbers is carried out by calling a subprogram. Instead of the main program containing four different subprograms: one each for addition, subtraction, multiplication and division, it loads in whichever one it needs as an overlay and calls that one.

Ideally, we would like to think of the process as though the main program were resident continually and just the subprogram was being changed. Unfortunately, CHAIN does not act like that. After the CHAIN command, the program formed is treated as being a new program. However, we can go a long way towards making the process act as we would like. The 'ALL' option ensures that all the variables set up before the CHAIN takes place are preserved. In addition, by using the option to start executing the resulting program at a particular line number, we can make BASIC continue by executing the statement immediately after the CHAIN as though nothing had happened.

The only problem is that we cannot loop round the main body of code a certain number of times by using a FOR...NEXT loop. This is

because the FOR statement would be executed before the CHAIN and the NEXT statement after it. The CHAIN statement causes BASIC to forget all about any structures it may be in at the time. Therefore, when BASIC encountered the NEXT, it would have forgotten all about the start of the loop and give an error. The way the program has got around this problem is to use a label at the start of the main body of code to which it jumps if the counter of the number of times the code has been executed is less than five.

Memory Management

When running a BASIC program, the amount of memory you have available is divided into three distinct areas. These contain the 'stack', the 'heap' and the 'BASIC data'. These terms are explained below. The amount of memory assigned to each can be changed using the CLEAR command.

CLEAR used on its own erases all variables and arrays and closes any open files. In addition, it can be given one or two arguments which allocate space for the BASIC data and stack respectively. For example:

```
CLEAR, 50000, 1024
```

This allocates 50000 bytes for the BASIC data area and 1024 bytes for the stack. All the remaining memory is made available to the heap. The smallest value which can be assigned using CLEAR is 1024 bytes (1k). The default values are 25000 bytes and 4789 bytes respectively.

The Stack

The stack is used internally by BASIC to hold information about the flow of control of a program. For example, whenever BASIC encounters a GOSUB, function call or subprogram call, it has to remember its current position in the program so that it can return to the correct place after executing the appropriate body of code. This information is stored on the stack. Similar situations arise in the case of loops. The address of the start of each loop has to be stored so that when the end of the loop is reached, BASIC knows where to jump back to.

If a second subroutine call is made from within the body of another then a second address is placed on the stack. Then if the body of the second subroutine call contains a loop structure, the stack has a third address added to it. The stack continues to grow in this way until the innermost level is reached. However, when a subroutine etc ends, its return address is removed from the stack so the amount of stack space used decreases again.

This means that the amount of stack space required depends on how deeply you nest structures. It is independent of the size of the program or the number of independent structures the program contains.

BASIC Data Area

This area holds the text of the current program, all the variables which the program uses and the buffers for all the files it opens. Since the default size is only 25000 bytes, many programs will need to increase the size of this area using CLEAR. Even if a program is significantly less than 25000 bytes in length, it can soon use up the remaining memory by making heavy use of arrays, string variables or files. When this happens, an 'Out of memory' error will be generated.

If the actual programs themselves are greater than 25000 bytes in size then this poses a problem. There is no point in placing the CLEAR command inside the program because there isn't enough memory to load and run the program in order to execute the CLEAR command! The way to solve this is to use a small initialisation program as follows:

```
CLEAR 50000
CHAIN "MainProg"
```

If lack of memory starts to become a problem there are several things you can do to ease the situation:

If your program contains a block of code which is replicated, turn this into a subprogram and replace the occurrences of it by subprogram calls.

Consider splitting the program into segments which CHAIN each other, rather than having it all in memory together as one large program.

Reduce the buffer size for sequential files when opening them. Note that this may make your program run slower.

Keep array sizes down to a minimum and use the ERASE command to free the space they used as soon as you have finished using them.

Check that any numeric variables your program contains are of the minimum size. Remember that the default type is single precision which takes up four bytes. Often these are accidentally used as counters etc in loops when a two-byte short integer would do the job faster, as well as taking up less space.

Finally, as a last resort, cut down the comments in your code, remove blank lines and make variable names shorter. However, don't try to save memory by removing the indentation of structures. Indenting lines costs no extra memory, so 'un-indenting' them will have no effect other than making the program less readable.

The Heap

The heap is used for the screen display and sound buffers. Therefore, a program which creates new screens and windows requires plenty of heap space. Often, more memory is used for the screen display than for anything else, so the following points are worth noting:

Doubling the screen resolution, in either direction, doubles the amount of memory used by every window in it. Therefore don't automatically declare a screen to be in mode four – think carefully whether you actually need the extra resolution or not.

Similarly, the depth of a screen affects the memory significantly, so don't be too extravagant with colour.

Creating windows which will be redrawn automatically, when necessary, and which can change in size requires a buffer large enough to hold the whole screen to be made available. Therefore this should be avoided whenever possible.

BASIC doesn't have a heap of its own: it shares the system heap with other tasks. Therefore, you can make more heap space available for your programs by closing down any other applications which are running at the same time.

The FRE Function

This function returns information about the amount of memory being used. Its argument determines the particular area of memory as follows:

–1	Number of bytes free in the heap
–2	Number of bytes never used by the stack
any other number	Number of bytes free in the BASIC data area

Background Tasks

There is one final type of event which has not been mentioned so far in this book. This is the event which occurs after a certain length of time has passed.

You can turn time event trapping on or off by using TIMER ON, TIMER OFF or TIMER STOP. These act in the standard way. TIMER ON activates time event trapping. TIMER OFF turns the trapping off completely so that any events which occur are ignored. TIMER STOP still traps time events but does not react to them until a TIMER OFF statement is executed.

When time event trapping is turned on, you can instruct BASIC to call a particular subroutine whenever a set period of time has passed. For example:

```
ON TIMER(60) GOSUB minute
TIMER ON
WHILE INKEY$ = ""
WEND
END

minute:
BEEP
RETURN
```

The argument to the ON TIMER statement specifies the time interval you want in seconds. This can be any number between one and 86400 which is the number of seconds in 24 hours. The above program uses the value 60 to cause the subroutine 'minute' to be executed every 60 seconds. This subroutine simply beeps and flashes the screen. Press any key to stop the program running.

Timer events are principally used to enable secondary activities to be carried out at regular intervals. For example, the main activity of the following program is to draw a pattern. However, it also maintains a display of the time in a second window.

```
Init:
SCREEN 1,640,256,4,2
WINDOW 1,"Pattern",( 20, 20) - (500,220),16,1
WINDOW 2,"Time",   (520, 20) - (600, 40),16,1

PaletteInit:
PALETTE 0,0,0,0
FOR loop% = 1 TO 15
  shade = (15-loop%)/15
  PALETTE loop%,shade,shade,shade
NEXT

WindowSelect:
WINDOW OUTPUT 1
TimerInit:
ON TIMER(1) GOSUB clock
TIMER ON

DrawPattern:
col% = 1 : ang = 0.0
cenx% = 240 : ceny% = 100
FOR size% = 100 TO 10 STEP -1
  col% = (col% + 1) MOD 15 + 1
  ang = ang + .1
  xoff% = COS(ang)*size%
  yoff% = SIN(ang)*size%
  xpos1% = cenx% + xoff%*2
  ypos1% = ceny% + yoff%
  xpos2% = cenx% - yoff%*2
  ypos2% = ceny% + xoff%
```

```
    xpos2% = cenx% - xoff%*2
    ypos2% = ceny% - yoff%
    xpos2% = cenx% + yoff%*2
    ypos2% = ceny% - xoff%
    LINE (xpos1%,ypos1% - (xpos2%,ypos2%),col%
    LINE (xpos2%,ypos2% - (xpos3%,ypos3%),col%
    LINE (xpos3%,ypos3% - (xpos4%,ypos4%),col%
    LINE (xpos4%,ypos4% - (xpos1%,ypos1%),col%
NEXT
END

clock:
WINDOW OUTPUT 2
CLS
PRINT TIME$
WINDOW OUTPUT 1
RETURN
```

The program works as follows:

Init:
The program starts by creating a new screen which has a depth of four (which allows 16 colours) containing two windows. The only user option they have is that their contents will be refreshed after being covered. They cannot be resized, moved or closed.

PaletteInit:
Then the palette is set up. Colour 0 is defined as black. The other colours are set up so that they give a grey scale with colour one as white and colour 15 as the darkest shade of grey. The palette is set up like this, rather than using the more obvious method of having 0 as black and 15 as white with one to 14 giving intermediate values, so that the title bars will remain visible. These are displayed in colour one.

WindowSelect:
This selects the pattern window as the current window.

TimerInit:
This sets up a routine for handling timer events which are to occur every second and turns timer event trapping on.

DrawPattern:
Next comes the main block of the program which draws a pattern. It starts by selecting the next colour in the order 0, 1,...15, 0...and increases the rotation angle. For the current value of the rotation angle it calculates the x and y offsets using COS and SIN. Using these offsets it calculates the four corners of the box. Note that the x-coordinates are doubled to compensate for the pixels being roughly twice as high as they are wide. Finally, the box described by these four corners is drawn.

clock:
This subroutine is the timer event handler. It changes the window to the one containing the time, clears it to rub out the existing text, prints the current value obtained from TIME$, reselects the other window and returns. Note that WINDOW OUTPUT n is used to select the windows rather than WINDOW n. This is because WINDOW brings a window to the front as well as making it current. Doing so causes its contents to be cleared and refreshed which flashes the screen. The program is relying on the fact that both windows are visible at all times, so that all we need to do is make them current. This means that the pattern drawing continues happening as normal apart from a brief pause.

12 : Machine Code From Basic

Calling Machine Code Routines

This chapter is for those people who really want to get the ultimate in performance from their Amiga. It shows how 'machine code' routines can be used from BASIC. Machine code is the computer's own language - it consists entirely of numbers which are passed to the computer's central processing unit (CPU). Each of these numbers is treated by the CPU as either an instruction or data for an instruction.

There are two type of machine code routines which you can 'call', those you write yourself and those provided by the Operating System. This chapter deals with both. It starts by explaining what machine code is and how you can write it. Then it goes on to describe the procedures for accessing your own machine code routines from BASIC. Finally, it takes a brief look at what the OS provides and how you can access it.

Please note that what follows is only a very brief overview of machine code. The subject is enormous and you are referred to specific books on the subject. However, what follows should give you an insight into the topic and whet your appetite.

Machine Code

Machine code is the natural language of the computer. All the BASIC statements which you write have to be analysed by AmigaBASIC before your Amiga can understand them and execute them. The advantage of using BASIC is that the statements are easy to write and understand. The disadvantage is that the processing which AmigaBASIC has to perform as it runs your program takes a considerable time. This process of identifying and executing is called 'interpretation' and is undertaken by the part of AmigaBASIC called the 'interpreter'. Therefore, if you replace part or all of your program

with the equivalent machine code instructions, it runs many times faster.

The disadvantage with doing this is that machine code is definitely not easy to read and write!

The Central Processing Unit

The type of machine code which a computer can understand depends on what type of central processing unit (CPU) it contains. The CPU is the brain of the computer and has a significant effect on what the final performance of the computer will be like. Early home computers used relatively simple processors such as the Z80 and 6502. Today's micros are based on more powerful processors which means that they can carry out operations many times faster. You may think that speed isn't everything. However, it has consequences which you may not expect. For example, the higher the screen resolution you want to use, the more work the computer has to do to display your pictures for you. Unless the CPU is powerful enough, it just cannot cope with high-resolution graphics.

The processor which your Amiga contains is the Motorola MC68000, which is normally referred to as just 'the 68000'. It has become the most widely used processor in the home computer market.

Machine Code or Assembly Language

The CPU's function is to read a sequence of instructions from memory and carry them out. These instructions are simply a series of values stored in memory locations. The CPU will read the first 16-bit number of this series and interpret it as an operation identifier. In some cases this will be all the information it needs to enable it to act. It will then perform the operation as instructed and go on to fetch the next number and repeat the process.

In other cases it will need data as well: for example, if the instruction is telling it to jump to a different location (the machine code equivalent of a GOTO), it needs to know the address of the location. The CPU decides how many items of data it requires purely by looking at what operation identifier it has been given. It then goes and fetches the next one or more 16-bit numbers and interprets these as the data it needs.

Having executed the instruction, the CPU moves on to the next number which once again is treated as an instruction.

These values in memory are known as 'machine code'. A machine code instruction always starts with a number which is known as the operation code or 'opcode' since it defines the operation to be carried out. Then this may be followed by other numbers which are the data to be operated on called the 'operand'.

It is possible for you to program in machine code by simply building the required list of numbers in memory. However, this is an extremely difficult process. Unless you could remember the opcodes for all the different instructions you wanted to use, you would have to thumb through reference books and work out what they were. This process is further complicated by the fact that the description given above is an over simplification of the actual truth. Of the 16 bits in the opcode only four bits are used to determine the kind of operation to be performed. The others provide more information about it, such as the 'size specifier' and the 'register' it is affecting etc (more about these later). Therefore you have to work out all the separate pieces which reflect each item of information held and combine them together in a particular order to generate the actual opcode value.

Writing such a program would be bad enough, but debugging it would be a nightmare. At first sight it would be virtually impossible to distinguish the opcodes from the data.

To ease the task, several companies have produced 'assemblers' for the Amiga. These allow you to produce statements in a more readable form known as 'assembly language statements' using mnemonics to represent the different instructions etc. The assembler program can then be 'run' which makes it convert the mnemonics into the corresponding machine code. When people talk about a routine being 'written in machine code', what they normally mean is that it was written in assembly language and then converted into machine code using as assembler. Other terminology you may come across is 'source code' which is the assembly language routine the assembler takes as its input and 'object code' which is the machine code produced by it.

Although assemblers from different sources are not identical in all respects, the mnemonics they use will be the same since they are part of the definition of the 68000 instruction set. Hence, the examples that follow in this chapter should apply, whichever system you buy.

A Very Brief Overview

One of the fundamental differences between BASIC and assembly language is that BASIC instructions act on variables, whereas assembly language ones act on data in memory or on internal memory locations called 'registers'.

You can include as many variables as you like within a BASIC program. These can be used to handle different types of data: strings, integers and floating point numbers. When producing machine code, however, you are limited to just 16 registers. Eight of these are data registers called D0, D1, D2, D3, D4, D5, D6 and D7, which each hold a 32-bit value. The other eight are address registers called A0, A1, A2, A3, A4, A5, A6 and A7 for holding the addresses of memory locations.

The way in which you have to get around the problem of having a limited number of registers to work on is to store all the values you are using in blocks of memory and load them temporarily into registers to work on them. For example, to add two numbers together, you would load the first number from an address in memory into one data register, load the second into another data register and then use the assembly language instruction ADD to add one to another.

The other problem is that you don't have different types of registers for storing different types of data. Fundamentally, everything you act upon in assembly language is an integer. This is fine for characters, since they can be manipulated in their ASCII format. However, what about floating point numbers? The practical answer is don't use assembly language if you want to deal with floating point numbers. Although it is ideal for handling integers, it provides no facilities for floating point values.

You normally get a choice about what size of integer you are working on; these can contain either 8-bits (a byte), 16-bits (a word) or 32-bits (a long word). The standard 32-bit register is used for holding all types.

However, the mnemonic for the instruction is followed by a letter indicating what size of operand it is to act upon:

B Byte
W Word
L Long Word

The omission of a size specifier normally indicates that words are being used. For example:

```
ADD.W      D0, D1
```

and:

```
ADD        D0, D1
```

both add the bottom word of the value in register D0 to the value stored in register D1.

It is possible in certain cases to use constant values in assembly language statements. For example:

```
ADD.W      #500, D0
```

adds the value 500 to the contents of register D0. The '#' is used to indicate that what follows is a constant. You can specify the values in hexadecimal notation by adding an '&H' as follows:

```
ADD.W      #&H100, D0
```

This adds the hexadecimal constant 100 (256 in decimal notation) to the contents of register D0.

When values are being read from or written to memory, the actual location being used is determined by an 'address'. This is a 32-bit number which uniquely identifies a particular location. The address to be used can be calculated by combining registers and numbers in different ways, each method being known as an 'addressing mode'. This is easier to explain by means of examples using the MOVE instruction. As its name implies, this instruction moves values

between registers and/or memory addresses. It is one of the most widely used 68000 instructions.

Instruction	Description of value moved into D0
MOVE D1,D0	Contents of register D1
MOVE (A0),D0	Contents stored at the address held in A0
MOVE D1(A0),D0	Contents stored at the memory location whose address is given by the value held in A0 plus an offset of the contents of D1

The above should be enough to indicate roughly what producing machine code entails. If you feel you want to try it out for yourself then the next step is to equip yourself with a book describing the 68000 instruction set and a good assembler. These together should teach you about the full list of instructions provided and allow you to experiment with them.

Accessing Machine Code From AmigaBASIC

The first stage in the process of accessing machine code from AmigaBASIC is to develop the machine code program itself. The following is a short assembly language program which will be used for this demonstration:

```
MOVEM.L    D0/A0,-(A7)
MOVE.L     12(A7),D0
MOVEA.L    16(A7),A0
BRA        test loop
EORI.B           #&H20,0(A0,D0.W)test
DBRA             D0,loop
MOVEM.L    (A7)+,D0/A0
RTS
```

This program, when converted into machine code, can be executed from BASIC using the CALL statement. The first instruction stores the contents of the 68000's registers D0 and A0 onto the stack, which is indicated by A7. The stack pointer is decremented so that it again points to the next free position (this is specified by the '-' sign which precedes it).

The next instruction again references the stack using an offset of 12 bytes to load a word into register D0. This offset of 12 is used since there are now three long words (ie 12 bytes) of data which have been pushed onto the stack since the call was made. Two of these are the registers D0 and A0 pushed by the first instruction and the other is the return address. Thus the offset of 12 skips over all of these and loads what was the first stacked word immediately before the call was made. This will be the first argument to the CALL statement. Similarly, the third instruction fetches the second argument to the CALL statement from the next highest stack position using an offset of 16. This value is placed in register A0.

Thus, D0 contains the first parameter and A0 contains the second. These should be the length and the address of the string to be processed.

The program now enters the processing loop by branching to the end test of the loop using the BRA (branch always) instruction to move to the instruction following the label 'test'.

The main body of the loop consists of a single instruction. This operates on the data in the string one byte at a time. Each byte is loaded from the address held in A0 plus the position count in D0 and is Exclusive-ORed with the value 32 (&H20). This will convert upper-case letters into lower-case letters and vice-versa. In fact, the last character of the string is the first to be processed since D0 initially holds the length of the string.

We now come to the DBRA instruction. This decrements register D0 by one and if the result is not -1 branches back to the label 'loop', thus causing the next byte to be processed. Once the count in D0 has reached zero, there is nothing more to be done and so this instruction will not cause a branch back and so the program goes on to the next instruction. This is the reverse of the original store onto the stack. It retrieves the original values of A0 and D0 and resets the stack pointer in the process.

Finally, the program returns to the caller via an RTS instruction.

When this program is assembled, it produces the following machine code:

```
48E7 8080
202F 000C
206F 0010
6000 0008
0A30 0020 0000
51C8 FFF8
4CDF 0101
4E75
```

This needs to be inserted into a BASIC shell program from which it can be called. For example:

```
DIM code %(16)
FOR I% = 0 TO 15
READ code%(I%)
NEXT
A$ = "Hello"
B$ = "World"
C$ = A$+B$
length = LEN(C$)
address& = SADD(C$)
start = VARPTR(code%(0))

CALL start (length&,address&)
PRINT C$
END
DATA &H48E7,&H8080
DATA &H202F,&H000C
DATA &H206F,&H0010
DATA &H6000,&H0008
DATA &H0A30,&H0020,&H0000
DATA &H51C8,&HFFF8
DATA &H4CDF,&H0101
DATA &H4E75
```

This program starts by setting up an array 'code%' which is large enough to hold the 32 bytes of machine code. The program then extracts the machine code from DATA statements using `READ code%(I%)` inside a FOR loop. The first machine code instruction must be placed in `code%(0)`.

The program then sets up a string 'C$' and two numeric variables 'length&' and 'address&' which, as their names suggest, hold the length of C$ and the address in memory at which it is stored. Also a variable 'start' is created using the function VARPTR to find the address in memory of the very first element of the array which contains the machine code.

Next, the machine code program is called by using CALL with this address and passing the length and address of the string. When the final RTS instruction of the machine code is executed, control passes back to BASIC and the program ends by printing out the converted string which should appear as follows:

```
hELLOwORLD
```

Operating System Access

Besides calling your own machine code, you can also access the machine code contained in the Operating System. The AmigaDOS Operating System consists of a number of separate libraries, each of which can be accessed using a special keyword 'LIBRARY' which is provided by AmigaBASIC. This 'opens', ie makes available all the routines contained by, a library. Up to five libraries can be opened at any time.

The LIBRARY CLOSE statement is provided to close down libraries once they are no longer needed.

Before a library can be accessed in this way, a special file has to be produced which contains a list of the routines inside the library, what arguments they require and where in the library they live. A utility program, 'ConvertFD', is provided in the BASICDemos directory of the Extras disk to produce such a file for you.

For example, one of the libraries is called 'Intuition'. This contains all the commands for operating the Amiga's user interface. To create the appropriate file for this, run the ConvertFD utility program giving the file name:

```
Extras:FD1.2/Intuition_lib.fd
```

for the existing library file and:

```
Extras:BasicDemos/Intuition.bmap
```

for the file to be created. This produces a .BMAP file whose name can be used in conjunction with the LIBRARY statement, for example:

```
LIBRARY "Extras:BasicDemos/Intuition.library"
```

Once a library has been opened in this way, the various routines within it can be called using the CALL statement, for example:

```
CALL DisplayBeep&(0)
```

This calls the routine inside the Intuition library which makes the screen flash.

To make full use of the LIBRARY statement, you need access to details of the various libraries which exist and the routines within them. This is beyond the scope of this book. However, many other books are available which provide this information.

13 : Devices

This chapter covers a mixture of subjects. The first, how to deal with discs, should be of use to all users. It brings together all the information you require to format discs, create directories, move around the directory structure, etc. The others are relevant to those with extra hardware attached to their Amiga, eg joysticks and printers. These sections explain the commands which BASIC provides to support these extra items of hardware. In addition, they show how different devices can be used for general purpose input and output.

Using Discs

On the Amiga, if you wish to keep any programs, data files, pictures, etc, for later use, you do so by saving them onto a disc. As you accumulate more and more information, it becomes very important to organise it correctly. Otherwise, finding a specific file again later becomes a tedious process. This organisation has two stages. The first is deciding what to keep on each disc. The second is how to arrange the information on an individual disc.

This section goes through the stages involved in preparing a new disc so that it is ready for storing your files and setting up the different drawers within it so you can organise the information neatly.

Formatting a Disc

When you save a file onto disc, it writes out all the individual bits of information onto 'tracks' or magnetic grooves on the surface of the disc. However, when you buy a blank disc, these tracks aren't present. This is because different makes of computers expect different numbers of tracks and different spacings between the tracks etc. Therefore disc manufactures have chosen to make general purpose products which can be 'formatted' to work with any machine. The process of formatting a disc lays down the tracks in the way in which a particular machine is going to expect to find them.

You have to format discs while you are running the Workbench; you cannot format them while you are 'inside' the BASIC system. It is best to do so before you enter BASIC so you have a spare disc ready should you choose to use it.

Once you are in the Workbench, eject the current disc from the built in disc drive and replace it by the blank one which you wish to format. Because it is blank, the Workbench won't be able to read information from it and so will give it the name 'DF0:BAD'. Then, click on the disc icon and select Initialize from the Disk Menu.

You will be prompted to insert the Workbench Disk. Then, a few moments later you will be asked for the blank disc back again. At this stage the requester:

```
OK to Initialize disk in
drive DF0:
(all data will be erased) ?
```

will appear. This is giving you a final chance to change your mind. In this example, you have no reason to do so; therefore you should click on the Continue gadget.

The requester is really provided for a different situation: that is when you are trying to re-format a disc which already contains data. This process totally destroys all the files which you have on the disc so you should be very careful when carrying it out. Accidentally deleting one file is bad enough but losing a whole disc full can be disastrous!

When the initialisation process has started, the window shows which track is currently being formatted or 'verified' and how many are left to go. The total number of tracks placed on the disc is 80: these are numbered 0 to 79. After it has formatted each track, the Amiga goes through the process of 'verifying' or checking it. Normally, this will just be a routine process. However, you may occasionally find that you get an error message displayed. If this happens you should try repeating the formatting process. Often, the problem will then go away. It it doesn't, then your disc is probably faulty and you should throw it away and start again with a different one.

When the disc has been verified, the final phase is for some control information to be written out to it. While this is happening you will be warned not to remove the disc. After this has finished, your disc will finally ready to use.

Naming a Disc

The first thing you should do when you have formatted a disc is to give it a name. The default one which the Amiga gives it after formatting it is 'Empty'. To change this, click on the disc icon and select Rename from the Workbench Menu. This will display an input line on the screen. Point at this, delete the existing name using the DEL key and then type the name you want to use, for example 'BASICProgs1', and press RETURN. The name of the disc will then change.

The file name you choose can contain up to 30 characters. These characters can be a mixture of upper and lower case letters, numbers and the underline character '_' which is useful for separating words.

Copying BASIC Across

It is a good idea to have a copy of AmigaBASIC on each of the discs which you want to use to store programs. This means that if you only have a single disc drive then you won't need to keep swapping discs in the middle of a BASIC session. To do this, double-click on the 'BASICProgs1' disc icon to display the window for this disc. Then, eject the disc from the disc drive and replace it with the Extras Disk and click on the Extras icon. When the Extras window is displayed, move the mouse pointer to the AmigaBASIC icon, press and hold down the left mouse button, drag the icon to the BASICProgs1 disc window and release the mouse button. This will commence the copying and you will be asked to swap discs several times while the process is being carried out.

Creating Drawers

Now is the time to create 'directories' or 'drawers' so that you can organise the information on the disc in a sensible fashion. An example of the sort of structure you should aim to create is shown in the diagram shown in figure 13.1

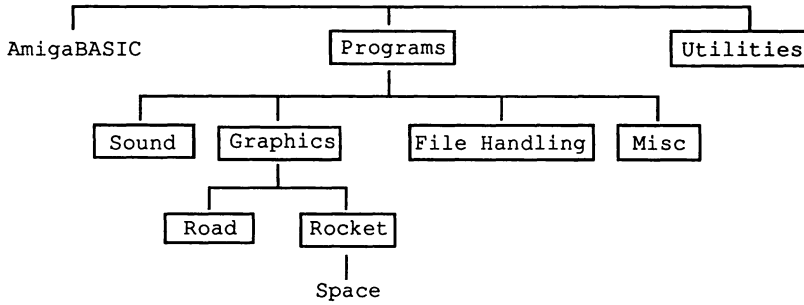


Figure 13.1. Part of a directory structure

Note that you can create directories within other directories.

The easiest way to create a directory is to copy the Empty drawer from the Workbench menu to where you want it and then rename it.

The Current Directory

If you try to load a file from within the AmigaBASIC system by giving just its file name, then the 'current directory' will be searched for the file. Normally the current directory is the top level directory which in our example above contained just AmigaBASIC itself. However, you can change it by using the command `CHDIR`. For example:

```
CHDIR "Programs"
```

will set the current directory to 'Programs'. You can then use the command a second time, to set the current directory to one of the directories contained in 'Programs', ie 'Sound', 'Graphics', 'FileHandling' or 'Misc', for example:

```
CHDIR "Graphics"
```

Then you could use it again to choose between 'Road' or 'Rocket':

```
CHDIR "Rockets"
```

Alternatively, you can give the whole path in one go, separating the directory names using a '/', for example:

```
CHDIR "Programs/Graphics/Rocket"
```

The above demonstrates how to use CHDIR to move down the directory structure, to a directory which is below the current one. We also need to be able to move up again. To do this, type:

```
CHDIR "/"
```

This will take you back up to the previous level. For example:

```
CHDIR "Programs/Graphics/Rocket"
CHDIR "/"
```

will set the current directory to 'Graphics'.

A shortcut is to type:

```
CHDIR ":"
```

this will take you directly to the top level directory again. One further point to note about CHDIR is that you can set the current directory to be on a different disc by providing the disc name:

```
CHDIR "BASICProgs2:Programs/Speech"
```

Then, whenever you give a file name, it will look for it on the disc which is named 'BASICProgs2' and ask you to insert this disc into the drive if necessary. Note that the disc name is separated from the directories by a ':'.

Finally, if you cannot remember where in the directory structure you are or what files you have in the current directory then type:

```
FILES
```

This will start by giving the current directory name and will then list the entire contents of that directory, including the names of any other directories which it contains.

Providing Pathnames

As an alternative to loading a file from the current directory, you can provide the whole 'pathname' when you give the LOAD command. A 'pathname' is the full name of a file. It includes the file name, the directory containing the file and the list of directories between this directory and the top-level directory. These are given in the order you would have to move through them on the way down to find the file, for example:

```
LOAD "Programs/Graphics/Rocket/Space"
```

This will ignore the current directory setting and search down the directory structure given to load the file 'Space'. Similarly, this applies when you are saving files using SAVE and listing files using FILES.

If you wish, you can specify the disc to use here as well, for example:

```
FILES "BASICProgs2:"
```

This will list all the files in the top level directory of the disc whose name is 'BASICProgs2'. Alternatively, you can state that you wish to use the disc which is currently in a particular disc drive, for example:

```
SAVE "DF0:Programs"
```

will save a file onto the disc currently in drive DF0 (the internal disc drive) in the directory 'Programs' which will be found at the top level.

Acting on Files

There are a couple of other BASIC commands which are important for managing files. The first of these is NAME which can be used to change the name of an existing file. For example:

```
NAME "Space" AS "Space_old"
```

This changes the name of the file 'Space' in the current directory to 'Space_old'. This is the sort of thing you may want to do as you are developing a program. It allows you to keep the current version,

'Space_old', as a backup. After you have made further edits to the program currently in memory, you then save it as 'Space' as normal. However, if you make a mistake, you still have the previous version to return to.

The other, very important command is KILL. As it names suggests it gets rid of files. For example:

```
KILL "Space_old"
```

will delete the file 'Space_old'.

Making Backups

When you received your Amiga, you were advised to take copies of the original Workbench and Extras discs. This means that if anything happens to the versions you are working with, you still have the originals available and can make new copies to work with.

The same applies to your own program discs. You should regularly make duplicates of the discs which you are working on. If you have an important piece of software which you are developing, then it isn't sufficient to keep a couple of versions of it on the same disc. Discs can and do develop faults and if this happens you could well end up not being able to access any of the information on the disc.

The following is provided to remind you how to copy a disc. First, enter the Workbench and select Duplicate from the Workbench Menu. A requester will appear asking you to place the disc you want to copy into drive DF0. When you have done this, select Continue. Next, you will be asked to replace the disc to be copied with the disc you want to copy the files to. This should either be a new, freshly formatted disc or one which contains files which aren't important and which you don't mind losing. Finally, continue swapping these two discs as requested until the process is complete.

Printers

One of the commonest and most useful 'extras' which can be added to an Amiga system is a printer. If you want to use your computer for

word processing then obviously it is vital – what is the use of letters which can only be stored on disc and which cannot be printed out and sent? But even if you aren't interested in using your Amiga for word processing then a printer is still handy to have around.

Within AmigaBASIC there are two ways in which you can profit from one. The first of these is that it allows you to print the results of your programs onto paper. The other is that it allows you to obtain a printed listing of your programs. How these can be achieved is demonstrated below.

Sending Output to a Printer

Previous chapters have shown how to send output to the current output window using the commands PRINT and PRINT USING. To send output to a printer, two corresponding commands exist: LPRINT and LPRINT USING. These take the same syntax as their screen counterparts.

LPRINT is the simpler of the two to use and is often all that you will require. It takes a list of string or numeric expressions which are to be printed separated by punctuation characters. These characters determine how the expressions will be positioned. For example, if two expressions are separated by a semi-colon ';' they will be printed immediately adjacent to each other. Conversely, if they are separated by a comma ',' the second will be printed starting at the next zone as determined by the current setting of WIDTH.

In the case of numeric expressions, the resulting number will always be followed by a space. In addition, positive numbers will be preceded by a space whereas negative ones will be preceded by a minus sign. They will be output in decimal format if they can be represented accurately with seven or fewer digits (16 or fewer for double precision values) and in exponential format otherwise. LPRINT on its own will print a blank line.

LPRINT USING is more complicated but it provides you with greater control over how things are formatted. It is particularly useful for tables of numbers since it allows you to state exactly the number of digits before and/or after the decimal point, whether plus or minus

signs are to be printed etc. The reference section at the end of this book covers all the options in detail.

Using the Printer's Features

Besides supplying the text which you want to appear on the paper and formatting information, you can also send the printer instructions on how to output it. The sort of thing you can do is to tell the printer to change to italic or bold scripts, to use a different character set or to set the margin or tab positions etc. These instructions are known as 'printer escape codes'. This is because they all start with the Escape character (ASCII value 27).

The Amiga provides a set of standard printer escape codes which you can use, whatever type of printer you have attached. These will then be translated into the particular sequences required for your printer. (You should have already told the Amiga which type of printer you have attached by using the Preferences tool in the Workbench. If you have not done so, follow the instructions outlined in the Introduction to the Amiga manual supplied with the machine.)

By supplying a standard set of printer escape codes, Commodore have ensured that you can send a printer instructions without having to know what type it is. If the printer connected supports the operation you asked for, then it will be passed the sequence of commands it recognises to perform the operation. Otherwise, your instruction will be ignored and have no effect. Therefore you cannot do any harm by asking a printer to do something which it is not capable of.

The full list of standard printer escape codes are listed in Appendix A. The following short example illustrates their use:

```
REM Select bold
LPRINT CHR$(27);"[1m";
REM Send line of text you want in bold type
LPRINT "bold text";
REM Convert back to ordinary type
LPRINT CHR$(27);"[22m";
REM Now print ordinary text
LPRINT " followed by ordinary text"
```

Note that you can use combinations of the features together. For example:

```
REM bold on
LPRINT CHR$(27);"[1m";"bold only"
REM underline on
LPRINT CHR$(27);"[4m";"bold and underline"
REM bold off
LPRINT CHR$(27);"[22m";"underline only"
REM underline off
LPRINT CHR$(27);"[24m";"back to normal"
```

Printed Listings

The simplest way of listing part or all of a program to a printer is to use the LLIST command. Its syntax is similar to that of LIST which you have been using to display a program in the list window. For example:

```
LLIST
```

will take the program which is currently in memory and send the whole thing to the printer, whereas:

```
LLIST l1
```

will print out the program starting from the label or line number l1. Finally:

```
LLIST l1-l2
```

will print out just the part of the program between label or line number l1 and label or line number l2.

A printed listing of a program is commonly referred to as a 'hard copy'. If you are having problems making a program work correctly, producing a hard copy of it can often help. One advantage is that it allows you to see the whole program in one go, rather than just a screen full. Hence, you can easily compare lines from the top of the program with lines at the bottom. In addition, some people find it easier to concentrate on a printed listing than on a screen display. Whether this is the case or not, the two are definitely different. So the

bug which you have been searching for and not noticing on the screen can often 'jump out at you' from the printed page.

Joysticks

Joysticks are particularly useful for game play. They can be used to control the movement of a particular object either up, down, left or right and normally provide a 'fire' button for additional action. The Amiga can accept input from either one or two joysticks which plug into the sockets labelled 1 Joystick and 2 Joystick at the back of the computer. Note that one of these is normally occupied by the mouse, so to use two joysticks at once you will have to unplug the mouse for a while.

BASIC provides two functions which return information from the joysticks. These are STICK which returns information about the direction of movement of the joystick and STRIG which returns information about whether or not the button has been pressed.

STICK takes one argument which determines which joystick you want to investigate and whether you are interested in movement in the X or Y direction:

Argument N	Meaning
0	Investigate X movement of joystick 1
1	Investigate Y movement of joystick 1
2	Investigate X movement of joystick 2
3	Investigate Y movement of joystick 2

The value returned is either –1, 0 or +1 which have the following meanings:

Return Value	Meaning
-1	Movement upwards or to the right
0	No movement
1	Movement downwards or to the left

STRIG takes one argument whose meaning is as follows:

Argument N	Meaning
0	Investigate if button 1 was pressed since last call.
1	Investigate if button 1 is currently pressed.
2	Investigate if button 2 was pressed since last call.
3	Investigate if button 2 is currently pressed.

Hence the values 1 and 3 investigate the current status of the button on one of the joysticks whereas the values 0 and 2 investigate whether a button press has occurred since the previous time STRIG(0) or STRIG(2) was used. The values 0 and 2 therefore prevent button presses being lost due to them occurring when the program is not checking for them.

The value returned by STRIG is as follows:

Return Value	Meaning
0	Button is not / has not been pressed
1	Button is / has been pressed

Input and Output Devices

When dealing with files, the Amiga needs to know where these files are kept. By default, it assumes that they are on disc in the current directory. However, you can direct the the Amiga to output information to or input it from other places by using a 'device name'.

We have come across one device name already. That is 'DF0:', the internal disc drive. Preceding a file name by this device allows you to save files etc to the disc in a particular drive rather than to the current disc.

The others work in a similar manner. For example 'PTR:' and 'LPT1' are both similar and can be used to select the printer device for output. Giving the command:

```
LIST ,"PTR:"
```

lists the program in memory to the attached printer and is therefore equivalent to:

```
LLIST
```

Another device which can be selected for output is 'SCRN:'. Any data sent to this device will appear in the current Output window. For example:

```
LIST ,"SCRN:"
```

lists the program in the Output window rather than the List window.

A common device to choose for input is the keyboard, which has the device name 'KYBD:'. For example, if you give the instruction:

```
OPEN "KYBD:" FOR INPUT AS 1
```

then any subsequent INPUT$ statement using filenumber 1 will take its input from the keyboard. This is demonstrated by the following simple filehandling program:

```
OPEN "KYBD:" FOR INPUT  AS 1
OPEN "SCRN:" FOR OUTPUT AS 2
char$ = INPUT$(1,1)
PRINT#2,char$
CLOSE 1,2
```

This selects the keyboard for input and the screen or more specifically the Output window for output. It then attempts to read a single character from the input device and therefore waits for you to press a key. The character input is not automatically reflected to the screen by the INPUT$ command. However the PRINT# statement reflects it for you by sending the character input to output purpose 2, ie the screen. Finally, the program tidies up by closing the output purposes.

One final device which is worth mentioning is 'COM1:' which can be used to send information to or read it from the 'serial port' or more strictly speaking the 'RS232 interface'. This is a standard interface which is supplied on most computers and thus allows different machines to be linked together by a single cable. They can then communicate with each other by sending information backwards and forwards between them. The obvious use for this is to transfer data files or even programs from one machine to another when other

methods won't work, for example when they have incompatible disc formats.

The following program provides an example program for receiving data on an Amiga from a different computer. The sending program should start by transmitting a series of 'X' characters. As soon as the Amiga notices one of these it will reply by sending back the characters 'HELLO' followed by a carriage return. This tells the sending program that the Amiga end is ready to receive the real data so it can start to send it.

The data is expected to consist of sets of four ASCII values, each value representing a character '0' – '9' or 'A' – 'F'. Therefore, each set of four values provides information about a word (ie a 16-bit number). For example the hexadecimal number &H1234 should be represented by the four values: ASC("1"), ASC("2"), ASC("3") and ASC("4") which should be sent in reverse order ie low-byte first. Therefore the data transmitted to send this word should be:

```
52
51
50
49
```

Once they have been received, they will be converted back again to form the original word.

The first set should represent the total number of words to be transmitted, ie the number of sets of four ASCII values. This should be followed by the data itself. Finally, one last set should be sent which represents the 'checksum' for the data. A checksum is a value which somehow represents all the pieces of information sent and allows a check to made that the data received is correct. In this case, the check sum is obtained by XORing all the words together.

Provided that the check sum sent is identical to the check sum calculated from the data received, the Amiga will create the file "object" in the current directory and output all the words of data received to that file.

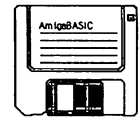
```
CLS
PRINT "Listening ..."

OPEN "COM1:19200,N,8,2" FOR INPUT AS #1
OPEN "COM1:19200,N,8,2" FOR OUTPUT AS #2

a$ = "."

WHILE a$<>"X"
a$ = INPUT$(1,#1)
WEND

REM as$ is the first "X" seen

PRINT "HELLO"

PRINT #2,"HELLO"
PRINT #2,CHR$(13)

REM Strip any "X" chars which overran

b$ = INPUT$(1,#1)
WHILE b$="X"
  b$ = INPUT$(1,#1)
WEND

c$ = INPUT$(1,#1)
d$ = INPUT$(1,#1)
e$ = INPUT$(1,#1)
num% = VAL("&H"+e$+d$+c$+b$)

PRINT "Expecting ";num%;" words."

DIM code%(num%)

check&=0

FOR I = 1 TO num%
  b$ = INPUT$(1,#1)
  c$ = INPUT$(1,#1)
  d$ = INPUT$(1,#1)
  e$ = INPUT$(1,#1)
       code%(I) = VAL("&H"+e$+d$+c$+b$)
```

```
PRINT ;"..";
check& = (check& XOR code%(I)) AND &HFFFF
NEXT

b$ = INPUT$(1,#1)
c$ = INPUT$(1,#1)
d$ = INPUT$(1,#1)
e$ = INPUT$(1,#1)
crack& = VAL("&H"+e$+d$+c$+b$)

PRINT
PRINT "My check ="HEX$(check&)
PRINT "Yr check ="HEX$(crack&)

IF crack& = check& THEN
  PRINT "Data checks OK"
  PRINT "Sending to object file"
  OPEN "object" FOR OUTPUT AS #3
  FOR I = 1 TO num%
       word$ = RIGHT$("0000"+HEX$(code%(I)),4)
       lowbyte$ = CHR$(VAL("&H"+RIGHT$(word$,2)))
       hibyte$      = CHR$(VAL("&H"+LEFT$(word$,2)))
       PRINT #3,hibyte$;lowbyte$;
  NEXT
ELSE
  END
END IF

CLOSE
```

The main thing to note about this program is that the 'COM1:' device can take up to four parameters. The first of these is the 'baud rate'. This is the rate at which characters are received. The value of 19200 used is the highest possible rate. The other possible ones are 9600, 7200, 4800, 3600, 2400, 1800, 1200, 600, 300, 150 and 110. The next is either an 'O', 'E' or 'N'. This determines the 'parity' which is the mode for checking data as it is received. The values stand for 'odd', 'even' and 'none'. In this case no parity check is being made. The third value is the number of bits in each byte which contains information. The possible values are 5, 6, 7 or 8. In this example the full eight bits are being used each time. The final value is the number of 'stop bits'.

Essentially this is the size of the gap left between the individual bytes of information. The possible values are 1 and 2. The program reads the bytes of data sent in sets of four and converts them back into the word which they represent. As each word is obtained, it is stored in an integer array and its value is combined with the current check sum.

After all the data has been received, the check sums are compared. Then if everything is OK, the values are removed from the array and output to the file as two separate bytes: the high byte first (top eight bits) followed by the low byte (bottom eight bits).



Appendix A: Command Reference

Introduction

This section provides a thorough and comprehensive index of all AmigaBASIC's keywords, and should prove invaluable when you are programming. The commands are arranged alphabetically and each description follows an identical format, again for ease of reference.

After the command name the command syntax is given and this is followed by an example of the command use. Where necessary a short program is included to provide you with a better understanding of the command's operation. This is in turn followed by a brief one sentence description of the command under one heading 'Brief' and is then followed by a more detailed description, which will invariably explain the full syntax of the command. Finally, and where appropriate, a list of associated keywords is given.

The following conventions are used in the syntax description:

< > Angle brackets are used to hold compulsory parameters.

[ ] Square brackets are used to hold optional parameters.

... These dots indicate that more parameters of the same format may be included as required.

ABS

Syntax: ABS <expression>

Example:

```
LET A=-123
PRINT ABS(A)
B=ABS(A)
```

Brief

Returns the absolute value of the expression supplied.

Description

The expression following the command is evaluated and any negative sign is stripped to provide a positive or absolute value. The expression may be a formula, a variable or a number.

Associated: SGN

AREA

Syntax: AREA [STEP] (X,Y)

Example:

```
COLOR 3,0
AREA (100,150)
AREA (400,150)
AREA (120*RND, 150*RND)
```

Brief

Defines one of a series of points describing a polygon to be drawn with AREAFILL.

Description

The AREA statement in effect defines a corner of a multi-sided shape. Up to 20 such corners can be defined. The space which lies within the series of corners can then be filled-in with the AREAFILL command.

The STEP argument is optional. If STEP is included then X and Y will be taken as offsets from the current position of the graphics cursor (ie relative to it). X and Y may be expressions for evaluation.

Associated: AREAFILL

AREAFILL

Syntax: AREAFILL [0/1]

Example:

```
COLOR 3,1
DIM pat%(1)=$HFFFF
PATTERN, pat%
AREA (20,20)
AREA STEP (0,50)
AREA STEP (50,0)
AREAFILL 0
```

Brief

'Draws' the multi-sided shape as defined by previous AREA commands.

Description

This command displays, by drawing and in-filling, the shape defined by the previous two or more (up to a maximum of 20) AREA statements. The command may be followed by either 0 or 1 which defines how AREAFILL acts:

0 Pixels corresponding to bits in the pattern which are set are coloured in the foreground colour. Pixels corresponding to bits which are cleared are coloured in the current background colour. Fills the area with a pattern defined by the PATTERN statement.

1 Pixels corresponding to bits in the pattern which are set are inverted. Pixels corresponding to bits which are cleared are left unaltered. Inverts the area being filled according to the pattern defined by the PATTERN statement.

Mode 0 is the default mode of operation.

Associated: AREA, PATTERN, COLOR

ASC

Syntax: ASC(<string$>)

Examples:

```
LET name1$="Sharron"
LET surname$="Fellows"
PRINT ASC(name1$)
v=ASC(surname$)
PRINT v
```

Brief

Returns the ASCII number of the first character in a string.

Description

Returns the ASCII code number for the first character held in the named string. In the above example the ASCII code for 'S' would be returned from name1$, ie, 83.

Amiga characters represented above the normal range of ASCII characters, ie those with code numbers in the range 128-255, also would be given to this function and the appropriate value returned.

Associated: CHR$

ATN

Syntax: ATN(X)

Example:

```
A=0.5
PRINT ATN(A)
PRINT ATN(-1.2)
```

Brief

Returns the arctangent of value X in radians.

Description

The arctangent of the argument supplied is returned in radians and is in the range $-\pi/2$ to $\pi/2$ radians. The evaluation is performed in either single or double precision as indicated by the argument supplied to the function.

BEEP

Syntax: BEEP

Example:

```
PRINT "Wake up there!"
BEEP
```

Brief

Sounds a beep and flashes the screen.

Description

This command makes the Amiga emit a short and simple beep. In addition the screen display is flashed once. BEEP is the equivalent of printing CHR$(7).

BREAK ON/OFF/STOP

Syntax: BREAK ON/OFF/STOP

Example:

```
BREAK ON
ON BREAK GOSUB handleit
INPUT ANSWER$
BREAK OFF
:
handleit:
PRINT "You tried to exit"
RETURN
```

Brief

Enables, disables or halts BREAK event trapping.

Description

If BREAK ON is executed then programs may be halted by the user Amiga-period, CTRL-C or selecting Stop on the Run Menu. This is called break event trapping. BREAK OFF disables BREAK ON and therefore the event trapping.

BREAK STOP suspends the event trapping. Events are noted, but execution of an ON...BREAK GOSUB sequence (see example above) does not occur until after a BREAK ON statement has been executed.

Associated: ON BREAK

CALL

Syntax: CALL <name>[<argument list>]

Example:

```
CALL calculate (x,y,(z))
CALL code(VARPRT(x))
CALL string(SADD(a$))
```

Brief

Calls either an AmigaBASIC subprogram or a machine code routine/library routine.

Description

1) BASIC: control is passed to the subprogram defined by SUB and named by name. Any arguments given are assigned to the variables contained in the subprograms parameter list. Simple variables and array elements are passed by reference unless they are enclosed in brackets in which case they are passed by value.

   When used to call a BASIC subprogram the CALL statement itself is optional and the subprogram may be called by name alone. in which case the brackets around the arguments must also be omitted.

```
SUB TWONUMS (a%,b%)STATIC
c%=a%+b%
PRINT c%
END SUB
CALL TWONUMS (3,4)
TWONUMS 4,5
```

2) MACHINE CODE: Control is passed to a machine code program located at the address held in the named variable. An argument list may be given to pass information to the machine code routine. Both string and numeric data may be supplied but it is the address of the string or number that is passed and not the data itself.

String addresses are passed using the SADD function and variable addresses using the VARPTR function:

```
CALL code (VARPTR(a))
CALL code (SADD(a$))
```

3) LIBRARY: Control is passed to machine code routines that have been attached to AmigaBASIC. See LIBRARY for details and example.

Associated SUB, VARPTR, SADD

CDBL

Syntax: CDBL <NUM>

Example:

```
PRINT CDBL (X!+Y!)
```

Brief

Converts a numeric argument into a double precision number.

Description

The function takes the argument which may be any type of integer or floating point number and returns the corresponding double precision number.

For instance:

```
X!=1234
Y!=100000
PRINT (X!*Y!)
PRINT CDBL(X!*Y!)
```

would respond:

```
1.234E+08
123400000
```

the first result being single precision, the second being double precision.

Associated: CINT, CLNG, CSNG

CHAIN

Syntax:

CHAIN [MERGE],<file> [,[expression],[,[ALL],[,DELETE<range>]

Example:

```
CHAIN "PROG2"
```

Brief

Loads and runs another program with or without passing current variables to it.

Description

The program specified by <file> is loaded over the current one and run. By using the MERGE option a subroutine may be loaded and 'overlaid', ie merged with the current program to become part of it rather than replacing it. The program to be MERGEd must be an ASCII file.

The <expression> option allows a line number (but not a label) to be nominated as the starting position in the called program. By default, the execution starts at the first line.

The ALL option ensures that every variable, other than local variables, are passed to the named program. If ALL is omitted then only those variables listed in a COMMON statement will be passed to the incoming program.

The DELETE option allows a <range> of lines to be deleted from the calling program to make way for it. This may be to delete a previous overlay. <range> may be given in line numbers or labels separated by a hyphen.

Note:

- A comma must be used if ALL is used but the <expression> is not.
- CHAIN leaves files open.
- CHAIN turns event trapping off. If it is required it should be re-enabled by the incoming program.
- The current OPTION BASE is not altered if MERGE is used.

- Variable types are not preserved unless MERGE is used.

Associated: COMMON, MERGE, SAVE

CHDIR

Syntax: CHDIR <string>

Example:

```
CHDIR "df1:c"
```

Brief

Changes the current directory, ie CHange DIRectory.

Description

The <string> is taken to be the new device and/or directory path that is to become the current directory.

CHDIR "/" moves to the top level directory.

CHR$

Syntax: CHR$<num>

Example:

```
PRINT CHR$(12)
FOR L=65 TO 127
PRINT CHR$(L)
NEXT L
PRINT CHR$(7)
```

Brief

Returns the character whose ASCII code is supplied.

Description

CHR$ evaluates <num> and returns the ASCII character it represents. It performs the opposite task to ASC.

Its main use, in conjunction with PRINT, is for sending control characters to the screen.

In the above example, the Output Window is cleared with CHR$(12) and the ASCII character set from 65 to 127 is printed. A BEEP is then issued through CHR$(7).

Associated: ASC

CINT

Syntax: CINT(<num>)

Example:

```
PRINT CINT (5.23)
```

Brief

Converts <num> into an (16-bit) integer by rounding any decimal portion.

Description

CINT expects a number in the range -32768 to 32767 and will round it to the nearest whole integer value. If the number is not within the specified range an "Overflow" error will occur.

Note that numbers with a .5 decimal value will be rounded towards zero if <num> is even and rounded away from zero if <num> is odd. Thus:

```
PRINT CINT (-35.5)
PRINT CINT (42.5)
PRINT CINT (345.21)
```

will return:

```
-36
42
345
```

Associated: CLNG, CDBL, CSNG, FIX, INT

CIRCLE

Syntax: CIRCLE [STEP] (x,y),radius [,color [,start,end [,aspect]]]

Example:

```
CIRCLE (50,50),500
CIRCLE (220,100),100,3,-1.57,4.71
```

Brief

Draws a circle, arc or ellipse.

Description

As a minimum CIRCLE expects three values. The first pair, within parentheses, give the x and y coordinates of the centre of the circle to be drawn. These are followed by the radius of the circle. x, y and radius are specified in pixels.

The inclusion of STEP prior to the x and y co-ordinates will identify that x and y are to be taken as positions relative to that of the current position of the graphics cursor.

<color> defines the colour to be used for drawing the circle and corresponds to the color number as defined by PALETTE. If no value is given then the default foreground colour is used.

Arcs may be drawn by specifying the <start> and <end> angles in radians. The range is $-2*\pi$ to $2*\pi$, and the table below gives values in terms of degrees which should help when constructing arcs.

Angle in degrees	Angle in radians
0	0
45	0.79
90	1.57
135	2.36
180	3.14
225	3.93
270	4.71
315	5.50
360	6.28

If either <start> or <end> are given as negative values, the circle or ellipse is connected to the centre point, so that a segment rather than an arc is displayed and the absolute value is used to determine the angle.

<aspect> provides the display aspect ratio, which is the ratio of the width to the height of a single pixel. This is effectively a calibration figure which allows true circles to be drawn. The aspect ratio of monitors varies and so CIRCLE will draw a true circle if <aspect> is set to the aspect ratio of the monitor in use.

CLEAR

Syntax: CLEAR [,<basicdata>] [,<stack>]

Example:

```
CLEAR , 30000
```

Brief

Erases all variables, strings and arrays, shuts any open files and optionally re-allocates memory.

Description

In its basic form this command will erase the contents of all variables and arrays by setting them to zero and will erase all strings setting them to a null string, "". In addition CLEAR closes all files and resets any DEF statements.

In addition, two optional parameters may be passed to CLEAR. <basicdata> is a numeric expression, 1024 or greater, which defines the amount of memory in bytes to be allocated to AmigaBASIC for holding the program text, variables and file blocks. In the above example 30000 bytes were allocated.

<stack> is a numeric expression, 1024 or greater, which defines the amount of memory in bytes to be allocated to AmigaBASIC the system stack.

The remaining memory is made available to the heap.

Associated: FRE

CLNG

Syntax: CLNG <num>

Example:

```
PRINT CLNG (5.23)
```

Brief

Converts <num> into an (32-bit) integer by rounding any decimal portion.

Description

CLNG expects an a number in the range –2147483648 to 2147483647 and will round it to the nearest whole integer value. If the number is not within the specified range an "Overflow" error will occur.

Note that numbers with a .5 decimal value will be rounded towards zero if <num> is even and rounded away from zero if <num> is odd.

Associated: CINT, CDBL, CSNG, FIX, INT

CLOSE

Syntax: CLOSE [[#]<filenumber>[,[#]<filenumber....]]

Example:

```
CLOSE #2
```

Brief

Closes one or more files.

Description

CLOSE acts in opposition to OPEN. It closes any number of open files, as defined by specifying their <filenumber> handle. A file may be closed for any of the following reasons:

- To write the contents of the buffer to a sequential the file after the final PRINT# etc.
- To update file information with regards to length etc.
- Free the file number for use by another file.
- To enable the file to be opened again, eg for read access after it has had write access.

CLOSE on its own closes all open files.

Associated: CLEAR, END, NEW, OPEN, STOP, SYSTEM

CLS

Syntax: CLS

Example:

```
CLS
PRINT "Screen ready for action"
```

Brief

Clears current Output Window.

Description

The command erases the contents of the current Output Window and places the text cursor in the top left hand corner of the window. The CLS command only affects the current Output Window.

COLLISION

Syntax: COLLISION (<object id>)

Example:

```
test=COLLISION (0)
```

Brief

Tests for object collision.

Description

COLLISION is a function which returns information about object collisions. If the parameter <object id> is a number greater than zero then it is taken as an OBJECT.SHAPE object identifier. The value returned is a number indicating either the id of a second object which it collided with, or a negative value indicating that the object collided with a window border. The four borders are represented as follows:

–1	Top border
–2	Left border
–3	Bottom border
–4	Right border

The values 0 and –1 may be passed as the <object id> and these have specific functions:

0 Returns the id number of the object that has collided with a second object. The information is not removed from the collision queue (see below). This value can then be passed as a parameter to a second call to find what it collided with.

–1 Returns the window number in which the collision occurred. Collisions are added to the collision queue as they occur. The queue is limited to 16 collisions and, once full, all subsequent collisions are ignored and not added to the queue.

Associated:

OBJECT.SHAPE, COLLISION ON/OFF/STOP, ON COLLISION

COLLISION ON/OFF/STOP

Syntax: COLLISION ON/OFF/STOP

Example:

```
COLLISION ON
ON COLLISION GOSUB x
```

Brief

Enables, disables or suspends collision event trapping.

Description

COLLISION ON sets BASIC into detective mode and it actively looks to see when a collision occurs.

COLLISION OFF has the opposite effect to ON and stops the computer from looking for collisions.

COLLISION STOP will still detect collisions but will not act on them, ie execute a ON COLLISION...GOSUB statement until such stage that COLLISION ON is re-issued.

Associated: COLLISION, ON COLLISION, OBJECT.SHAPE etc.

COLOR

Syntax: COLOR [foreground] [,background]

Example:

```
COLOR 1,0
```

Brief

Sets foreground and background colours.

Description

The COLOR (use of 'colour' is not permissible) command allows the foreground and background colour numbers to be set. The foreground colour determines the colour in which text and graphics are drawn, while the background colour determines the screen colour.

If no COLOR statement is issued, AmigaBASIC uses color 0 for the background and colour 1 for the foreground.

The PALETTE command (and the settings in Preferences) defines the colours allocated to each colour number. The default values are:

0 Blue
1 White
2 Black
3 Orange

COMMON

Syntax: COMMON <variable list>

Example:

```
COMMON coords, axis, apex, facts(), name1$
CHAIN "Prog2"
```

Brief

Passes the named variable(s) to a chained program.

Description

COMMON is used in tandem with a CHAIN statement, though they may appear anywhere within a program and do not have to be tied together. A list of named variables, in any order, is given after the command, each separated by a comma.

The variables appearing in the list will be passed to the incoming chained program.

Points to note:

- The same variable must not appear in more than one COMMON statement.
- Array variables are identified by the use of parentheses at the end, ie array().

It is good practice to place all COMMON statements at the start of your program to ensure against mistakes and to allow ease of checking.

Associated: CHAIN

CONT

Syntax: CONT

Brief

Restarts program execution following a forced interruption.

Description

The command continues program execution after one of the following has occurred:

CTRL and C keys pressed
Amiga and full stop keys pressed
STOP statement in program

The program continues from the point where it was interrupted.

Note that CONT may not be used if the program has been edited since the interrupt.

COS

Syntax: COS(<num>)

Example:

```
cosval=COS(0.524)
PRINT COS(x+y)
```

Brief

Returns the cosine of <num>.

Description

The cosine of <num> is evaluated in the precision of the value supplied, ie a single precision value is evaluated as a single precision number and a double precision value is evaluated as a double precision number. The value of <num> is expected in radians.

CSNG

Syntax: CSNG(<num>)

Example:

```
X%=1234
PRINT CSNG (X%)/3
```

Brief

Converts a numeric argument into a single precision number.

Description

The function takes the argument, which may be any type of integer or floating point number and returns the corresponding single precision number.

Associated: CDBL, CINT

CSRLIN

Syntax: CSRLIN

Example:

```
line=CSRLIN
```

Brief

Reads the current text line number.

Description

This function returns the approximate line number of the text cursor.

The lines start at 1 for the top line of the current output window and increases by one for each line down. The value is only approximate if a non-standard font is being used, in which case the number returned is based on the height of the character 'O' in this font

Associated: POS, LOCATE

CVD

Syntax: CVD(<8-byte string>)

Example:

```
OPEN "ex4" AS 1 LEN=8
FIELD #1, 8 AS sbuf$
GET#1,1
PRINT CVD(sbuf$)
```

Brief

Converts a eight-byte random access file string into a double precision number.

Description

This function returns a double precision number from a string created using NKI$. This pair of functions is normally used for converting short integers to strings to be saved in random access files and then converting them back to numeric values when they are read from the file.

Associated: MKD$, VAL

CVI

Syntax: CVI(<2-byte string>)

Example:

```
OPEN "ex1" AS 1 LEN=2
FIELD #1, 2 AS sbuf$
GET#1,1
PRINT CVI(sbuf$)
```

Brief

Converts a two-byte random access file string into a short integer.

Description

This function returns a short integer from a string created using MKI$. This pair of functions is normally used for converting short integers to strings to be saved in random access files and then converting them back to numeric values when they are read from the file.

Associated: MKI$, VAL

CVL

Syntax: CVL(<4-byte string>)

Example:

```
OPEN "ex2" AS 1 LEN=4
FIELD #1, 4 AS sbuf$
GET#1,1
PRINT CVL(sbuf$)
```

Brief

Converts a four-byte random access file string into a short integer.

Description

This function returns a long integer from a string created using MKL$. This pair of functions is normally used for converting long integers to strings to be saved in random access files and then converting them back to numeric values when they are read from the file.

Associated: MKL$, VAL

CVS

Syntax: CVS(<4-byte string>)

Example:

```
OPEN "ex3" AS 1 LEN=4
FIELD #1, 4 AS sbuf$
GET#1,1
PRINT CVs(sbuf$)
```

Brief

Converts a four-byte random access file string into a single precision number.

Description

This function returns a single precision from a string created using MKS$. This pair of functions is normally used for converting single precisions to strings to be saved in random access files and then converting them back to numeric values when they are read from the file.

Associated: MKS$, VAL

DATA

Syntax: DATA <constant list>

Example:

```
DATA 1,2,3,4,5
DATA "Mercury", "Venus", "Earth"
```

Brief

Stores constants which can be assigned to variables using READ statements.

Description

DATA are non-executable statements that are used to hold constant information for use by the READ statement. DATA statements may be used anywhere in a program and may contain as many constants as will fit on the line. String constants must be enclosed with quotes if the string contains commas, colons or significant spaces at either end.

DATA statements are read in order and a special read pointer is used by AmigaBASIC to keep the position of the next item to be read. This may be reset using RESTORE. The <constant list> may contain any mixture of numeric constants. See READ for an example of DATA.

Associated: READ, RESTORE

DATE$

Syntax: DATE$

Example:

```
PRINT DATE$
```

Brief

Returns the current date.

Description

The DATE$ function reads the system clock and returns a ten character string in the American form: `mm-dd-yyyy`

A typical response on New Year's Eve 1999 would be:12-31-1999

Associated: TIME$

DECLARE FUNCTION

Syntax: DECLARE FUNCTION <id> [(parameter list)] LIBRARY

Example:

```
DECLARE FUNCTION Cube%() LIBRARY
CB%=Cube%()
```

Brief

Causes AmigaBASIC to search all open libraries for the function <id> whenever it is subsequently used.

Description

This statement allows you to call (later on) a machine code routine residing in a library that returns a value (ie the machine code routine is itself a function). The machine code function must be contained within a library that is already opened. The '<id>' itself is any valid identifier signified by the appropriate trailing declaration characters, namely, %, &, ! and #. Obviously the appropriate identifier must be chosen to match the incoming value. In the example given above the function Cube%() is identified and then called with the 16-bit integer value returned placed into CB%.

Note that <parameter list> is a list of the parameters for the function. However, this list is, in fact, ignored by AmigaBASIC but can be given for documentation purposes.

Associated: LIBRARY, CALL

DEF FN

Syntax: DEF FN <name> (<parameter list>)=<Func def>

Example:

```
DEF FNaddtwo(T)=T+2
DEF FNhi$(d$)="HI "+UCASE$(d$)
newval=FNaddtwo(4)
L$=FNhi$("Sharron")
PRINT newval, L$
```

Brief

Defines a user function.

Description

Allows user-definable functions to be constructed in AmigaBASIC. <name> must be a legal variable name and the function is called when FN<name> is used within the program.

<parameter list> is optional and contains a list of variables and/or constants to be passed to the function. Values passed must be of the same type as the parameters they are being passed to, ie strings cannot be passed to integer variables. Variables must be separated by commas.

<Func def> is an expression which is limited to one line. It defines the action of the function.

In the above example 'newval' is assigned the value '6' ('4' is passed to FNtwoadd where it has '2' added to it), and L$ is assigned the string 'HI SHARRON' ('Sharron is passed to FNhi$ where it is converted to upper case and preceded by "HI".

Note:

- Functions cannot be called until the appropriate DEF FN has been executed, ie so that AmigaBASIC knows it is there!
- If a DEF FN <name> is specified twice then the latest definition is used.
- DEF FN only applies to the program in which it is defined.

DEFDBL

Syntax: DEFDBL <letter range>

Example:

```
DEFDBL a-c,z
```

Brief

Defines that variables starting with the given letters be treated as double precision.

Description

When issued AmigaBASIC assumes that all variables beginning with the letters in the specified range are to be treated as double precision. However if a type designator such as $ is used then this takes priority.

In the above example all variables (and arrays) beginning with the letters a-c and z (ie, a,b,c and z) which don't have a type designator are treated as double precision variables.

DEFINT

Syntax: DEFINT <letter range>

Example:

```
DEFINT a-c,z
```

Brief

Defines that variables starting with the given letters be treated as short integers.

Description

When issued AmigaBASIC assumes that all variables beginning with the letters in the specified range are to be treated as short integers. However if a type designator such as $ is used then this takes priority.

In the above example all variables (and arrays) beginning with the letters a-c and z (ie a,b,c and z) which don't have a type designator are treated as short integer variables.

DEFLNG

Syntax: DEFLNG <letter range>

Example:

```
DEFLNG a-c,z
```

Brief

Defines that variables starting with the given letters be treated as long integers.

Description

When issued AmigaBASIC assumes that all variables beginning with the letters in the specified range are to be treated as long integers. However if a type designator such as $ is used then this takes priority.

In the above example all variables (and arrays) beginning with the letters a-c and z (ie a,b,c and z) which don't have a type designator are treated as long integer variables.

DEFSNG

Syntax: DEFSNG <letter range>

Example:

```
DEFSNG a-c,z
```

Brief

Defines that variables starting with the given letters be treated as single precision variables.

Description

When issued AmigaBASIC assumes that all variables beginning with the letters in the specified range are to be treated as single precision variables. However if a type designator such as $ is used then this takes priority.

In the above example all variables (and arrays) beginning with the letters a-c and z (ie a,b,c and z) which don't have a type designator are treated as single precision variables.

Note that, by default, variables are treated as single precision values anyway.

DEFSTR

Syntax: DEFSTR <letter range>

Example:

```
DEFSTR a-c,z
```

Brief

Defines that variables starting with the given letters be treated as string variables.

Description

When issued AmigaBASIC assumes that all variables beginning with the letters in the specified range are to be treated as string variables. However if a type designator such as % is used then this takes priority.

In the above example all variables (and arrays) beginning with the letters a-c and z (ie a,b,c and z)which don't have a type designator are treated as string variables.

DELETE

Syntax: DELETE [line no/label] [- [line no/label]]

Examples:

```
DELETE 200-300
DELETE start
DELETE info-
DELETE -endofdata
```

Brief

Deletes program lines in the specified range.

Description

Allows one or more lines to be deleted from a program. DELETE on its own deletes the whole program. If just a single line no/label is given on its own (with no hyphen) then just that line is deleted. A range of lines can be deleted by specifying the start and end line numbers or labels separated by a hyphen. If either option is omitted then lines are DELETEd either from the start of the program to the specified line/label or from the specified line/label to the end of the program.

DIM

Syntax: DIM [SHARED] <variable list>

Examples:

```
DIM AB(93)
DIM SHARED dates%(12,2)
```

Brief

Dimensions named arrays.

Description

Named arrays are created and storage space allocated for them as specified in the array subscript variable. Arrays may be multi-dimensional to a maximum of 255 dimensions, subject to enough memory being available.

In the first example above, a one-dimensional array called AB is defined as having 94 elements (0-93). In the second example a two dimensional array called dates% is declared as having 13*3 elements.

Note that the OPTION BASE command can be used to specify a minimum array subscript value of 1 rather than 0. If this had been issued previously AB and dates% would then contain 93 (1-93) and 12*2 elements respectively.

If SHARED is included in the definition then the variables are made globally accessible to the entire program; as such the DIM SHARED statement must be used within the main program (and not within a subprogram). When an array is dimensioned, the following actions take place:

- Space is allocated according to subscripts.
- All elements are set to zero or the null string.
- If the array has already been dimensioned a "Re-dimensioned array" error is issued.

By default, AmigaBASIC will automatically set the array size to 10 elements if the array is not declared before use.

```
DIM TEST (10) : REM this statement is optional
FOR N=0 TO 10
TEST(N)=N
NEXT N
FOR N=0 TO 10
PRINT TEST(N)
NEXT N
```

Associated: SHARED, LBOUND, UBOUND, OPTION BASE

END

Syntax: END

Brief

Terminates execution of a program.

Description

END statements may be placed anywhere within a program. When AmigaBASIC encounters one the programs, operation is terminated and all open files are closed.

If the END statement is omitted, termination occurs when the bottom of the program is reached.

Associated: STOP

EOF

Syntax: EOF <file number>

Example:

```
REM assume file #1 open for input
WHILE NOT EOF(1)
  INPUT#1, surname$
  PRINT surname$
WEND
CLOSE #1
```

Brief

Tests for the end of an open file.

Description

When data is being read from a file of unknown length EOF can be used to see if the end of file has been reached.

For a sequential file, EOF returns -1 (true) after the last item has been read. For a random access file, EOF returns true if the last GET failed to read a whole record. Otherwise EOF returns 0 (false) indicating that there is still information left unread in the file.

ERASE

Syntax: ERASE <array> [,<array>]

Example:

```
ERASE DataArray, NumArray
```

Brief

Erases the named array from AmigaBASIC memory.

Description

The contents of the named array(s) are destroyed and the memory allocated to the array(s) by AmigaBASIC is freed. The array(s) may be re-dimensioned after they have been ERASEd.

Associated: DIM, CLEAR

ERR

Syntax: ERR

Example:

```
ON ERROR GOTO traperror
:
traperror:
IF ERR=52 THEN PRINT "Bad File Number"
RESUME NEXT
```

Brief

Returns error number of most recent error.

Description

ERR will return the code number of the previous error and this can be used to identify the error so that appropriate action can be taken.

Associated: ON ERROR, ERL, ERROR

ERL

Syntax: ERL

Example:

```
ON ERROR GOTO traperror:
:
traperror:
PRINT "Error at line :"ERL
RESUME NEXT
```

Brief

Returns line number at which an error occurred.

Description

The line number at which an error has occurred will normally be returned by ERL. If there is no line number, then ERL will return the number of the first numbered line that precedes the error. If no numbered line precedes the line containing the error then 0 is returned.

Associated: ON ERROR, ERR

ERROR

Syntax: ERROR <integer>

Example:

```
ON ERROR GOTO errortrap
IF A$<>"Y" THEN ERROR 200
 :
 errortrap:
 IF ERR=200 THEN PRINT "Press 'Y' to continue!"
RESUME NEXT
```

Brief

Generates an error with a particular number.

Description

ERROR allows you to simulate the occurrence of an existing error condition (by giving an error number already defined) or to create your own error (by giving an error number not used by AmigaBASIC). The number must be in the range 1- 255. Note that if you define your own error number but do not trap the error with a suitable error handler then AmigaBASIC will issue an appropriate error message and stop execution.

Associated: ON ERROR, ERL, ERR

EXP

Syntax: EXP <num>

Example:

```
PRINT EXP(n*2):epn=EXP(1)
```

Brief

Returns e to the power of <num>.

Description

The natural logarithm of e (2.718282) to the power of <num> is calculated and returned.

An 'overflow' error message will be generated if <num> is greater than 88 or 709 for single or double precision numbers respectively but execution will continue and the highest number representable in the given precision will be returned.

Associated: LOG

FIELD

Syntax: FIELD [#] <file No.>,<field length> AS <string>

Example:

```
FIELD #1, 15 AS prenom$, 25 AS surname$
```

Brief

Makes space for variables in random file buffer.

Description

FIELD allows you to make space in a data record buffer for incoming data strings. <file No.> defines the file number where the information is to be written (as defined when the file was OPENed); <field length> defines the length of the field (in characters) which then has the string name, <string>, assigned to it.

In the above example 15 spaces are allocated to the first field (prenom$) and 25 spaces to the second field (surname$).

The total number of spaces used in a FIELD statement must not exceed the record length specified when the file was originally opened – the default length of which is 128 characters.

Note that a field variable name should not be assigned to, other than by using LSET or RSET to place strings in the buffer. Assigning it a normal string value using LET etc will mean that it no longer 'points' into the buffer.

Note also that each FIELD statement in a program starts assigning the string variables to space in the random file buffer starting at the beginning. Therefore giving two FIELD statements such as:

```
FIELD #1, 20 AS a$
FIELD #1, 20 AS b$
```

would just provide alternative string variable names pointing to the same area of the buffer. Therefore the whole record description must be given on one line.

Associated: GET, LSET, OPEN, PUT, RSET

FILES

Syntax: FILES [<string>]

Example:

```
FILES "c"
FILES "df0:"
```

Brief

Catalogues all the files in the specified directory.

Description

<string> is evaluated and may contain a drive number and/or directory path. The contents of the specified directory are then listed – it is assumed that a disc is in the drive specified. If no <string> is provided then the current directory is catalogued.

FIX

Syntax: FIX <num>

Example:

```
PRINT FIX (-12.34)
```

Brief

Returns the integer portion of a number.

Description

FIX truncates the number supplied and returns the integer portion only. It does not round numbers up or down in the manner of INT.

For example:

```
PRINT FIX(-12.34)
PRINT FIX (23.45)
```

returns:

```
-12
23
```

Associated: CINT, INT

FOR...NEXT

Syntax: FOR<control variable>=<expr1> TO <expr2> [STEP <expr>]

NEXT[<control variable>] [,<control variable2>]...

Examples:

```
FOR L=0 TO 10 STEP 2
PRINT L
NEXT
:
A=3
B=5
FOR N=A TO (B*2) STEP 0.5
PRINT N
NEXT N
:
FOR X=10 TO 2 STEP -1
PRINT X
NEXT X
:
FOR A=1 TO 10
 FOR B=1 TO 5
   FOR C= 2 TO 4
PRINT A,B,C
NEXT C,B,A
```

Brief

Executes a loop a set number of times.

Description

FOR and NEXT mark the boundaries of a loop in which everything is executed a given number of times. The number of repetitions is defined by the first line. The <control variable> is initially assigned the value of <expr1>. Provided that it is less than <expr2> the loop is executed. When the NEXT is executed, the value of <control variable> is incremented and the test is performed again. When the value of <control variable> finally equals or exceeds that of <expr2> the statement after the NEXT is jumped to. <expr1> and <expr2> may be numbers, variables or expressions to be evaluated. The use of STEP is

optional and this can alter the rate at which the <control variable> is adjusted. If STEP is omitted then a STEP rate of one (+1) is assumed. Note that the stop size may be negative in which case the loop is repeated until <control variable> <= <expr2>

FOR...NEXT loops may be nested within other FOR...NEXT loops provided the NEXT statement for the inner loop concludes the loop before the NEXT of the outer loop, and that all loops have unique control variable names.

Associated: WHILE...WEND

FRE

Syntax: FRE <-1>/<–2>/<0>

Examples:

```
PRINT FRE (-1)
PRINT FRE (-2)
PRINT FRE (0)
```

Brief

Returns number of free bytes.

Description

FRE returns information concerning the amount of free space, in bytes, in specified areas of memory. The argument supplied identifies the area thus:

–1	Bytes free in system.
–2	Bytes not used on stack.
other	Bytes free in AmigaBASIC memory.

Associated: CLEAR

GET

Syntax: GET [#]<file no.>[,<record no.>]

GET (x1,y1)-(x2,y2), <name> [,(index[,(index...])]

Examples:

```
OPEN "example" AS 1 LEN 10
FIELD #1, 10 AS sbut$
GET #1, 1
PRINT sbuf$
CLOSE

DIM rect%(626)
GET (120,90) - (200,140), rect%
```

Brief

Reads a record from a random access file or reads an array of bits from the screen.

Description

GET has two quite separate functions and these are examined separately below.

Random Access File GET
Here the command reads the record whose number is given by <record no.> from the specified file, defined by <file no.>. If no <record no.> is given then the next record in the sequence is taken. The largest possible <record No.> is 16777215.

After the command has been executed the contents of the buffer may be read, using the transfer variables defined by FIELD and, if necessary, the string to numeric converter CVD, CVI and CVS.

Screen GET
Here the command allows a section of screen image to be read into a data array, thus making it a useful way in which to transfer graphic images.

The area to be read by GET is defined by specifying the upper left hand corner and lower right hand corner of an imaginary rectangle

around the area of screen to be read. This is (x1,y1)-(x2-y2) in the syntax above. <name> is the name of the array that is to be used to hold the image, which may be of any type except string.

The size of the array (in bytes) required to hold the image can be calculated as follows:

```
height*2 INT ((width+15)/16)*D+6
```

where height=y2-y1+1, width=x2-x1+1, and 'D' is the depth of the screen, for which 2 is the default value.

The bytes per array element are:

2 bytes for a short integer array
4 bytes for a long integer or single precision array
8 bytes for a double precision array

The first six bytes of the array will be used to hold the following information about the saved image:

1st pair : width
2nd pair : height
3rd pair : depth

If a multi-dimensioned array is used then several segments or views of the screen can be saved, thus allowing them to be PUT back to screen quickly in succession.

Associated: PUT

GOSUB...RETURN

Syntax: GOSUB <line no>/<label>

RETURN <line no>/<label>

Example:

```
GOSUB addvals
:
addvals:
 a=a+b
 b=b+c
 RETURN
```

Brief

Temporarily hands program flow to a subroutine, before returning.

Description

On encountering a GOSUB command AmigaBASIC notes the position of the command immediately following the GOSUB. It then locates the position of the subroutine given by <line no>/<label> and starts executing the commands located at this point. On encountering a RETURN, AmigaBASIC returns control to the point after calling the GOSUB as originally recorded.

An optional <line> may be specified after RETURN in which case control is returned to the first execute statement following this. This is essentially the same as finishing the subroutine with GOTO statement and so should be avoided in order to keep programs well structured and readable.

The use of subroutine allows a commonly used piece of code to be executed at various times within the program without the code having to be repeated. However subprograms provide a better method of achieving this.

GOTO

Syntax: GOTO <line no>/<label>

Example:

```
GOTO newline
```

Brief

Transfers program control to the line specified.

Description

On encountering the GOTO command, AmigaBASIC identifies the line given by <line no>/label and transfers program control to this point. The program continues from this point and any intervening statements are totally ignored. Use GOTO with care and sparingly since they make codes less readable. GOTOs should not normally be used to jump into and out of any form of loop structure.

HEX$

Syntax: HEX$ <decimal argument>

Example:

```
H=255
PRINT HEX$(H)
```

Brief

Returns a string containing the hexadecimal representation of the decimal value passed.

Description

The decimal value passed to the function is converted into hexadecimal format and then converted to a string. In the above example "FF" will be returned.

IF...GOTO

Syntax: IF <expression> GOTO <line>/<label> [ELSE <else clause>]

Example:

```
IF A=1 GOTO single ELSE GOTO notsingle
```

Brief

Branches to another part of a program if a condition is met.

Description

IF...GOTO can be thought of as a conditional GOTO command. The <expression> after the IF command is evaluated and if met the GOTO is executed and program control is recommenced at <line>/<label>.

An optional ELSE may be included. In such cases if the expression is false the statements after the ELSE are executed.

Use GOTO with care and sparingly. GOTOs should not normally be used to jump into and out of any form of loop structure.

IF...THEN....ELSE

Syntax: IF <expression> THEN <then clause> [ELSE <else clause>]

Example:

```
INPUT A$
IF A$=+ THEN X=X+Y ELSE X=X-Y
```

Brief

Executes a statement depending on whether a certain condition is met.

Description

The <expression> after the IF command is evaluated and if true the commands following THEN are executed. An optional ELSE may be included. In such cases if the expression is false the statements after the ELSE are executed. The statements attached to the <then clause> and <else clause> may not extend beyond a single line (see IF..THEN..ELSE Block for multi-line structure details).

IF...THEN...ELSE Block

Syntax: IF <expression> THEN
<statement block>
ELSEIF <expression> THEN
<statement block>
ELSE
<statement block>
END IF

Example:

```
INPUT A
 IF A=1 THEN
 PRINT "You Pressed 1"
ELSEIF A=2 THEN
 PRINT "You Pressed 2"
ELSE
 PRINT "You didn't press 1 or 2"
END IF
```

Brief

Executes a block of statements depending on whether certain conditions are met.

Description

The <expression> after the IF command is evaluated and, if true, the commands following THEN are executed. Once these have been executed, then the program execution resumes at the first statement after the END IF. If the <expression> is not true then the first ELSEIF <expression> is evaluated and, if this is true, then the statements following this are executed. Once these have been executed, then the program execution resumes at the first statement after the END IF. This is repeated for all ELSEIFs which are specified. If none of the expressions are true, then the statements following ELSE are executed if the ELSE part is present; otherwise nothing is executed. The block structure is terminated by an END IF statement.

The <statement block> may itself contain nested IF...THEN...ELSE blocks. The appropriate ELSEIF, ELSE and ENDIF statements act as terminating markers for the <statement block> preceding THEN.

The ELSEIF and ELSE blocks, as always, are optional and either or both may be omitted as required.

(If anything other than a REM follows THEN, AmigaBASIC will treat the structure as a single line IF...THEN...ELSE structure.)

INKEY$

Syntax: INKEY$

Example:

```
A=0
WHILE A=0
  k$=INKEY$
  IF k$="Y" THEN A=1
  IF k$="N" THEN A=2
WEND
PRINT A
```

Brief

Returns a one-character string from a keyboard keypress.

Description

INKEY$ does not give a prompt or wait for something to be input. It checks to see if any characters are waiting in the keyboard buffer (ie if any keys have been pressed) and returns the first of these as a string; otherwise it returns the nullstring.

Associated: INPUT, INPUT$

INPUT

Syntax: INPUT [;][<prompt>;/,]<variable list>

Example:

```
INPUT "What is your name  ", name$
INPUT "What is your age ", age
```

Brief

Reads the keyboard for input during a program.

Description

INPUT allows string or numeric data to be read in from the keyboard during the operation of a program. INPUT may be followed by a <prompt>, this is a string enclosed in quotes which is printed on-screen. The command then waits for information to be entered and the end of this is signified by pressing the RETURN key. The information is then assigned to the appropriate variables.

In the above example a comma has been used to delimit the <prompt> from the first variable. This may be replaced with a semi-colon, ';' in which case a question mark will be printed by AmigaBASIC after the <prompt> has been displayed. Thus:

```
INPUT "What is your name ", name1$
INPUT "What is your name "; name1$
```

will be displayed as follows:

```
What is your name FRED
What is your name ? JIM
```

If a semi-colon is given after the INPUT keyword, the RETURN given to end the input is not reflected to the screen and so the next PRINT statement will output text on the same line. If the semi-colon is omitted, a linefeed is issued.

Note that the number of items of data input must be the same as the number of variables in the list. These must all be input in one go,

separated by commas. In addition, the type of data input must match the type of variable it is being assigned to. If either of these conditions is not met, the user will be prompted to try again.

Associated: INKEY$, INPUT%

INPUT$

Syntax: INPUT$ (X[,[#]<file No.>])

Example:

```
PRINT "2 characters please"
a$=INPUT$ (2)
PRINT a$
```

Brief

Returns <X> characters from a file or the keyboard.

Description

If <file no.> is specified then a string of <X> characters is read from this file and returned. Otherwise, the function waits for the characters to be entered at the keyboard. Note that no RETURN is required to end the input and that the characters are not reflected on the screen as they are typed.

Associated: INKEY$, INPUT

INPUT#

Syntax: INPUT# <file no.>,<Variable list>

Example:

```
INPUT #1, A$,A
```

Brief

Reads data from a sequential file and assigns it to named variables.

Description

INPUT# is similar to INPUT$. However, here the information is read from a disc file rather than the keyboard. <file no.> identifies the file (already OPENed) which is the source of the information to be read. <Variable List> identifies the variables into which the information is to be placed and the variable types must match the incoming information.

AmigaBASIC ignores leading spaces, linefeeds and returns. For numbers, the end digit is identified when BASIC sees either a comma, space, return or linefeed.

String data can be identified by a starting quote,", and terminated by a second quote, in which cases spaces, commas etc within the string are preserved. Otherwise, the end of a string is identified by a comma, return, linefeed or the 255th character having been reached.

INSTR

Syntax: INSTR([<start>,]first$,second$)

Example:

```
A$="Epypidimovasostomy": B$="as"
foundone=INSTR(A$,B$)
foundtwo=INSTR(14,A$,B$)
```

Brief

Locates the position of a substring within a string.

Description

The string <first$> is searched for the first occurrence of the string <second$> and its start position in the string is returned. If <start> is specified then the search for <second$> starts from <start> characters into the string, ie <start> acts as an offset from the beginning of the string from which the search begins. If <second$> is not found within <first$> then 0 is returned

INT

Syntax: INT (<numeric expression>)

Example:

```
nuval=INT(X)
```

Brief

Returns the largest integer that is less than or equal to <numeric expression>.

Description

INT evaluates <numeric expression> and rounds it down to the nearest whole number less than or equal to it . In the above example the following values would be assigned to nuval:

value of X	nuval
9.23	9
3.01	3
4.00	4
88.99	8
-45.4	-46

Associated: CINT, FIX

KILL

Syntax: KILL "<filename>"

Example:

```
KILL "mugger"
```

Brief

Deletes named file from disc.

Description

KILL is used to delete a file from disc. <filename> may be any legal Amiga filename. If the named file is open, then a message to that effect will be issued and the file will not be erased.

LBOUND

Syntax: LBOUND (<array name> [, Dim])

Example:

```
 DIM array (1,2,3,4)
FOR loop=LBOUND (array,4) TO UNBOUND (array,4)
 PRINT array(0,0,0,loop)
 NEXT loop
```

Brief

Returns the lower bound of an array dimension.

Description

This function returns the Lower BOUNDary of the named array, <array name>. Used in conjunction with UBOUND it allows the values used within a DIM and OPTION BASE statement to be found.

LBOUND returns the value allocated by OPTION BASE and will therefore return either 0 or 1.

The use of <DIM> is optional and is used to indicate which dimension in a multidimensional array is to be used. The default value is 1, meaning the first dimension. Note that the result for all dimensions in a particular array will yield the same result.

Associated: UBOUND

LEFT$

Syntax: LEFT$(<string>,<N>)

Example:

```
B$="AmigaBASIC"
A$=LEFT$(B$,5)
PRINT A$
```

Brief

Returns <N> characters from the left-hand side of <string>.

Description

LEFT$ returns<N> characters from the left-hand side of <string>. If the number of characters asked for is greater than the number of characters in the string, then the entire string is returned. If the number is zero, then the null string is returned. In the example, A$ is assigned the string "Amiga".

Associated: MID$, RIGHT$

LEN

Syntax: LEN<string$>

Example:

```
A$="AmigaBASIC"
B=LEN(A$)
PRINT B
```

Brief

Returns the length of the named string.

Description

LEN counts the number of characters in the named string. The count includes non-printable characters and spaces if they form part of the string.

LET

Syntax: [LET]<variable>=<expression>

Example:

```
LET value=100
nextvalue=3200
```

Brief

Assigns a value to a variable.

Description

This keyword is entirely optional and may be used to make a program more 'readable'. The named variable has the expression to the right of the equals sign assigned to it. LET may be used to assign a value to any type of variable.

LIBRARY

Syntax: LIBRARY "<filename>"/CLOSE

Example:

```
LIBRARY "screen.library:"
LIBRARY CLOSE
```

Brief

Opens or closes a library of machine code subroutines and functions.

Description

AmigaBASIC allows up to five library files to be "attached" to a BASIC program at any one time. <filename> specifies where AmigaBASIC can locate the file (any legal Amiga file specification). These libraries remain active until a NEW, RUN or LIBRARY CLOSE statement is issued.

Note that before LIBRARY can be used, a library of routines must have a '.BMAP' file created for it. This lists their routines, their positions within the library and any parameters they have. The program 'Convert Fd.bas' supplied on the Extras disk may be used to produce a file of this type.

LIBRARY CLOSE closes all libraries that have been opened.

Associated: CALL, DECLARE FUNCTION

LINE

Syntax: LINE [[STEP](x1,y1)]-[STEP] (x2,y2) [,(<color>)[,B/BF]]

Example:

```
LINE (10,10)-(100,100),,BF
```

Brief

Draws a line or box in current Output Window.

Description

This command will draw a line, a rectangle or a filled-in box. A straight line is drawn by giving a start and end point using the standard co-ordinate syntax. The following command will draw a line across the screen:

```
LINE (10,10)-(100,100)
```

If the first set of co-ordinates are omitted, then the co-ordinates supplied are treated as the destination set and a line is drawn there from the current graphics cursor position, ie:

```
LINE (10,10)-(100,100)
LINE -(150,175)
```

If STEP is introduced into the command, then the co-ordinates supplied are treated as relative to the current graphics cursor position.

If a B is postfixed to the command line, then a box is drawn using the co-ordinates at the opposing corners, ie:

```
LINE (10,10)-(100,100),B
```

If BF is appended, then the box is filled in when drawn.

The colour of the line may be determined by specifying the colour id in <color>. If no colour id is supplied, then the current foreground colour is used.

Associated: PALETTE, COLOR, CIRCLE, AREA

LINE INPUT

Syntax: LINE INPUT [;][<prompt string>;]<string variable>

Example:

```
LINE INPUT "Your details :";det$
```

Brief

Reads an full line from the keyboard and assigns it to the <string variable> specified.

Description

LINE INPUT is similar in function to INPUT. <prompt> is an optional string that may be included and may contain any text. If included this is displayed on the screen at which point AmigaBASIC waits for the user to respond. The keyboard is read until RETURN is pressed, at which point the text entered is assigned to the <string variable>.

A semi-colon immediately following LINE INPUT is optional. If included, the RETURN typed by the user to terminate the string is not echoed to the screen. In other words, the next output to the screen will appear immediately to the right after the user typed input.

Note that, unlike INPUT, LINE INPUT will not issue a '?' prompt after the <prompt string>.

Associated: INPUT

LINE INPUT#

Syntax: LINE INPUT# <file no.>;<string variable>

Example:

```
LINE INPUT# 1; details
OPEN "exdat" FOR OUTPUT AS !
LINE INPUT " Your :";det$
PRINT#1,det$
CLOSE#1
OPEN "exdat" FOR INPUT AS 1
LINE INPUT#1, det$
PRINT det$
CLOSE
```

Brief

Reads an entire line from a sequential file.

Description

This command is identical in fashion to LINE INPUT. It differs in that the incoming data comes from disc and not the keyboard. The file from which the data is to be read is specified by <file no.>. The RETURN character identifies the end of the incoming data.

Associated: INPUT#

LIST

Syntax: LIST [<line>]

LIST [<start line>][-[<end line>]], <filename>

Examples:

```
LIST
LIST graphics
LIST startgraf-endgraf, "DF0:File1"
```

Brief

Lists the current program.

Description

LIST displays the program currently held in memory. LIST on its own lists the program in its entirety whereas LIST graphics lists all lines from the graphics label.

A listing may be sent to a file for later inspection by specifying the full file path in <filename>. For example:

```
LIST graphics, "DF0:Start"
```

will send a listing of the lines, from the beginning of the program up to the line labelled 'graphics', to a file called 'Start' in drive DF0:.

Using this form of the syntax, other options are available:

```
LIST - midway, "DF0::start"
```

will list lines, from the beginning of the program up to the line labelled "midway:", to the file;

```
LIST startgraf - endgraf, endgraf, "DF0:start"
```

will list lines, between and including the "startgraf:" and "endgraf:" labels, to the file.

A listing can be directed to an attached printer by using "PRT:" as the <filename>. The printer must be connected and ready to receive the listing.

Associated: LLIST

LLIST

Syntax: LLIST [<line>][-<line>]]

Example:

```
LLIST start
LLIST dothis-dothat
```

Brief

Sends a program listing, or part thereof, to the printer.

Description

LLIST sends a listing of the program currently held in memory to an attached printer. LLIST on its own lists the program in its entirety. There are a number of options:

LLIST graphics	List all lines from the graphics label.
LLIST -midway	List all lines from beginning of the program up to the line label 'midway:'
LLIST startgraf-endgraf	List all lines between, and including, the 'startgraf:' and 'endgraf:' labels.

LLIST is identical in operation to LIST when directing the listing to the device PRT:.

Associated: LIST

LOAD

Syntax: LOAD [<filename>[,R]]

Example:

```
LOAD "DF1:money.maker",R
```

Brief

Loads named file into memory.

Description

The file named in <filename>, which may be any legal Amiga filename specification, is loaded into memory. If the optional R postfix is used, the program will be loaded and automatically run. If <filename> is omitted, a requester box will appear and ask for the filename.

Note:

- LOAD closes all OPEN files, unless the R option is used.
- LOAD deletes all variables.
- The incoming program will erase any other program in memory.

Associated: CHAIN, MERGE, SAVE

LOC

Syntax: LOC (<file no.>)

Example:

```
file=LOC(1)
```

Brief

Returns the last record read/written (random access files) or the increment (for sequential disc files).

Description

When dealing with random access files, LOC returns the number of the last record that was written to or read from disc (floppy, hard or RAM). The file is specified by <file no.>.

When dealing with sequential disc files LOC returns what is called the 'increment'. This is the number of bytes that have been written to or read from the sequential file, divided by the default record size for the file (128 bytes by default, though this may have been changed when the file was OPENed). The file is specified by <file no.>.

LOCATE

Syntax: LOCATE [<line no>] [,<column no>]

Example:

```
LOCATE 10,2
PRINT "a";
```

Brief

Moves the text cursor to the position specified.

Description

The pen is relocated to the line and column specified in the current Output Window. If either <line no> or <column no> is omitted then the current setting is used. Thus:

```
LOCATE 2,10
LOCATE 3
```

Here the cursor will be located at 2,10 and in the second instance at 3,10.

The upper left hand corner of the screen is 1,1.

The maximum figures for the default screen mode, and therefore the figures which cause the cursor to move to the bottom right hand corner, are line 21, column 62 (LOCATE 21,62)

LOF

Syntax: LOF (<fileno>)

Example:

```
OPEN "ex" AS 1
length=LOF(1)
```

Brief

Returns the length of a file.

Description

The file specified by <fileno> is examined and its length is returned in bytes. Files that are opened to SCRN:, KYBD: or LPT1: return 0.

LOG

Syntax: LOG(<value>)

Example:

```
num=1
lrthm=LOG(num)
PRINT LOG(2), lrthm
```

Brief

Returns the natural logarithm of the the number supplied.

Description

This function calculates the natural logarithm (base e) of the value supplied, where the <value> must be greater than 0. The calculation is carried out in either single or double precision as defined by the precision of the value supplied.

LPOS

Syntax: LPOS (X)

Example

```
IF LPOS (0) >30 AND LPOS(0)<40 PRINT "."
.....LPOS (0) >30.....LPOS(0),40.....
```

Brief

Returns current position of the printer's print head.

Description

LPOS is similar to POS in action, but it gives the theoretical position of the printer head. However this is not necessarily the actual physical position of the printer head, as the value is calculated by counting the number of characters in the current line that have been output to the printer.

The value of "X" is consequential and is, in effect, a dummy argument.

LPRINT

Syntax: LPRINT [<expression list>]

Example:

```
LPRINT "Hello printer"
```

Brief

Sends items to printer.

Description

LPRINT works identically to PRINT except that the output is sent to the printer. The information is placed in the print buffer and transferred to the printer when a return code is encountered.

See PRINT for more information.

Associated: LPRINT USING, PRINT

LPRINT USING

Syntax: LPRINT USING <format string>;<expression string>

Example:

```
LPRINT USING "##.###";10.5, 10.6123
```

Brief

Sends items to printer in specified format.

Description

LPRINT USING works identically to PRINT USING except that the output is sent to the printer. The information is placed in the printer buffer and transferred to the printer for printing when a return code is encountered.

Associated: LPRINT, PRINT USING

LSET

Syntax: LSET <string$>=<string expression>

Example:

```
OPEN "example" AS 1 LEN=20
FIELD #1, QO AS sbuf$, 10 AS sbuf 2$
RSET sbuf1$="FRED"
LSET sbuf2$="JIM"
PUT#1
GET#1
PRINT sbuf1$
PRINT sbuf2$
CLOSE
```

Brief

Transfer data from memory to random access buffer in readiness for PUT.

Description

LSET is used in combination with PUT for writing data to random access files. The purpose of LSET is to move the data to the random file buffer. <string$> (which has been defined as the transfer string in FIELD) is assigned <string expression> and transferred to the file buffer.

LSET will left justify the string if it is shorter than the length assigned by FIELD. Any spaces at the start of the record are padded with spaces. This is particularly useful for lining up numeric values.

Associated: RSET, PUT, FIELD

MENU

Syntax: MENU <menu id>,<item id,<state>[,<title$>]

Example:

```
MENU 5,0,1, "Neptune"
MENU 5,1,0, "Triton"
MENU 5,2,1, "Nereid"
WHILE 1=1
WEND
```

Brief

Allows menus to be created.

Description

MENU can be used to construct menus in AmigaBASIC.

Four items of information can be passed with the command as follows:

<menu id> A number in the range 1 to 10 is the number assigned to the menu bar selection.

<item id> A number in the range 1 to 19 which identifies the menu option, ie its position in the list of options. The whole menu is identified by setting <item id> to 0.

<state> This defines the current state of the menu item (or all items of <item id>=0) and there are three possible settings:

- 0 Disable menu item(s)
- 1 Activate menu item(s)
- 2 Activate menu item(s) and place a check mark next to it. Note that you should always include two spaces ahead of your menu item name to allow for space for check marks.

<title$> If this is included, then the string is written at the appropriate point on the menu (either as the menu item name or the menu name)

Associated:

MENU (0/1), MENU RESET, ON MENU, MENU ON/OFF/STOP

MENU (0/1)

Syntax: MENU (<0/1>)

Example:

```
number=MENU (0)
item=MENU (1)
```

Brief

Provides information about last menu option chosen.

Description

The MENU function can provide two items of information as follows:

MENU (0)	Returns the number (from 1 to 10) of the menu from which the last option was selected.
MENU (1)	Returns the number (1 to 19) of the item selected in the menu.

MENU RESET

Syntax: MENU RESET

Brief

Restores AmigaBASIC's default Menu Bar.

Description

Any existing user defined menus are removed and AmigaBASIC's menu bar is restored.

Associated: MENU ON/OFF/STOP,ON MENU,SLEEP

MENU ON/OFF/STOP

Syntax: MENU <ON/OFF/STOP>

Examples:

```
:
ON MENU GOSUB menuhard
MENU ON
:
```

Brief

Enables, disables or suspends menu event trapping.

Description

A menu event occurs whenever a menu item, defined by the MENU statement, is selected from a menu. MENU ON enables this event trapping while MENU OFF disables event trapping.

MENU STOP disables the ON MENU...GOSUB command from taking effect even though menu event trapping is allowed to continue. Any events noted are only acted upon when MENU ON is issued.

Associated: MENU, ON MENU MENU(0/1)

MERGE

Syntax: MERGE <file name>

Example:

```
MERGE "extras"
```

Brief

Appends the named file to the program currently resident in memory.

Description

AmigaBASIC looks for <file name> and, once found, appends the file to the program that is already in memory. <file name> may be any legal Amiga filename (including device) and the file to be appended must be in the form of an ASCII file, ie originally saved using the A option (see SAVE).

Associated: SAVE, CHAIN

MID$

Syntax: MID$(<string$>,<position>[,<length>])=<newstring>

<section$>=MID$(<string$>,<position>[,<length>])

Examples:

```
A$="AmigaBASIC: A Goodish Guide"
MID$(A$,15)="Dabhand"
A$="AmigaBASIC"
B$=MID$(A$,6,5)
```

Brief

Either writes a substring over part of another string or reads a portion of a string from another string.

Description

MID$ can operate in one of two ways:

As a statement it can be used to write one string over part of another. In the first example given above, 'Goodish' is replaced by 'Dabhand'. <string$> defines the main string, <position> the starting point into the string, and <new string> the string which is to replace the characters of <string$> at this point. If <length> is stated, then only the number of characters specified by <length> will be overwritten, otherwise the whole of <new string> will be used. If <position> is larger than the number of characters in <string$> then nothing happens.

As a function MID$ can be used to read a substring from within a string. In the second example above, B$ is assigned the string "BASIC". <string$> defines the main string and <position> the point within the string at which characters are to be taken. <length> determines the number of characters to be extracted. If this is not specified, then all characters from the point determined by <position> to the end of the string are returned. If <position> is larger than the number of characters in <string$> then a null string is returned.

Note that <position> and <length> must be in the range 1-32767.

Associated: LEFT$,RIGHT$

MKD$

Syntax: MKD$(<double precision expression>)

Example:

```
number#=123456.789
value$=MKD$(number#)
```

Brief

Converts a double precision value into an eight-byte string ready for insertion into a random access file buffer.

Description

Before numeric variables can be put into a random access file, they must be converted into string variables. This may also be done for sequential files to save time and storage space. MKD$ takes any double precision expression and returns it in string format.

Associated: CVD,LSET,RSET,FIELD,PUT#

MKI$

Syntax: MKI$(<short integer expression>)

Example:

```
number%=1234
value$=MKI$(number%
OPEN "ex1" AS 1 LEN=2
FIELD #1,2 AS sbuf$
LSET sbuf$=MKI$(4
PUT#1,1
```

Brief

Converts a short integer into a two-byte string ready for insertion into a random access file buffer.

Description

Before numeric variables can be put into a random access file they must be converted into string variables. This may also be done for sequential files to save time and storage space. MKI$ takes any short integer (16-bit) expression and returns it in string format.

Associated: CVI,LSET,RSET,FIELD,PUT#

MKL$

Syntax: MKL$(<long integer expression>)

Example:

```
number&=123456
value$=MKI$(number)
```

Brief

Converts a long integer into a four-byte string ready for insertion into a random access file buffer.

Description

Before numeric variables can be put into a random access file, they must be converted into string variables. This may also be done for sequential files to save time and storage space. MKL$ takes any long integer (32-bit) expression and returns it in string format.

Associated: CVL,LSET,RSET,FIELD,PUT#

MKS$

Syntax: MKS$(<single precision expression>)

Example:

```
number!=1234.56
value$=MKD$(number!)
```

Brief

Converts a single precision value into a four-byte string ready for insertion into a random access file buffer.

Description

Before numeric variables can be put into a random access file, they must be converted into string variables. This may also be done for sequential files to save time and storage space. MKS$ takes any single precision expression and returns it in string format.

Associated: CVS,LSET,RSET,FIELD,PUT#

MOUSE

Syntax: MOUSE(<num>)

Example:

```
ON MOUSE GOSUB mousehand
MOUSE ON
WHILE 1=1
WEND
mousehand:
WHILE MOUSE(0)=0
WEND
LINE(MOUSE(3),MOUSE(4))-(MOUSE(5),MOUSE(6))
RETURN
```

Brief

Monitors the left mouse button and the mouse cursor location.

Description

Mouse performs a variety of functions, dependent on the value assigned to <num>. These are summarised in the table below and detailed afterwards.

<num>	Function
0	Returns left button status
1	Returns 'current' X co-ordinate of mouse cursor
2	Returns 'current' Y co-ordinate of mouse cursor
3	Returns 'starting' X co-ordinate of mouse cursor
4	Returns 'starting' Y co-ordinate of mouse cursor
5	Returns 'ending' X co-ordinate of mouse cursor
6	Returns 'ending' Y co-ordinate of mouse cursor

MOUSE(0)

MOUSE(0) returns the status of the left mouse button and also makes AmigaBASIC remember the start and end positions of the mouse cursor until the command is issued again. These are used by MOUSE(3), MOUSE(4), MOUSE(5) and MOUSE(6).

MOUSE(0) will return values as follows:

0 The left mouse button has not been pressed since the last MOUSE(0) command.

1 The left mouse button is not being pressed at the moment but has been pressed since the last MOUSE(0) command.

2 The left mouse button is not being pressed at the moment but has been pressed twice since the last MOUSE(0) command. (Note: a value of three indicates that the left mouse button has been pressed three times since the command was last issued, etc.)

-1 The user is pressing and holding down the left mouse button after clicking it once.

-2 The user is pressing and holding down the left mouse button after clicking it twice since the last MOUSE(0) command. (Note: a value of -3 indicates that the left mouse button has been clicked three times since the command was last issued, etc.)

MOUSE(1)
This returns the X co-ordinate of the pointer the last time the MOUSE(0) command was used.

MOUSE(2)
This returns the Y co-ordinate of the pointer the last time the MOUSE(0) command was used.

MOUSE(3)
This returns the X co-ordinate of the pointer the last time the left button was pressed before the last MOUSE(0) command was used.

MOUSE(4)
This returns the Y co-ordinate of the pointer the last time the left button was pressed before the last MOUSE(0) command was used.

MOUSE(5)
If the left button was depressed on the last MOUSE(0) call, then this returns the X co-ordinate of the pointer when MOUSE(0) was used. If the left button was not pressed, then it returns the X co-ordinate of the pointer when the left button was released.

MOUSE(6)
If the left button was depressed on the last MOUSE(0) call, then this returns the Y co-ordinate of the pointer when MOUSE(0) was used. If the left button was not pressed, then it returns the Y co-ordinate of the pointer when the left button was released.

Associated: MOUSE ON/OFF/STOP, ON MOUSE

MOUSE ON/OFF/STOP

Syntax: MOUSE <ON/OFF/STOP>

Examples:

```
ON MOUSE GOSUB Mousehand
```

MOUSE ON

Brief

Enables, disables or suspends event trapping based on pressing of the mouse button.

Description

A mouse click can be used to cause an event and, if mouse event trapping is enabled with MOUSE ON, then any ON MOUSE...GOSUB will be executed. If mouse event trapping is disabled with MOUSE OFF, any ON MOUSE...GOSUBs will be ignored. MOUSE STOP does not disable mouse event trapping but inhibits the execution of ON MOUSE...GOSUB commands until a MOUSE ON command is executed.

Associated: MOUSE, ON MOUSE

NAME

Syntax: NAME "<old name>" AS "<new name>"

Example:

```
NAME "Credits" AS "Overdraft"
```

Brief

Renames the file called <old name> to <new name>.

Description

AmigaBASIC locates the file called <old name> (on floppy, hard or RAM disc) and renames it to the name given in <new name>. Note that <new name> must not already exist.

NEW

Syntax: NEW

Brief

Deletes any program in memory and clears the list window and all variables.

Description

NEW is used to perform a 'soft' BASIC reset. The command is entered in immediate mode and clears memory of any programs and variables that may be present. The list window is also erased and closes all files and turns trace mode off.

NEW may be used from within programs, in which case it stops current program execution and returns AmigaBASIC to edit mode.

NEXT

Syntax: NEXT[<variable>[,<variable>[,...]]]

Example:

```
FOR a=1 TO 10
FOR b=1 TO 10
NEXT b,a
```

Brief

Loop end marker.

Description

Used in conjunction with FOR. See FOR for more details.

OBJECT.AX

Syntax: OBJECT.AX <id>,<rate>

Example:

```
OBJECT.AX 2,5
```

Brief

Sets acceleration for object in horizontal direction.

Description

This command sets the rate of acceleration for the object number given by <id> in a horizontal (X) direction. The <rate> is given in pixels per second per second. A positive number accelerates the object towards the right-hand side of the screen left to right. A negative number accelerates it towards the left.

Associated: OBJECT.AT,OBJECT.START,OBJECT.VS

OBJECT.AY

Syntax: OBJECT.AY <id>,<rate>

Example:

```
OBJECT.AY 2,5
```

Brief

Sets acceleration for object in vertical direction.

Description

This command sets the rate of acceleration for the object number given by <id> in a vertical (Y) direction. The <rate> is given in pixels per second per second. A positive number accelerates the object downwards. A negative number accelerates it upwards.

Associated: OBJECT.AX,OBJECT.START,OBJECT.VY

OBJECT.CLIP

Syntax: OBJECT.CLIP (x1,y1)-(x2,y2)

Example:

```
OBJECT.CLIP (10,10)-(200,200)
```

Brief

Inhibits drawing of objects outside of rectangular area.

Description

This command defines a box using the two sets of coordinates. This marks the limits for normal object movement. A collision will normally be registered when an object reaches any edge of the box.

The default value of the clip box is the border of the current Output Window. However, if you alter the size of the window with the sizing gadget, the boundaries of the box are not automatically updated; you need to use OBJECT.CLIP to align the clipping window to the new actual window extent.

Associated: OBJECT.HIT

OBJECT.CLOSE

Syntax: OBJECT.CLOSE [<id>[,...]]]

Example:

```
OBJECT.CLOSE 2,3
```

Brief

Releases all memory held by specified object(s).

Description

Each object requires an amount of memory to hold its description. When an object is no longer required, this memory may be freed with this command. The object's id should be given. This allows the id to be reused. If no ids are given then all the objects are closed. The command works on the current Output Window.

OBJECT.HIT

Syntax: OBJECT.HIT<id>[,<Object Mask][,<Hit Mask>]

Example:

```
OBJECT.HIT 2,,0
```

Brief

Defines which objects may collide with other objects.

Description

By default all objects collide with each other and the borders. Using this command you may determine which objects may be made invisible to one another (ie they will not collide) and therefore which objects will be allowed to collide. Collision is tested for by using collision event trapping (see ON COLLISION GOSUB).

<id> is the object's id as defined by the OBJECT.SHAPE statement. <Object Mask> is a short integer which defines a mask for the object itself and <Hit Mask> is a short integer which describes the objects that the object <id> is to collide with.

If and when two objects 'hit', the first mask value of one and the second mask value of the other are logically ANDed together. If the result is zero then the objects do not collide. If the result is non-zero then the objects do collide.

Note, bit 0 of <Hit Mask> refers to the borders. Therefore, if this is set to 0 (ie the value is even), object <id> will not collide with the borders.

Associated: OBJECT,CLIP,ON COLLISION

OBJECT.OFF

Syntax: OBJECT.OFF[<id>[,<id]...]

Example:

```
OBJECT.OFF
```

Brief

Makes specified objects invisible.

Description

The objects defined by the <id> list are made 'invisible'. They disappear and they cannot collide. The command also stops the objects from moving. If no <id> is specified then all the objects in the current output window are made invisible. The <id> is the <id> defined by an OBJECT.SHAPE statement.

Associated: OBJECT.ON

OBJECT.ON

Syntax: OBJECT.ON[<id>[,<id]...]

Example:

```
OBJECT.ON
```

Brief

Makes specified objects visible.

Description

The objects defined by <id> are made 'visible' so that they appear on the screen and may collide. If no <id> is specified then all the objects in the current output window are made visible. The <id> is the <id> defined by an OBJECT.SHAPE statement.

Associated: OBJECT.OFF, OBJECT.SHAPE, OBJECT.START

OBJECT.PLANES

Syntax: OBJECT.PLANE <id> [,<8bit num1>[,<8bit num2>]

Example:

```
OBJECT.PLANE 2,4
```

Brief

Defines in which bit planes an object will appear.

Description

<8bit num1> is an eight bit number (0-255) which determines which bitplanes the object, given by <id>, will appear in. <8bit num2> (et al) determines what happens in other bitplanes.

OBJECT.PRIORITY

Syntax: OBJECT.PRIORITY <id>,<value>

Example:

```
OBJECT.PRIORITY 3,27
```

Brief

Defines an object's priority.

Description

This command allows you to define a priority number for an object. If two objects are occupying the same space on the screen then the object with the higher priority will be displayed in front of the one with the lower priority. Two objects with the same priority are drawn in a random order.

<value> may be in the range -32768 to 32767.

Note that this only applies to bobs.

OBJECT.SHAPE

Syntax: OBJECT.SHAPE <id>,<definition$>

OBJECT.SHAPE <id1>,<id2>

Examples:

```
OPEN "<file name>" FOR INPUT AS 1
 OBJECT.SHAPE, INPUT$(LOF(1),1)
 CLOSE #1
#
??????
OBJECT.SHAPE 3,2
```

Brief

Defines the shape and colour of an object.

Description

This command has two syntaxes, the first of which allows an object shape to be defined and the second of which allows an object shape's definition to be copied to a second object.

Syntax 1

This first syntax allows an object's shape and other attributes to be defined. The information must be supplied in the form of a definition string, <definition$>, and is assigned to the object defined by <id>, this being its id number. The definition string is best read from a sequential file, created using the Object Editor program which can be found in the BASICDemos drawer of the Extras Disk.

Syntax 2

This allows you to copy the definition assigned to <id2> to <id1>. As both objects will share a reasonable amount of memory, the memory overheads required by a series of objects are reduced using this method. However, you may specify differing attributes, such as direction of movement etc, to each using other OBJECT statements; it is only their appearance which they share.

OBJECT.START

Syntax: OBJECT.START[<id>[,<id>[,...]]]

Example:

```
OBJECT.START 1,2,3
```

Brief

Sets one or more objects into motion.

Description

The objects in the current output window whose numbers are specified in the <id> list are set into motion. If no <id> numbers are specified then AmigaBASIC starts all objects in the current Output Window.

When two objects collide, AmigaBASIC performs an OBJECT.STOP on the objects (or object if collision is with a border).

Associated:OBJECT.SHAPE, OBJECT.STOP, OBJECT.VX, OBJECT.VY

OBJECT.STOP

Syntax: OBJECT.STOP[<id>[,<id>[,...]]]

Example:

```
OBJECT.STOP 1,2,3,
```

Brief

Freezes the motion of one or more objects.

Description

The objects whose numbers are specified in the <id> list are frozen and not allowed to move. If no <id> numbers are specified then AmigaBASIC stops all objects.

When two objects collide, AmigaBASIC performs an OBJECT.STOP on the objects (or object if collision is with a border).

Associated: OBJECT.SHAPE, OBJECT.START

OBJECT.VX

Syntax: OBJECT.VX <id>,<Value>

<var>=OBJECT.VX(<id>)

Example:

```
OBJECT.VX 2,5
speedX=OBJECT.VX(2)
```

Brief

Defines or returns velocity of an object in a horizontal direction.

Description

The keyword may be used as a statement or as a function.

In statement form, the object specified by <id> is assigned a horizontal velocity in pixels per second, given by <Value>. If used as a function then the horizontal speed of the object given by <id> is returned in pixels per second. A positive value indicates movement to the right, a negative value to the left.

Associated: OBJECT.VY,OBJECT.START,OBJECT.AX

OBJECT.VY

Syntax: OBJECT.VY <id>,<Value>

<var>=OBJECT.VY(<id>)

Example:

```
OBJECT.VY 2,5
SpeedY=OBJECT.VY(2)
```

Brief

Defines or returns velocity of an object in a vertical direction.

Description

The keyword may be used as a statement or as a function.

In statement form, the object specified by <id> is assigned a vertical velocity in pixels per second, given by <Value>. If used as a function, then the vertical speed of the object given by <id> is returned in pixels per second. A positive value indicates movement downwards, a negative value is upwards.

OBJECT.X

Syntax: OBJECT.X<id>,<Value>

<Xpos>=OBJECT.X(<id>)

Examples:

```
OBJECT.X 5,100
Xpos=OBJECT.X(5)
```

Brief

Places an object at a coordinate on horizontal axis or returns its horizontal position.

Description

This keyword may be used as a statement or as a function. As a statement, the object given by <id> is positioned at a horizontal location in the current output window, as given by <value>. This command may be used in conjunction with OBJECT.Y which specifies the vertical co-ordinate. The co-ordinate may be in the range -32768 to 32767.

As a function, the horizontal co-ordinate of the object given by <id> is returned.

In both cases, the position is that of the left-hand side of the object rectangle.

Associated: OBJECT.Y

OBJECT.Y

Syntax: OBJECT.Y <id>,<Value>

<Ypos>=OBJECT.Y(<id>)

Examples:

```
OBJECT.Y 5,100
Ypos=OBJECT.Y(5)
```

Brief

Places an object at a co-ordinate on the vertical axis or returns its vertical position.

Description

This keyword may be used as a statement or as a function. As a statement, the object given by <id> is positioned at a vertical location in the current output window, as given by <value>. This command may be used in conjunction with OBJECT.X which specifies the horizontal co-ordinate. The co-ordinate may be in the range -32768 to 32767.

As a function, the vertical co-ordinate of the object given by <id> is returned.

In both cases, the position is that of the top of the object rectangle.

Associated: OBJECT.X

OCT$

Syntax: OCT$(<num/var>)

Example:

```
PRINT OCT$(number%)
```

Brief

Converts a decimal value into an octal one and returns it in string format.

Description

This statement converts the decimal number argument into an octal value (ie base 8). The number supplied is rounded to a whole integer first. The result is returned as a string.

Associated: HEX$

ON BREAK

Syntax: ON BREAK GOSUB <line/label>

ON BREAK GOSUB 0

Example:

```
ON BREAK GOSUB brkhandler
BREAK ON
```

Brief

Executes GOSUB routine on break event.

Description

Should a break event occur, then the last statement of this kind, ie the active one, is executed. Program control is passed to the subroutine at <line/label>. GOSUB 0 disables the break event.

The statement has no effect until a BREAK ON statement has been executed.

A break event occurs when:

- the user presses CTRL-C
- the user selects Stop from the Run Menu

Associated: BREAK ON/OFF/STOP

ON COLLISION

Syntax: ON COLLISION GOSUB <line/label>

ON COLLLISION GOSUB 0

Example:

```
ON COLLISION GOSUB collhandler
COLLISION ON
```

Brief

Executes GOSUB routine on collision event.

Description

Should a collision event occur then the last statement of this kind, ie the active one, is executed. Program control is passed to the subroutine at <line/label>. COLLISION 0 disables the collision event.

The statement has no effect until a COLLISION ON statement has been executed.

ON ERROR GOTO

Syntax: ON ERROR GOTO <line/label>

ON ERROR GOTO 0

Example:

```
ON ERROR GOTO doerror
```

Brief

Hands program control to specified line when an error occurs.

Description

If an error occurs, then AmigaBASIC will hand control to the most recently executed ON ERROR GOTO statement. Control is then passed to the line specified in <line/label>.

Executing ON ERROR GOSUB 0 will disable error handling.

Associated: RESUME

ON...GOTO

Syntax: ON <expression> GOTO <line/label(s)>

Example:

```
ON number% GOTO addone,addtwo,addthree
```

Brief

Passes control to line determined by <expression>.

Description

A list of line numbers/labels may be given and the value returned by <expression> will determine which line control is passed to. For example, if the value returned by <expression> was two, then the second line in the line list after the GOTO will be jumped to. If the result returned by <expression> has a decimal portion, then this is rounded. However, if the result is 0 or greater, then the number of entries in the line list then no GOTO is executed and control moves to the next statement.

Note that values outside of the range 0-255 cause an error.

Associated: ON GOSUB

ON...GOSUB

Syntax: ON <expression> GOSUB <line/label(s)>

Example:

```
FOR child%=1 TO 10
INPUT "Name:";name1$
INPUT "Grade:";grade%
ON grade% GOSUB 1,2,3
NEXT
END
1 : PRINT "Well done "name1$
RETURN
2 : PRINT "That's OK "name1$
RETURN
3 : PRINT "You must do better "name1$
RETURN
```

Brief

Passes control to subroutine determined by <expression>.

Description

A list of line numbers/labels may be given and the value returned by <expression> will determine which line control is passed to. For example, if the value returned by <expression> was two, then the subroutine starting at the second line in the line list after the GOSUB will be executed. If the result returned by <expression> has a decimal portion, then this is rounded. However, if the result is 0 or greater than the number of entries in the line list, then no subroutine will be called and control will move to the next statement.

Note that values outside of the range 0-255 cause an error.

Associated: ON GOTO

ON MENU

Syntax: ON MENU GOSUB <line/label>

ON MENU GOSUB 0

Example:

```
MENU 5,0,1,"Colour
MENU 5,1,1,"Red"
MENU 5,2,1,"Green"
MENU 5,3,1,"Blue"
ON MENU GOSUB menuhand
MENU ON
WHILE 1=1 : WEND
col%(1)=0:col%(2)=0:col%(3)=0
col%(MENU(1))=1
PALETTE 0,col%(1),col%(2),col%(3)
CLS
RETURN
```

Brief

Executes subroutine after a menu event.

Description

The most recently execute ON MENU command will be invoked and the appropriate subroutine executed whenever the MENU(0) function would return a non-zero result, ie when the user selects a menu item thereby generating a menu event.

Control is passed to the subroutine starting at the line specified in <line/label>. The ON MENU command will have no effect until a MENU ON statement has been executed.

ON MENU GOSUB 0 has the effect of disabling menu events.

Associated: MENU ON/OFF/STOP

ON MOUSE

Syntax: ON MOUSE GOSUB <line/label>

ON MOUSE GOSUB 0

Example:

```
ON MOUSE GOSUB domouse
```

Brief

Executes subroutine after a mouse event.

Description

The most recently executed ON MOUSE command will be invoked and the appropriate subroutine executed whenever a mouse event occurs.

Control is passed to the subroutine starting at the line specified in <line/label>. The ON MOUSE command will have no effect until a MOUSE ON statement has been executed.

ON MOUSE GOSUB 0 has the effect of disabling mouse events.

Associated: MOUSE, MOUSE ON/OFF/STOP

ON TIMER

Syntax: ON TIMER (<num>) GOSUB <line/label>

ON TIMER GOSUB 0

Example:

```
ON TIMER 2 GOSUB timeout:
```

Brief

Executes subroutine specified after given time interval has elapsed.

Description

After execution of this statement, AmigaBASIC will trap timer events every <num> seconds, at which point the named subprogram will be called.

<num> must be greater than 0 and less than 86400 (24 hours).

The events can be disabled by calling GOSUB 0.

ON TIMER does not come into effect until a TIMER ON statement has been executed to enable the timer events.

OPEN

Syntax: OPEN <mode>, [#]<file no.>, <file name>[,<size>]

OPEN <file name> [FOR <mode>] AS [#]<file No> [LEN=<size>]

Examples:

```
OPEN "hockey" FOR INPUT AS 1
OPEN File$ AS 2
```

Brief

Allows input and output to a disc.

Description

The OPEN command has two flavours both of which operate much the same. The end result being that a disc file is opened to allow data to be written to it or data to be read from it.

<file No.> is a number which is unique and assigned to the file <file name> for the period that the file is opened. The <file no.> is sometimes referred to as its file 'handle'.

<file name> is the name of the file and may be any legal Amiga file name and can include the path (ie directory tree) of the file. SCRN:, KYBD:, PRT: and SER: may be used to specify I/O to devices other than disc.

<size> is the size of the file buffer.

By default a <size> of 128 bytes is allocated. The maximum length is 32767 bytes and in the case of random access files the <size> should be the same as the record length.

As a rule the larger the file buffer, the faster it will run because the disc will be accessed less often. However, this amount of memory will be taken from AmigaBASIC's data area and this may affect the operation of the program.

In the first syntax <mode> is a character string whose first character is O (sequential output), I (sequential input), R (random I/O) or A (sequential append).

In the second syntax the mode is specified using one of three keywords: OUTPUT (sequential output), INPUT (sequential input) or APPEND (sequential output with file pointer set to end of file). The default, if none of these is given, being to use random access mode.

Associated: CLOSE, FIELD, GET, PUT, PRINT#, INPUT#...

OPTION BASE

Syntax: OPTION BASE<1/0>

Example:

```
OPTION BASE 1
```

Brief

Sets minimum value for array subscripts.

Description

Defines whether the first subscript of an array will be 0 or 1. Using any other number will cause an error. This command must be executed before arrays are defined or used.

Associated: DIM, LBOUND

PAINT

Syntax: PAINT [STEP] (<x,y>) [,<Paint id>[,<border id>]]

Example:

```
CIRCLE (300,100),80,1
PAINT (300,100),3,1
```

Brief

Paints area in selected colour.

Description

The x and y coordinates are used as the base point from which a region of screen is painted. If STEP is included then the coordinates are measured relative to the current position of the graphics cursor. The area is painted until a border is reached. The border may be the outline of a box or circle etc.

<paint id> defines the colour in which the area will be painted. If this is not specified, the current foreground colour is used.

<border id> defines the colour used to denote the boundary for painting. Any pixels met matching this colour prevent the painting from spreading any further in that direction. Pixels in any other colour will be overdrawn. If <border id> is not specified, then the <paint id> colour is used.

PALETTE

Syntax: PALETTE <color>,<red>,<green>,<blue>

Example:

```
COLOR 1,0
PALETTE 0,0,0,0
PALETTE 3,0.75,0.50,0.25
CLS
LINE (100,50)-(200,100),3,bf
```

Brief

Assigns an actual colour to a colour number.

Description

This command defines what colour will appear when graphics are drawn using a particular colour number. The command is followed by the colour number to be defined. This can be any value between 0 and the number of colours available in the current screen mode – 1. For example, using the default screen and output window, four colours are available so the colour number must be 0, 1, 2 or 3. This is followed by three further numbers each between 0.00 and 1.00 which determine the amounts of red, green and blue the colour is to contain. The higher the number, the higher the amount of that particular colour.

Using this command gives you access to the full range of colours which the Amiga supports. Although you may only have a limited number of colours available at once, these can be any shade you like. Note that if you change the colour associated with a particular colour number then any graphics previously drawn in this colour number will also change.

Associated: COLOR, CIRCLE, LINE, PAINT, PRESET, PSET

PATTERN

Syntax: PATTERN [<line-pattern>] [,<area-pattern>]

Example:

```
COLOR 1,3
DIM pat%(3)
pat%(0) = &H8888
pat%(1) = &H4444
pat%(2) = &H2222
pat%(3) = &H1111
PATTERN ,pat%
AREA (100, 80)
AREA (500, 80)
AREA (250,150)
AREAFILL
```

Brief

Defines pattern for drawing lines and/or filling polygons.

Description

The [line-pattern], if present, is a 16-bit integer whose bits define the pattern to be used for drawing lines. This pattern determines how one segment of the line will appear and the full line is drawn by repeating this segment several times until the length required is reached. The segment is 16 pixels long; each pixel can either be drawn or missing depending on whether the associated bit is 1 or 0.

The [area-pattern] is the name of an integer array containing the pattern to be used for filling polygons when using the AREAFILL command. The number of elements in the array must be a power of two, that is: 2, 4, 8, 16, 32, 64, etc. Each element defines a strip 16 pixels wide (one bit of the number defining whether a pixel in the pattern will be set or clear). The number of elements defines the height of the pattern block.

When the pattern is used, the way in which it affects the screen is determined by the argument given to AREAFILL. One option is for bits set to be drawn in the foreground colour and bits which are clear in the background colour. The other is for bits which are set to invert

what is already on the screen and bits which are clear to leave it unchanged.

Associated: AREA, AREAFILL, LINE

PEEK

Syntax: PEEK(<address>)

Example:

```
PRINT PEEK(0)
```

Brief

Returns the 8-bit (one byte) contents of a memory location.

Description

This function requires one argument which is the address of a particular memory location. The value returned by the function is the 8-bit number stored in that memory location. This function can be used to examine the contents of any byte in the Amiga's memory. The maximum address allowed is 16777215; however the maximum meaningful value depends on the amount of RAM you have fitted in your machine.

Associated: PEEKL, PEEKW, POKE

PEEKL

Syntax: PEEKL(<address>)

Example:

```
A& = 1
addr& = VARPTR(A&)
PRINT PEEKL(addr&)
```

Brief

Returns the 32-bit (one long word) contents of a memory location.

Description

This function requires one argument which is the address of a particular memory location. The value returned by the function is the 32-bit number stored in the four bytes starting at that memory location. Note that the address must be an even number.

A long word can be used to hold a long integer. Therefore, this function can be used to return the value of a long integer variable, given its address.

Associated: PEEK, PEEKW, POKEL

PEEKW

Syntax: PEEKW(<address>)

Example:

```
A% = 1
addr& = VARPTR(A%)
PRINT PEEKW(addr&)
```

Brief

Returns the 16-bit (1 word) contents of a memory location.

Description

This function requires one argument which is the address of a particular memory location. The value returned by the function is the 16-bit number stored in the two bytes starting at that memory location. Note that the address must be an even number.

A word can be used to hold a short integer. Therefore, this function can be used to return the value of a short integer variable, given its address.

Associated: PEEK, PEEKL, POKEW

POINT

Syntax: POINT (X,Y)

Example:

```
PRINT POINT(50,50)
```

Brief

Returns the colour number of a point in the current output window.

Description

This function requires the x and y-coordinates of a position relative to the current output window. It then returns the colour number of the point identified. If the coordinate lies outside of the window, the number returned is –1.

Associated: PRESET, PSET

POKE

Syntax: POKE <address>,<data>

Example:

```
OPTION BASE 1
DIM block%(100)
addr& = VARPTR(block%(1))
POKE addr&,1
```

Brief

Writes an 8-bit (one byte) value into a memory location.

Description

This command requires two arguments. The first is the address of a particular memory location. The second is the 8-bit value which is written to that location.

Note that altering the values stored in memory locations arbitrarily can be dangerous. If you accidentally overwrite a location which the Operating System or AmigaBASIC itself is using for something important, you may produce a fatal error and be forced to reboot the whole system.

Associated: PEEK, POKEL, POKEW

POKEL

Syntax: POKEL <address>,<data>

Example:

```
A& = 1
addr& = VARPTR(A&)
POKEL addr&,2
PRINT A&
PRINT PEEKL(addr&)
```

Brief

Writes a 32-bit (one long word) value into a memory location.

Description

This command requires two arguments. The first is the address of a particular memory location. The second is the 32-bit (long word) value which is written to the four bytes starting at that location.

A long word can be used to hold a long integer. Therefore, this function can be used to alter the value of a long integer variable, given its address.

Note that altering the values stored in memory location arbitrarily can be dangerous. If you accidentally overwrite a location which the operating system or AmigaBASIC itself is using for something important, you may produce a fatal error and be forced to reboot the whole system.

Associated: PEEKL, POKE, POKEW

POKEW

Syntax: POKEW <address>,<data>

Example:

```
A% = 1
addr& = VARPTR(A%)
POKEW addr&,2
PRINT A%
PRINT PEEKW(addr&)
```

Brief

Writes a 16-bit (one word) value into a memory location.

Description

This command requires two arguments. The first is the address of a particular memory location. The second is the 16-bit (word) value which is written to the two bytes starting at that location.

A word can be used to hold a short integer. Therefore, this function can be used to alter the value of a short integer variable, given its address.

Note that altering the values stored in memory locations arbitrarily can be dangerous. If you accidentally overwrite a location which the operating system or AmigaBASIC itself is using for something important, you may produce a fatal error and be forced to reboot the whole system.

Associated: PEEK, PEEKL, PEEKW, POKE, POKEL

POS

Syntax: POS(X)

Example:

```
PRINT "A";
curlin% = CSRLIN
curcol% = POS(0)
LOCATE 10,10
PRINT "1"
LOCATE curlin%,curcol%
PRINT "B"
```

Brief

Returns the current text column number.

Description

This function returns the approximate column number of the text cursor. The is the column in which the next character will appear if the cursor is not explicitly moved beforehand. The columns start at one for the left-hand column and increase by one for each character position to the right. The value is only approximate if a proportionally spaced font is used, in which case the number returned is based on the width of the character 'O' in this font.

The value of 'X' is inconsequential and is, in effect, a dummy argument.

Associated: CSRLIN, LOCATE

PRESET

Syntax: PRESET [STEP] (x,y) [,color]

Example:

```
FOR I% = 100 TO 500 STEP 10
 PRESET (I%,50),1
NEXT
```

Brief

Plots a point at a specified position.

Description

This command will draw a single point at the position given using the standard coordinate syntax. If the position defined lies outside of the current output window then the command is ignored.

If STEP is introduced into the command, then the coordinates supplied are treated as relative to the current pen position.

The colour of the point may be determined by specifying the colour number to use in <color>. The actual colour will then be as defined with the PALETTE statement. If no colour number is supplied then the specified point is set to the background colour. Therefore, in the example, if the ',1' was deleted, nothing would be visible.

Associated: PSET, POINT

PRINT

Syntax: PRINT [<expression list>]

Example:

```
INPUT "What is your name";yourname$
PRINT "Hello ";yourname$
```

Brief

Displays data in the current Output window.

Description

PRINT on its own prints a blank line. If it is followed by an expression list, then the values of these expressions are displayed on the screen in the current output window.

The individual items may be numeric or string expressions. They can be separated from each other by different punctuation marks which have the following meanings:

<space>	Next item printed adjacent to end of previous one.
;	Next item printed adjacent to end of previous one.
,	Next item printed at start of next zone (as set by WIDTH)

If the expression is terminated by a ';' or ',' then the following PRINT statement will start printing on the same line either adjacent to or at the start of the next zone after the previous item printed. Otherwise, printing will start at the beginning of the next line.

Numbers are always followed by a space and preceded by either a space if they are positive or a minus sign if they are negative. They are printed in scientific notation if they cannot be represented by a maximum of seven digits (for single precision numbers) or sixteen digits (for double precision numbers).

Note that a question mark, '?', can be used instead of the word PRINT in programs to save time.

Associated: PRINT USING, WIDTH, LOCATE

PRINT USING

Syntax: PRINT USING <format string>;<expression list>

Example:

```
PRINT USING "##.##";percent1,percent2,percent3
```

Brief

Displays formatted data in the current output window.

Description

PRINT USING allows strings or numbers to be output to the screen in a particular format. The command must be followed by a string containing special characters which specify how the data is to be formatted. This, in turn, is followed by a semi-colon and the list of expressions to be printed in this way.

The following characters have a special meaning in the format string:

Character	What is to be output
!	Only the first character of each string
\n spaces\	2+n characters of each string
&	The entire string
#	A digit
.	A decimal point
+	A plus or minus sign
–	A trailing minus sign for negative numbers
**	Asterisks in the place of leading spaces
$$	A dollar sign before numbers
,	Commas every three places before the decimal point
^^^^	Numbers in exponential format
_	One of the above symbols as a literal character

If a number is too large to fit into the field allocated for it, a '%' character is printed to indicate that it has overflowed.

Associated: PRINT, LOCATE

PRINT#

Syntax: PRINT# <fileno.>,<expression list>

Example:

```
firstname$ = "Paul"
surname$ = "Smith"
OPEN "O",#1,"junk"
PRINT#1,firstname$;",";surname$
CLOSE
OPEN "I",#1,"junk"
INPUT#1,a$,b$
PRINT a$,b$
CLOSE
```

Brief

Sends data to a sequential file.

Description

This command takes the file number of a file opened for output and a list of expressions. The values of these expressions are sent to this file.

The important point to note about this command is that the data appears in the file almost exactly as it would appear on the screen if PRINT was being used to send it. Therefore, sending two strings separated by a ';' leads to the strings appearing to be contiguous. Then, when they are read in again later using INPUT# they are read in as a single item. Similarly, sending two strings separated by a ',' leads to the two strings being sent with just spaces between them. Again this appears to be one long string later.

Therefore, if you wish to send more than one piece of information at a time, you must make sure that you also send characters which enable them to be identified as separate items. One method of doing this is shown in the example. In this case, an explicit semi-colon character is sent between the items.

You must also be careful that the strings themselves do not contain commas or semi-colons which would lead to the opposite problem – one string appearing to be two when it is read in. One method of

doing this is to send quotation marks (ASCII code 34) before and after each string. For example:

```
PRINT#1,CHR$(34);firstname$;CHR$(34);",";CHR$(34);surname$;CHR
$(34)
```

These double quotes ensure that anything within them will be treated as separate strings. The only problem with this method is that it is rather verbose – WRITE# provides a better method for strings.

Note that this is not a problem with numbers. Numbers are automatically sent with a trailing space which is sufficient to identify where one number ends and the next starts when they are read in again.

Associated: PRINT# USING, WRITE#, OPEN, INPUT#

PRINT# USING

Syntax: PRINT# <file no.>,USING <format string>;<expression list>

Example:

```
n1 = 11.11 : n2 = 22.22 : n3 = 33.33
OPEN "O",#1,"junk"
PRINT#1,USING " +##.##";n1;n2;n3
CLOSE
OPEN "I",#1,"junk"
INPUT#1,a,b,c
PRINT a,b,c
CLOSE
```

Brief

Send formatted data to a sequential file.

Description

This command takes the filenumber of a file opened for output, a string of special characters showing how the data is to be formatted and a list of expressions. The values of these expressions are sent to this file formatted as specified. The points made in PRINT# above about separating individual items also apply to PRINT# USING. The formatting characters are as described in PRINT USING.

Associated: PRINT#, PRINT USING, WRITE#, OPEN, INPUT#

PSET

Syntax: PSET [STEP] (x,y) [,color]

Example:

```
PSET (100,100),1
PSET ( 99,100)
PSET (101,100)
PSET (100, 99)
PSET (100,101)
```

Brief

Plots a point at a specified position.

Description

This command will draw a single point at the position given using the standard coordinate syntax. If the position defined lies outside of the current output window then the command is ignored.

If STEP is introduced into the command, then the coordinates supplied are treated as relative to the current pen position.

The colour of the point may be determined by specifying the colour number to use in <color>. The actual colour will then be as defined with the PALETTE statement. If no colour number is supplied then the specified point is set to the foreground colour.

Associated: PRESET, POINT

PTAB

Syntax: PTAB(X)

Example:

```
FOR loop% = 0 TO 10
 PRINT PTAB(300-loop%);"*";
 PRINT PTAB(310+loop%);"*"
NEXT
```

Brief

Moves the text cursor horizontally to a particular pixel.

Description

PTAB can be used in conjunction with PRINT and LPRINT to move the text cursor sideways to a given pixel position. This allows text to be located exactly, rather than being constrained to be in columns.

```
PRINT PTAB(x)
```

can be used to position the text cursor ready for the next PRINT instruction. Note that a semi-colon need not be given explicitly after it to stop the text cursor moving onto the next line.

Associated PRINT, LPRINT, TAB, SPC

PUT

Syntax: PUT[#]<file no.>[,<record no.>]

PUT [STEP] (X,Y),<name>[,(index[,index...])] [,action]

Examples:

```
OPEN "example" AS 1 LEN = 10
FIELD #1, 10 AS sbuf$
RSET sbuf$ = "Hello"
PUT #1
RSET sbuf$ = "Goodbye" : REM overwrite
PUT #1,1
CLOSE

DIM rect%(626)
PUT(220,90),rect%
```

(see GET)

Brief

Writes a record to a random access file or sends an array of bits to the screen.

Description

PUT has two quite separate functions and these are examined separately below.

Random Access File PUT

Here the command sends the record currently in the random file buffer to the specified file, defined by <file no.>. If a record number is given by <record no.> then the data will be written to this position in the file. Otherwise it will be added to the record number which is one greater than that used by the last PUT.

Screen PUT

Here the command allows a section of screen image which was previously stored in an array to be placed on the screen in the current output window. The bottom left hand corner of this rectangular area is placed at position (X,Y). See GET for details of the array contents.

If a multi-dimensioned array was used with GET to hold several screen images, a list of array indices can be given to PUT so that several images are drawn in succession.

The final optional argument must be one of the following: PSET, PRESET, AND, OR or XOR. This determines how the stored image is to interact with the one already on the screen. The default is XOR. The actions are as follows:

PRESET	overwrites existing graphics with inverse of saved image.
PSET	overwrites existing graphics with saved image.
AND	ANDs each bit of the image with corresponding screen bit.
OR	ORs each bit of the image with corresponding screen bit.
XOR	XORs each bit of the image with corresponding screen bit.

Associated: GET, FIELD, LSET, RSET, OPEN

RANDOMIZE

Syntax: RANDOMIZE [<X>/TIMER]

Example:

```
RANDOMIZE TIMER
FOR count = 1 TO 200
 CIRCLE (RND*620,RND*180),RND*60
NEXT count
```

Brief

Reseeds random number generator.

Description

The sequence of 'random' numbers generated by the RND function depends on what number was used to 'seed' the random number generator. If the same seed is used twice in a row, then the same sequence of numbers will be generated by subsequent RND functions. This command reseeds the random number generator.

Giving an expression, <X>, allows you to set the seed to a particular value so that the results will be repeatable. Using the RANDOMIZE TIMER command uses the number of seconds which have passed since midnight as the seed and so a different sequence will be obtained each time the program is run. If no argument is given, then the program will stop and ask to you to input a number. The value must be in the range –32768 – +32767.

Associated: RND

READ

Syntax: READ <variable list>

Example:

```
FOR loop% = 1 TO 5
 READ name1$(loop%),age%(loop%)
NEXT
:
DATA "Fred",4,"Sam",6,"Jim",4
DATA "Mary",5,"Sue",6
```

Brief

Reads values from DATA statements.

Description

READ can be used to read one or more items at a time from DATA statements and assign them to the variables whose names follow the READ keyword. The type of data read must be compatible with the type of variable it is being assigned to, otherwise an error message will be generated.

DATA statements are read in order and a special read pointer is used by AmigaBASIC to keep the position of the next item to be read. When one DATA line is exhausted, the next is moved to. Trying to read DATA when there is no more data left to read will generate an error message. However, the same data can be used more than once by using the RESTORE statement to reposition the read pointer.

Associated: DATA, RESTORE

REM

Syntax: REM <remark>

Example:

```
REM This is a remark
REM which will not be executed
' This is also a remark
```

Brief

Allows comments to be placed in programs.

Description

REM statements are not executed; everything on the line following the REM keyword will be ignored. However, they are very useful for allowing comments to be added to a program to explain what is happening. If a REM statement is jumped to using GOTO etc, then the first non-REM statement after the REM will be the first to be executed.

Note that an apostrophe can be used instead of the word REM in programs to save time.

RESTORE

Syntax: RESTORE [line no/label]

Example:

```
FOR count% = 1 TO 5
 RESTORE
 number% = FIX(RND*4 + 1)
 FOR count2% = 1 TO number%
      READ number$
 NEXT
 PRINT number$
NEXT
DATA one,two,three,four
```

Brief

Resets the read pointer.

Description

RESTORE repositions the read pointer so that items can be read from DATA statements in a non-sequential manner or even read several times.

If RESTORE is not followed by an argument, then the next READ statement will access the first item in the first DATA statement in the program. This sets the read pointer back to the position it was at before any data had been read.

If RESTORE is followed by a line number or label, then the next READ statement will access the first item in the first DATA statement following the line number/label given.

Associated: READ, DATA

RESUME

Syntax: RESUME [0/NEXT/<line no>/<label>]

Example:

```
ON ERROR GOTO errorhand
INPUT "Input a number :",num
recip = 1/num
:
errorhand:
IF ERR = 11 THEN
 PRINT "infinite"
ELSE
 ON ERROR GOTO 0
END IF
RESUME NEXT
```

Brief

Continues execution after an error handler has been activated.

Description

If error trapping is turned on, then any errors which occur will be dealt with by the error handler in force. RESUME should be used at the end of the error handler in order to continue execution of the main body of the program.

Option	Execution recommences at the statement
RESUME	Which caused the error
RESUME 0	Which caused the error
RESUME NEXT	Immediately after the one which caused the error
RESUME <line>	Following the line number specified
RESUME <label>	Following the label specified

Associated: ON ERROR GOTO

RETURN

Syntax: RETURN [<line no>/<label>]

Example:

```
:
GOSUB delay
:
delay:
  FOR c% = 1 TO 5000
  NEXT
RETURN
```

Brief

Returns execution to the body of the program after a subroutine call.

Description

All subroutines must finish with the RETURN statement. This identifies the end of the subroutine and causes execution to jump back into the main body of the program.

RETURN on its own returns execution to the statement immediately after the subroutine call. If a line number or label is given, then execution will return to the first executable statement following this. This is essentially the same as finishing the subroutine with a GOTO statement and so should be avoided in order to keep programs well structured and readable.

RIGHT$

Syntax: RIGHT$(<string>,<n>)

Example:

```
PRINT RIGHT$("AmigaBASIC",5)
```

Brief

Returns <n> characters from the right-hand side of <string>.

Description

RIGHT$ returns <n> characters from the right-hand side of <string>. If the number of characters asked for is greater than the number of characters in the string, then the entire string is returned. If the number is zero, then the null string is returned. In the example the string "BASIC" is printed.

Associated: LEFT$, MID$

RND

Syntax: RND [(X)]

Example:

```
CIRCLE (RND*620,RND*180),RND*60
```

Brief

Returns a random number between 0 and 1.

Description

RND on its own returns the next 'random' number. These aren't truly random: the sequence of them depends on the number used to seed the random number generator. If an argument is given, its value determines the action as follows:

No.	Effect
<0	Restart the same sequence
>0	Generate next number in sequence (same as no argument)
0	Repeat the last number generated

Associated: RANDOMIZE

RSET

Syntax: RSET <string$>=<string expression>

Example:

```
OPEN "example" AS 1 LEN = 20
FIELD #1, 20 AS sbuf$
RSET sbuf$ = "Fellows"
PUT #1
CLOSE
```

Brief

Transfer data from memory to random access buffer in readiness for PUT.

Description

RSET is used in combination with PUT for writing data to random access files. The purpose of RSET is to move the data to the random file buffer. <string$> (which has been defined as the transfer string in FIELD) is assigned <string expression> and transferred to the file buffer.

RSET will right justify the string if it is shorter than the length assigned by FIELD. Any spaces at the start of the record are padded with spaces. This is particularly useful for lining up numeric values.

Associated: LSET, PUT, FIELD

RUN

Syntax: RUN [<line no>/label]

RUN <filename>[,R]

Example:

```
RUN "Myprog"
```

Brief

Executes a program.

Description

On its own, RUN executes the program currently in memory starting from the beginning. A line number or label can also be given, in which case execution starts from that point. In both cases, the Output window will be moved to the front and cleared and any open files will be closed before execution starts.

If a filename is specified, then the named program is loaded from disc into memory and then executed. A requester will appear if a program was already in memory allowing it to be saved before it is overwritten. This syntax can also be followed by the letter 'R' which indicates that any data files which are open before the program is loaded will be left open so that they may be accessed.

SADD

Syntax: SADD (<string expression>)

Example:

```
A$ = "hello"
PRINT SADD(A$)
```

Brief

Returns the address of the first byte of a string expression.

Description

This function takes a string expression and returns the address of the first byte of data. It is used mainly to pass the address of strings to machine code routines.

Note that after a further string allocation has occurred, the value returned cannot be relied upon because the string may have been moved.

Associated: VARPTR, CALL

SAVE

Syntax: SAVE [<filename>[,A/P/B]]

Example:

```
SAVE "sound1",A
```

Brief

Saves a program to disc.

Description

This command saves the program which is current in memory to disc. If no filename is given, then one will be asked for. Any existing file of the same name will be overwritten.

The options have the following effects:

Option	Effect
A	Saves the file in ASCII format so it can be merged, transferred to other systems etc.
B	Saves the file in tokenised form so that it takes up less disc space (this is the default).
P	Protects the file so that it can be executed but not listed or edited.

Associated: LOAD, MERGE

SAY

Syntax: SAY <phoneme$>[,<mode>]

Example:

```
SAY TRANSLATE$("Have a nice day")
```

Brief

Speaks a list of phonemes using the voice synthesizer.

Description

SAY takes a string which contains a series of phoneme codes and speaks it. The simplest way to create the phonemes is to use TRANSLATE$ which will take a string of normal text and return a string of phonemes. However, TRANSLATE$ is not perfect so, to obtain the pronunciation you want, you may need to construct the series of phonemes yourself. Each phoneme is represented by one or two upper-case characters. These are listed below, together with words which give examples of their sound.

Vowels

AA	cot	AX	about	IX	solid
AE	cat	EH	let	IY	feet
AH	fun	ER	bird	OH	cord
AO	walk	IH	fit	UH	book

Diphthongs

AW	cow	EY	fade	OY	foil
AY	tide	OW	row	UW	flew

Consonants

B	bat	K	cot	T	tap
/C	block	L	lot	TH	this
CH	chat	M	mat	V	very
D	day	N	next	W	wide
DH	the	NX	king	Y	yacht
F	fat	P	pat	Z	zoo
G	get	R	rat	ZH	leisure
/H	hot	S	sat		
J	jam	SH	show		

Vowels and diphthongs may be followed by a digit 1-9 which determines the amount of stress to be placed on the sound. In addition, the punctuation marks; '.', '?', '-', ',', '(' and ')' can be used to help the intonation of sentences.

The optional <mode> is an integer array containing at least nine elements. These elements specify the voice to be used:

Element	Affects	Values	Effect
0	Pitch(hertz)	65	deep
		320	high and squeaky
1	Inflection	0	use inflections
		1	monotone, robot like
2	Rate(words per min)	40	slow
		400	fast
3	Voice	0	male
		1	female
4	Sampling freq (hertz)	5000	low
		28000	high and squeaky
5	Volume	0	silent
		64	loud
6	Channel	0 – 3	channel no
		4	channels 0 & 1
		5	channels 0 & 2
		6	channels 1 and 3
		7	channels 2 and 3
		8	any available left channel
		9	any available right channel
		10	any available pair of channels
		11	any available single channel
7	Mode	0	synchronous
		1	asynchronous
8	Asynchronous speech	0	wait for first to finish
		1	stop speech processing

control	2	interrupt first and execute second

Associated: TRANSLATE$

SCREEN

Syntax: SCREEN <screen no>,<width>,<height>,<depth>,<mode>

Example:

```
SCREEN 1,640,256,4,2
WINDOW 2,"My own output window",,31,1
FOR loop% = 0 TO 15
  CIRCLE (320,90),(loop%+1)*5,loop%
NEXT
```

Brief

Creates a new screen.

Description

This command creates a new screen. Any windows created in this screen will inherit the attributes of it, ie the number of colours which can be displayed at once etc.

<screen no> is the identity number which is associated with the screen. AmigaBASIC can handle up to four additional screens at once, each of which has an different identity number 1, 2, 3, or 4 (the Workbench screen is number –1). This number is used by other commands to tell them which screen is being acted on.

<width> and <height> limit the number of pixels in the screen which can be used.

<depth> determines the number of bits per pixel which can be used. It takes a number in the range 1–5 as described below:

Depth/Bits per pixel	Number of colours
1	2
2	4
3	8
4	16
5	32

<mode> is a number 1–4 which specifies the resolution and whether the screen is interlaced or not as follows:

Mode	Description
1	320 pixels across and 256 lines high
2	640 pixels across and 256 lines high
3	320 pixels across and 512 lines high
4	640 pixels across and 512 lines high

Associated: WINDOW

SCROLL

Syntax: SCROLL (X1,Y1)-(X2,Y2),<right>,<down>

Example:

```
LINE (20,20)-(40,40),,bf
COLOR ,3
FOR i% = 1 TO 10
 FOR j% = 1 TO 10000
 NEXT
 SCROLL (20,20)-(40,40),1,-1
NEXT
```

Brief

Scrolls a rectangular area of the current output window

Description

SCROLL acts on the rectangular area defined. <right> defines the number of pixels to scroll by sideways: it moves to the right if positive and to the left if negative. <down> defines the number of pixels to scroll by vertically: it moves down if positive and up if negative.

The new area will be filled in with the current background colour.

SGN

Syntax: SGN(<num>)

Example:

```
INPUT "Number :",x%
sign% = SGN(x%)
PRINT (ABS(x%)-1)*sign%
```

Brief

Returns the sign of a number as 1 (positive), –1 (negative) or 0 (zero).

Description

This function returns a value as follows:

Argument	Returns
>0	1
<0	-1
0	0

It can be used in conjunction with ABS which finds the absolute value of a number. The example above reduces the absolute value of an integer by 1 whilst still retaining its sign, ie it moves it one closer to zero.

Associated: ABS

SHARED

Syntax: SHARED <variable list>

Example:

```
DIM names$(20)
:
SUB ex STATIC
SHARED x%,y%,names$()
:
END SUB
```

Brief

Makes global variables accessible within subprograms.

Description

This command can only be used within subprograms. It must be followed by a list of variables which will be shared by the subprogram and the main program. Any arrays must be indicated as such by terminating them with brackets as for 'names$' in the example. (However, arrays can also be shared by using DIM SHARED when creating them.)

When a variable is shared, only one version of it exists. Assigning a value to it within the subprogram will alter its value in the main program and vice versa. If a variable is not shared, then assigning a value to it in the subprogram will have no effect on the value of the variable of the same name in the main program (and vice versa).

Associated: DIM SHARED, SUB

SIN

Syntax: SIN(X)

Example:

```
pi2 = 3.14159*2
FOR angdeg% = 0 TO 360 STEP 10
 PRINT SIN(pi2*angdeg%/360)
NEXT
```

Brief

Returns the sine of X.

Description

The sine of X is evaluated in the precision of the value supplied, ie a single precision value is evaluated as a single precision number and a double precision value is evaluated as a double precision number. The value of X is expected in radians.

Associated: COS, ATN

SLEEP

Syntax: SLEEP

Example:

```
WHILE 1 = 1
SLEEP
PRINT "Awake"
WEND
```

Brief

Suspends execution of a program until an event occurs.

Description

This command stops the program from executing anything until some event occurs which it recognises. Note that it isn't necessary for event trapping to be turned on for the event to have an effect. Events which end this temporary suspension are:

- A key being pressed.
- The left-mouse button being pressed.
- A menu entry being selected.
- A timer event occurring.
- A collision between two objects.

SOUND

Syntax: SOUND <freq>,<duration> [,[volume][,voice]]

Example:

```
SOUND 523.25,18.2,255
```

Brief

Produces a sound.

Description

SOUND produces a sound which is determined by the arguments it is given:

<freq> determines the frequency and hence pitch of the note. Values are expected in hertz. Any value outside the range 20–15000 will produce the minimum or maximum sound as appropriate. The following table shows the values for the middle octave – doubling the value produces the note one octave higher.

Note	Frequency
A	440.00
B	493.88
C	523.25
D	587.33
E	659.26
F	701.00
G	783.99

<duration> determines how long the note lasts. A value of 18.2 produces a note which lasts for one second; doubling this value makes the note last twice as long. Any value between 0 and 77 can be given.

<volume> can be used to produce a note which is silent (0) to very loud (255). The default is 127.

<voice> indicates which channel the sound will be on. Channels 0 and 3 are connected to the left speaker and channels 1 and 2 to the right. The default is 0.

The example produces a middle C lasting for 1 second at maximum volume from the left speaker.

Associated: WAVE

SOUND RESUME/WAIT

Syntax: SOUND RESUME/WAIT

Example:

```
SOUND WAIT
SOUND  523.25, 9.1, 127, 0
SOUND  659.26, 9.1, 127, 1
SOUND  783.99, 9.1, 127, 2
SOUND 1046.50, 9.1, 127, 3
SOUND RESUME
```

Brief

Stores sounds or starts to play previously stored sounds.

Description

SOUND WAIT stops further SOUND commands from being executed. Instead they are stored until a SOUND RESUME is executed. This enables sounds on different channels to be all played together and so enables different parts to be synchronised with each other.

Note that the buffer used to store the sounds is not infinite. An error occurs if it becomes full.

SPACE$

Syntax: SPACE$(<num>)

Example:

```
sp20$ = SPACE$(20)
PRINT "Hello";sp20$;"there"
```

Brief

Returns a string containing <num> spaces.

Description

This function rounds the argument to the nearest integer and returns a string containing that number of spaces. The argument must be in the range 0–32767.

Associated: SPC

SPC

Syntax: SPC(X)

Example:

```
PRINT "Hello";SPC(20):"there"
```

Brief

Generates X spaces in a PRINT statement.

Description

SPC can be used in conjunction with PRINT and LPRINT to output spaces. X must be between 0 to 255.

```
PRINT SPC(X)
```

This function can be used to position the text cursor ready for the next PRINT instruction. Note that a semi-colon need not be given explicitly after it to stop the text cursor moving onto the next line.

Associated: PRINT, LPRINT, TAB, PTAB, SPACE$

SQR

Syntax: SQR(X)

Example:

```
PRINT SQR(100)
```

Brief

Returns the square root of X.

Description

The square root of X is evaluated in the precision of the value supplied, ie a single precision value is evaluated as a single precision number and a double precision value is evaluated as a double precision number. The value of X must be greater than or equal to zero.

STICK

Syntax: STICK(N)

Example:

```
xpos% = xpos% + STICK(0)
ypos% = ypos% - STICK(1)
```

Brief

Returns a joystick direction.

STICK takes one argument: the number of the joystick you want to read times two, plus one if you are reading the Y vertical movement:

Argument N	Meaning
0	Investigate X movement of joystick 1
1	Investigate Y movement of joystick 1
2	Investigate X movement of joystick 2
3	Investigate Y movement of joystick 2

The value returned is either –1, 0 or +1 which have the following meanings:

Return Value	Meaning
-1	Movement upwards or to the right
0	No movement
1	Movement downwards or to the left

Associated: STRIG

STOP

Syntax: STOP

Example:

```
:
REM This branch should not be taken
STOP
```

Brief

Halts program execution and returns to immediate mode.

Description

STOP is very useful when debugging a program. It allows the code to be interrupted at particular positions to investigate what is going on. Execution can then be continued using CONT. Note that STOP does not close any open files.

Associated: CONT

STRIG

Syntax: STRIG(N)

Example:

```
If STRIG(0) PRINT "Fire"
```

Brief

Returns a joystick's button information.

Description

STRIG takes one argument whose meaning is as follows:

Argument N	Meaning
0	Investigate if button 1 was pressed since last call.
1	Investigate if button 1 is currently pressed.
2	Investigate if button 2 was pressed since last call.
3	Investigate if button 2 is currently pressed.

Hence the values 1 and 3 investigate the current status of the button on one of the joysticks whereas the values 0 and 2 determine whether a button press has occurred since the previous time STRIG(0) or STRIG(2) was used. The values 0 and 2 therefore prevent button presses being lost due to them occurring when the program is not checking for them.

The value return by STRIG is as follows:

Return Value	Meaning
0	Button is not / has not been pressed
1	Button is / has been pressed

Associated: STICK

STR$

Syntax: STR$(<num>)

Example:

```
PRINT STR$(1.23)
PRINT STR$(1)+STR$(2)+STR$(3)
PRINT STR$(1+2+3)
```

Brief

Returns the string representation of <num>.

Description

STR$ takes a number and generates a string which starts with either a space (for positive numbers) or a minus sign (for negative numbers) and contains one character for each of the digits etc in the string. In the examples, the strings " 1.23", " 1 2 3" and " 6" are printed.

Associated: VAL

STRING$

Syntax: STRING$(<I>,<J>)

STRING$(<I>,<string$>)

Example:

```
PRINT STRING$(40,"*")
```

Brief

Returns a string containing multiple copies of a single character.

Description

The first syntax returns a string containing <I> copies of the character whose ASCII code is <J>. The second returns a string containing <I> copies of the first character of <string$>. The example prints 40 asterisks in a row.

SUB...STATIC

Syntax: SUB <subprog name>[(parameter list)]STATIC

Example:

```
:
CALL delay(1000)
:
SUB delay(timewait%) STATIC
FOR c% = 1 TO timewait%
NEXT
END SUB
```

Brief

Defines the start of a subprogram.

Description

A subprogram is essentially a mini program. Each subprogram must have a unique name given by <subprog name> which can contain up to 30 characters. This can be followed by an optional parameter list which contains a series of variable names separated by spaces. Arrays can be given in this list provided that they are followed by brackets. Finally the keyword STATIC must be given.

The statements within the subprogram are only executed when a CALL is made to the subprogram from within the main body of the program. The number of arguments used in the call statement must equal the number of parameters in the parameter list for the subprogram. In addition, the values passed must be compatible with the types of the parameter variables.

Execution returns to the statement after the subprogram call when either an END SUB or EXIT SUB statement is reached. END SUB marks the end of the body of the subprogram code. EXIT SUB can be used anywhere within the subprogram in order to end execution of the subprogram at other positions.

There are certain statements which you are not allowed to use inside subprograms. These are:

- User-defined function definitions.
- Subprogram definitions (ie subprograms cannot be nested).
- COMMON statements.
- CLEAR statements.

Associated: END SUB, EXIT SUB, SHARED, CALL

SWAP

Syntax: SWAP <var1>,<var2>

Example:

```
IF max% < min% THEN SWAP max%,min%
```

Brief

Exchanges the values of two variables.

Description

SWAP takes two variables of the same type and exchanges their values. It is equivalent to:

```
temp = var1
var1 = var2
var2 = temp
```

Note that the second variable must have already been defined but the first one need not have been and will be initialised to zero.

SYSTEM

Syntax: SYSTEM

Brief

Returns to the Workbench.

Description

Giving the SYSTEM command is equivalent to selecting the Quit option from the AmigaBASIC Project menu. It closes any open files, produces a requester if any edited programs exist in memory to allow them to be saved and leaves the AmigaBASIC system.

TAB

Syntax: TAB(X)

Example:

```
PRINT TAB(4);"Name";TAB(20);"Age"
PRINT TAB(4);"Michael Smith";TAB(20);14
PRINT TAB(4);"Frederick Browning";TAB(20);15
```

Brief

Moves text cursor to column X.

Description

TAB can be used in conjunction with PRINT and LPRINT to position the text cursor in a particular column. The left-hand column is numbered 1 and the value given must be in the range 1–155.

If the cursor is already to the right of the column asked for, then it will move to that column on the next row down. This is illustrated by the example which produces:

Name	Age
Michael Smith	14
Frederick Browning	15

PRINT TAB(X) can be used to position the text cursor ready for the next PRINT instruction. Note that a semi-colon need not be given explicitly after it to stop the text cursor moving onto the next line.

Associated PRINT, LPRINT, PTAB, SPC

TAN

Syntax: TAN(X)

Example:

```
pi2 = 3.14159*2
INPUT "Angle in degrees :",angdeg%
PRINT TAN(pi2*angdeg%/360)
```

Brief

Returns the tangent of X.

Description

The tangent of X is evaluated in the precision of the value supplied, ie a single precision value is evaluated as a single precision number and a double precision value is evaluated as a double precision number. The value of X is expected in radians.

Associated: COS, SIN, ATN

TIME$

Example:

```
PRINT TIME$
```

Brief

Returns the current time.

Description

The TIME$ function reads the system clock and returns an eight character string in the form: hh:mm:ss

The response at half past eight in the morning would be: 08:30:00

Associated: DATE$, TIMER

TIMER

Syntax: TIMER

Example:

```
tim1& = TIMER
:
PRINT TIMER-tim1&;"seconds taken"
```

Brief

Returns the number of seconds which have passed since midnight.

Description

The TIMER function reads the system clock and returns an integer containing the number of seconds which have passed since midnight. It is useful for timing how long sections of program take to execute.

TIMER can also be used in conjunction with RANDOMIZE for reseeding the random number generator.

Associated: TIME$, RANDOMIZE

TIMER ON/OFF/STOP

Syntax: TIMER ON/OFF/STOP

Example:

```
ON TIMER(60) GOSUB min1
TIMER ON
WHILE 1 = 1
WEND
min1:
PRINT "Minute gone by"
RETURN
```

Brief

Enables, disables, or suspends event trapping based on the passage of time.

Description

The timer, noticing that a given number of seconds have passed, can be used to cause an event. An ON TIMER...GOSUB statement is used to specify the number of seconds and set up a handler for such an event.

TIMER ON activates timer event trapping so that any such events will be noted and acted on immediately by executing the handler.

TIMER OFF inactivates timer event trapping so any such events are ignored.

TIMER STOP leaves timer event trapping activated but does not act on the events immediately. Instead it waits until a TIMER ON statement is executed and then executes the handler.

Associated: ON TIMER ... GOSUB

TRANSLATE$

Syntax: TRANSLATE$(<string$>)

Example:

```
SAY(TRANSLATE$("Hi there"))
```

Brief

Converts a sentence into a string of phonemes suitable for SAY.

Description

TRANSLATE$ returns a string containing a series of phonemes which represent the text it was given. This string is in a format which is suitable to be passed straight to the SAY command so that it can be spoken using the speech synthesiser. Note that English is a complex and irregular language and TRANSLATE$ is not perfect. Sometimes, better results can be obtained if you spell words differently. For example: SAY(TRANSLATE$("Dayter")) produces a better English pronunciation of the word 'data' than the correct spelling does.

Associated: SAY

TROFF

Syntax: TROFF

Brief

Turns program tracing off.

Description

Giving the command TROFF is equivalent to selecting the Trace Off option from the Run menu. It causes the tracing of statements to be turned off so the remainder of the program is executed normally.

Associated: TROFF

TRON

Syntax: TRON

Brief

Turns program tracing on.

Description

Giving the command TRON is equivalent to selecting the Trace On option from the Run menu. It causes the statement being executed to be highlighted in the List window, if the window is visible. Thus allowing the path through the program and the effect of the individual statements to be seen.

Tracing is disabled if TROFF is used or a NEW command is given.

Associated:TROFF

UBOUND

Syntax: UBOUND <array name>[,<Dim>]

Example:

```
DIM array%(15)
FOR loop% = LBOUND(array%) TO UBOUND(array%)
 array%(loop%) = RND*10
NEXT
```

Brief

Returns the upper bound of an array dimension.

Description

This function returns the upper BOUNDary of the named array, <array name>. Used in conjunction with LBOUND it allows the values used within a DIM and OPTION BASE statement to be found.

UBOUND returns the value allocated by the DIM statement. Therefore, if the array has not been DIMensioned, the default value of ten will be returned.

The use of <Dim> is optional and is used to indicate which dimension in a multidimensional array is to be used. The default value is 1, meaning the first dimension. Giving a dimension which does not exist will produce the result 0.

This function is particularly useful for subprograms. It allows them to accept arrays of varying sizes and be able to handle them without having to be explicitly told their number of dimensions and size of each.

Associated: LBOUND

UCASE$

Syntax: UCASE$(<string expression>)

Example:

```
repeat:
INPUT "Continue Y/N :",ans$
ans$ = UCASE$(ans$)
IF ans$ = "N" THEN
 END
ELSEIF ans$ = "Y" THEN
 PRINT "Continuing"
ELSE
 GOTO repeat
END IF
```

Brief

Copies a string and converts all characters to upper-case.

Description

The string which is passed in <string expression> is not altered. A copy of it is taken, any lower-case characters in it are converted to upper-case and this new string is returned.

UCASE$ is very useful when dealing with user input since it reduces the number of checks which have to be made. For example, if you are testing to see if the user input the word "YES", you would have to otherwise test against "YES", "yes", "Yes" etc. Using UCASE$ to convert their input means that only the one test against "YES" is required.

In addition, it enables strings to be sorted alphabetically regardless of their case. If case is taken into account then all lower case letters are listed after upper case ones.

VAL

Syntax: VAL(<string$>)

Example:

```
PRINT VAL(TIME$);" hours"
```

Brief

Returns the numeric value of the string <string$>.

Description

The function VAL takes a string of digits and converts it into a number. It ignores any space or tab characters at the start of the string and returns the value of string up to the first character which cannot be treated as part of a number. In the example, only the hours part of the time will be returned as a number because that is separated from the minutes by a ':'.

Note that it is not only digits which can be treated as valid components of a number. PRINT VAL("1E2") prints the value 100. Although the 'E' is a not a digit, 1E2 is a valid way of representing a number using exponential format. Similarly, preceding '+' and '-' characters are handled correctly.

If, however, the characters of the string (ignoring spaces) do not start with a digit or a plus or minus sign, then VAL returns 0.

Associated: STR$

VARPTR

Syntax: VARPTR (<variable name>)

Example:

```
A% = 1
addr& = VARPTR(A%)
PRINT PEEKW(addr&)
```

Brief

Returns the address of the contents of a variable.

Description

This function takes a variable name and returns the address of the first byte of data identified with it. For numbers, the address is the location of the (start of) the number itself. For strings, it is the first byte of the string descriptor rather than the first character of the string.

VARPTR is for use mainly when calling machine code routines. Normally, this is done by loading the machine code into an array such as code% and using:

```
start% = VARPTR(code%(0))
CALL start%
```

to execute the code. Values can be passed to the machine code routine in a similar manner.

Associated: SADD, CALL

WAVE

Syntax: WAVE <voice>,<wave def>/SIN

Example:

```
DIM wf%(255)
FOR loop% = 0 TO 255
 wf%(loop%) = 127*RND
NEXT
WAVE 0,wf%
SOUND 523.25,18.2,255
```

Brief

Defines the waveform for a sound channel.

Description

The WAVE statement requires two arguments. The first is simply the channel which you want to assign the waveform to. Any sounds subsequently produced on that channel will use the waveform assigned to it. The second is either the word 'SIN' to select the default wave form or an array containing at least 256 integer elements. Each of these represents a point of the waveform. The maximum value of an element is 127 which means that the waveform is at the maximum height above the middle line. The minimum value is –128 which means that the waveform is at the greatest distance below the line.

WAVE is a very versatile command which allows a wide range of sounds to be produced from squeaks and buzzes to simulated string instruments.

Associated: SOUND

WEND

Syntax: WEND

Brief

Marks the end of a WHILE...WEND loop.

Description

See WHILE...WEND

WHILE...WEND

Syntax: WHILE <logical exp> [statements] WEND

Example:

```
tim& = TIMER
WHILE TIMER - tim& < 30
CIRCLE (RND*640,RND*180),RND*60,RND*3
WEND
```

Brief

Executes a group of statements whilst a given condition remains true.

Description

WHILE must be followed by an expression which gives the result true or false. This is normally followed by a series of statements which may be split over several lines and are terminated by a WEND. If the expression is true, then each of the statements is executed until the WEND is reached. Then control jumps back to the start of the loop and the expression is evaluated again. If the expression gives the result false at any stage, then the statements inside the loop are jumped over and execution starts at the first statement after the WEND.

Each WHILE statement must have one and only one WEND associated with it. However, WHILE statements can be nested inside each other in which case, when a WEND is executed, it is matched with the most recent WHILE which is still being used. Note that jumping into the middle of a WHILE...WEND loop can lead to problems because AmigaBASIC will not know what to do when it reaches the WEND but has not encountered the WHILE associated with it.

WIDTH

Syntax:

WIDTH [<output dev>,/<file no.>,/LPRINT] [<size>] [,<zone>]

Example:

```
WIDTH 40
PRINT STRING$ (50,"*")
```

Brief

Sets the line width and zone width.

Description

WIDTH affects the appearance of text output. The first argument specifies which output device is to be affected. If this is missing, then screen output is assumed. However, a particular device can be specified by giving the device name, <output dev>, the printer can be specified explicitly by using 'LPRINT' or a filenumber can be given if output to a file is to be affected, <file no.>. Note that when changing the line width or zone size for a file, the new settings only take effect the next time the file is opened using the OPEN command.

The next argument, <size>, is the number of standard characters which may be contained on one line. When this number is reached, a carriage return will automatically be inserted. 255 is treated in a special manner. Instead of specifying the number of characters, it is treated as meaning that the line is infinitely wide. 255 is the default value for the screen, 80 is the default value for a printer.

The final, optional argument, <zone> specifies the zone width. This is used to divide each line up. When a comma is used to separate items for PRINT and LPRINT statements, this forces the text cursor to move to the start of the next zone. The default value is 15.

Associated: PRINT, LPRINT, PRINT#

WINDOW

Syntax:

WINDOW <id> ,/<title>[,/(x1,y1)-(x2,y2)[,/<type>[,<sc id>]]
WINDOW <id>
WINDOW <n>

Example:

```
SCREEN 1,640,256,4,2
WINDOW 2,"New",(100,50)-(200,100),31,1
COLOR 5,6
CLS
PRINT WINDOW(6)
```

Brief

Either creates a new output window, makes an existing output window current or returns information about the current output window.

Description

WINDOW has three separate uses and these are examined separately below.

Creating New Windows

The WINDOW statement can be used to create an output window. The first argument is the identity number of the window. This can be any positive integer value. The AmigaBASIC output window has the number 1. Using a number greater than this will create an additional window rather than re-defining the existing one.

<title> is a string which will be used as the title of the window.

(x1,y1)-(x2,y2) defines the rectangular area of the window. If this argument is omitted, the window will be created at the default size for the screen which is the full screen size.

<type> determines the features which the window will contain. Each feature has a different value:

Value	Description
1	Sizing gadget provided to allow mouse control of window size.
2	Window can be moved about using Title Bar.
4	Back gadget provided to allow window to be moved to front/back.
8	Close gadget provided to allow window to be closed using mouse.
16	Window is redrawn if a window in front of it is moved.

The number used is the sum of all the values of the features required.

<sc id> is the identity number of the screen which is to contain the window. The default is to take the value -1 to add it to the Workbench screen. However, other numbers can be given to add it to your own screens which you have defined.

When the window is created, it is automatically selected as the current output window. Therefore the remainder of the program sends all its graphics output to this new window.

Making a Window Current

The WINDOW <id> statement makes the window whose identity number is <id> into the current output window. All text and graphics will be directed to this window. In addition, this window becomes the selected (active) window; it is brought to the front of the screen and highlighted.

Returning Information About a Window

The WINDOW <n> function returns information about the current or selected window. The type of information returned is determined by <n> as follows:

<n>	Returned information
0	Window identity no. for the current output window
1	Window identity number
2	Width
3	Height
4	X co-ordinate of output cursor
5	Y co-ordinate of output cursor

6 Maximum colour number allowed
7 Pointer to the Intuition window
8 Pointer to the RASTPORT

The last two of these are only required if you are calling operating system routines to manipulate the window.

Associated: SCREEN, WINDOW CLOSE/OUTPUT

WINDOW CLOSE/OUTPUT

Syntax: WINDOW CLOSE/OUTPUT <id>

Example:

```
WINDOW OUTPUT 1
```

Brief

Closes or makes current an Output window.

Description

Both commands affect the window whose identity number is given by <id>.

WINDOW CLOSE makes the window invisible.

WINDOW OUTPUT makes the window into the current output window. All text and graphics will be directed to this window. However, it is not selected: hence it is not brought to the front of the screen or highlighted.

Associated: WINDOW

WRITE

Syntax: WRITE [<expression list>]

Example:

```
INPUT "What is your name";yourname$
WRITE "Hello",yourname$
```

Brief

Displays data in the current output window.

Description

WRITE, on its own, prints a blank line. If it is followed by an expression list, then the values of these expressions are displayed on the screen in the current output window.

The individual items may be numeric or string expressions which must be separated by commas.

The items are output with commas between them. Numbers do not have any spaces added before or after them. Strings are surrounded by quotation marks. In the example, the following type of output will be produced:

```
"Hello","Fred"
```

Associated: PRINT, PRINT USING

WRITE#

Syntax: WRITE# <file no.>,<expression list>

Example:

```
WRITE#1 a$,b$,x#,y#
```

Brief

Writes data to a sequential file.

Description

WRITE# should be followed by the number of the file to which the data is to be sent, <file no.>. This, in turn, is followed by an expression list containing the items to be sent. The individual items may be numeric or string expressions and must be separated by commas.

The items are output with commas between them. Numbers do not have any spaces added before or after them. Strings are surrounded by quotation marks. A carriage return is always inserted at the end of each expression list – no terminating punctuation character should be placed here.

WRITE# is often the best way of sending data to a file. It eliminates the need to explicitly add delimiters which is often necessary with PRINT#.

The individual items may be numeric or string expressions which must be separated by commas.

The items are output with commas between them. Numbers do not have any spaces added before or after them. Strings are surrounded by quotation marks.

Associated: PRINT#

B : Error Messages

The following is a list of error messages which can occur when using AmigaBASIC. Some of these have error numbers associated with them. These are the errors which can occur whilst a program is running. They can be trapped using the ON ERROR and ERR statements.

Some errors are reported before the program starts to run. Many of these are caused by unpaired FOR and NEXT or WHILE and WEND or IF and END IF or SUB and END SUB statements. Note that occasionally the message may be misleading. For example, a 'SUB without END SUB' message can be caused by a subprogram containing other structures which are wrongly paired. Note also that these errors can be more easily located and largely eliminated by using correctly indented code.

Advanced Feature (73)
This should never occur.

Argument count mismatch (37)
You have called a subprogram and passed it a different number of arguments to the number declared in its definition.

Bad file mode (54)
This occurs when you try to use commands which aren't compatible with the files you are acting on. For example PUT, GET or LOF with sequential files, LOAD on a random access file, or MERGE on a non ASCII file. The other situation in which this can occur is when you have used a file mode other than 'A', 'O', 'I' or 'R' with the OPEN command.

Bad file name (64)
An illegal file name has been used. The format required is:

```
"<device name or
drive>:<directory>/.../<directory>/<filename>"
```

with each file or directory name being limited to 30 characters and the total pathname being limited to 255 characters.

For example:

```
"DemoDisk:BASIC/Strings/Search"
```

Possible devices are :

SCRN:	Screen
KYBD:	Keyboard
LPT1:	Printer
COM1:	Serial port

Bad file number (52)
An attempt has been made to reference a file with a file number which does not correspond to an open file. When OPEN is used, it assigns a number to the file being opened and this number must then be used whenever that file is to be accessed using commands such as PRINT#.

Bad record number (63)
This occurs when GET or PUT is used with a record number outside the range 1 to 16777215.

BLOCK ELSE/END IF must be the first statement on the line
An ELSE or END IF statement has been found on a line containing other statements.

Can't continue (17)
An attempt has been made to continue the execution of a halted program after the program has been altered or after an error has occurred.

Deadlock (77)
This should never occur.

Device I/O error (57)
The operating system failed to complete a disc access operation, possibly due to the disc being faulty. The Workbench will need reloading after this has occurred.

Device Unavailable (68)
An attempt has been made to input or output information from a device which BASIC cannot get access to.

Disk full (61)
BASIC has attempted to save data to a disc but insufficient room exists on it. The solution is to delete some of the files using 'KILL' or to save the data onto a different disc.

Division by zero (11)
An attempt has been made to divide a number by 0. Note, this can often occur in expressions involving division due to typing mistakes. Any variable takes the default value of 0, so if you type 'PRINT 1/xos%' instead of 'PRINT 1/xpos%' then this error will probably be generated.

Duplicate Definition (10)
This occurs if you try to dimension an array which has already been dimensioned. An obvious cause of this is to include the array in two definition statements. However two less obvious causes exist. One is when an element of an array has been referred to before it is DIMmed. This is because referencing an array element automatically assigns the default dimension of 10 to the array. The second is using OPTION BASE after a DIM statement, since this can alter the number of elements in the array.

If you wish to redefine an array, the ERASE command must be used to erase the contents of the first definition before the second is used.

Duplicate label (33)
The same label has been used more than once within a program.

ELSE/ELSEIF/END IF without IF
An ELSE, ELSEIF or END IF has been found although no corresponding IF statement exists. This is often due to a statement being placed after the THEN. This converts the IF statement into a single line IF rather than a block structured one.

EXIT SUB outside of a subprogram
An EXIT SUB has been found outside of the definition of a subprogram. It can only be used between the SUB and END SUB statements.

FIELD overflow (50)
The number of bytes allocated in a FIELD statement has exceeded the buffer size assigned when the OPEN command was used to open the random access file.

File already exists (58)
NAME has been used to change the name of a file to the name of another existing file.

File already open (55)
Either OPEN has been used to try to open a file that is already open or KILL has been used to try to delete a file which is open.

File not found (53)
The file being accessed could not be found. If just a filename was given this is possibly because the wrong directory is currently selected.

FOR without NEXT (26)
A FOR statement has been found which does not have a corresponding NEXT statement. Either it has been missed out altogether or the loop variable name for it has been mistyped.

IF without END IF
A block structured IF statement has been started but no corresponding END IF could be found. Possibly, the IF was meant to be a single line IF but no statement was typed after the THEN.

Illegal direct (12)
Most commands can be used directly by typing them in the output window. However, others such as DEF FN can only occur within programs. This error is given when one of the latter types of commands has been used directly.

Illegal function call (5)
This is a common error which occurs whenever you attempt to give an argument which is out of range or of the wrong type to a BASIC function. For example 'LOG(-1)'.

It will also occur if a negative array element is used, for example A(-1).

Input past end (62)
This error is given when you try to read data after the end of a file has been reached. EOF should be used to avoid this happening.

Internal error (51)
This should never occur.

Line buffer overflow (23)
This occurs within the editor if an attempt is made to create a line containing more than 255 characters. (It can also occur when the editor gets confused – if this happens, split the line and concatenate it again.)

Missing operand (22)
An operator such as '*' or 'AND' has been used without an expression either side of it.

Missing STATIC in SUB statement
A SUB statement has been found which doesn't contain the obligatory STATIC statement after the name or parameter list.

NEXT without FOR (1)
A NEXT statement has been found which does not have a corresponding FOR statement. Either it has been missed out altogether or the loop variable name has been mistyped.

No RESUME (19)
An error handling routine set up by an ON ERROR statement has been entered but no RESUME statement has been found to return control back out of the handler.

Out of DATA (4)
A READ statement has been used when there is no more DATA left unread. This could be because RESTORE had been used to set the data pointer to the wrong place.

OUT OF HEAP SPACE (14)
This is a special message which means that the system hasn't got enough workspace left to continue executing your program. More heap memory can be obtained by closing down other applications and by using the CLEAR command to reallocate the memory available.

Out of memory (7)
This indicates that the BASIC data area is out of memory. Use CLEAR to increase the memory available for your program and its variables. Alternatively, try reducing the size of your program as described in Chapter 11.

Overflow (6)
This occurs when the result of a calculation is larger than the maximum value allowed for the number format being used. If single precision was being used, changing to double precision may help.

Permission Denied (70)
An attempt has been made to write to a disc which is write protected.

Rename across discs (74)
A different disc name has been specified when trying to rename a file using the NAME command. NAME cannot be used to move a file from one disc to another.

RESUME without error (20)
A RESUME statement has been encountered although an error handler hasn't been activated. This is normally because an END statement is missing between the end of the main program and the definition of the error handler so execution fell through into the error handling code.

RETURN without GOSUB (3)
A RETURN statement has been encountered although a GOSUB hasn't been executed. This is normally because an END statement is missing between the end of the main program and the subroutine code so execution fell through into the body of the subroutine accidentally.

SHARED outside of a subprogram
A SHARED statement has been found outside of the definition of a subprogram. It can only be used between the SUB and END SUB statements. It makes sense to place the SHARED statement at the start of a subprogram both for clarity and to prevent this occurring.

Statement illegal within subprogram
Certain commands may not be used within subprograms. These are: 'DEF FN', 'COMMON' and 'CLEAR'. Note also that 'SUB' may not be used either, although this results in a different message being given.

String formula too complex (16)
This can occur if a very long or complicated string expression has been used involving deeply nested string commands such as MID$. The solution is to split the expression up into separate stages, using temporary variables if necessary.

String too long (15)
An attempt has been made to create a string containing more than 32767 characters.

SUB already defined
Two subprograms with the same name have been found within a program.

SUB without END SUB
A SUB statement has been found which does not have a corresponding END SUB.

Subprogram already in use (36)
A subprogram has been called from within itself, which is not allowed. This could occur because a direct CALL has been used from one subprogram to itself or because one subprogram has CALLed a second which has CALLed the first etc.

Subscript out of range (9)
An attempt has been made to access an array element which does not exist. This could be because the subscript is greater than the value used when the array was dimensioned (or 10 if no DIM statement has been used). Alternatively it can be given because the wrong number of subscripts have been given.

Syntax error (2)
This is a common error which can occur for a number of reasons. These include having unequal numbers of opening and closing brackets round expressions or using the wrong punctuation within a statement.

Too many files (67)
This should never occur.

Too many subprograms
This occurs when a program is found to contain more than 255 subprograms.

Tried to declare SUB within a SUB
A subprogram has been found which contains the definition of another subprogram. Note that there are other restrictions over what

subprograms can contain, which are covered by a different error message.

Type mismatch (13)
This happens when string variables are assigned numeric values or vice versa. It can also occur when an attempt is made to SWAP the values of two variables of different numeric types.

Undefined array (38)
An array has been referenced in a SHARED statement before it has been created.

Undefined label (8)
A label has been used in a GOTO, GOSUB, RESTORE etc statement which does not exist.

Undefined subprogram (35)
This is a common error which occurs when a command is misspelt. BASIC, because it doesn't recognise the command, looks for a subprogram of that name and because it doesn't find one gives the message.

Alternatively, you may really have used a CALL statement to access a subprogram which is not defined.

Undefined user function (18)
A user defined function has been used before its definition has been executed.

Unknown volume (49)
You have given a disc name which cannot be found.

Unprintable error (many)
This should not occur.

WEND without WHILE (30)
A WEND statement has been found which does not have a corresponding WHILE statement.

WHILE without WEND (29)
A WHILE statement has been found which does not have a corresponding WEND statement.

Glossary

Address
An address is an integer number which identifies a particular memory location. Each memory location has a different address and so individual ones can be identified to allow data to be stored in them or read from them.

AmigaDOS
AmigaDOS is the name given to the Amiga's Operating System.

Argument
An argument is another name for a parameter. It is a piece of data which is passed between the main program and a routine which it is calling. For example, the AmigaBASIC function SIN takes one argument which is the angle whose sine is to be returned.

Arithmetic Operators
Arithmetic operators are operators which act on numbers to produce a numerical result. Some take just one operand, ie '+' (unary plus), '-' (unary minus). Others require two to produce a result: these are: '+' (addition), '-' (subtraction), '*' (multiplication), '/' (division), '^' (raise to the power), 'MOD', (integer remainder) and '\' (integer division).

ASCII
ASCII stands for American Standard Code for Information Interchange. It is a code adopted by most computer manufacturers whereby the numbers between 1 and 128 are used to represent different characters. The first 32 represent control characters, the other 96 represent the printable characters. For example, the letter 'a' has an ASCII code of 97.

Assembler
An assembler is a program which converts assembly language statements into machine code. It is the means by which the speed and versatility of machine code can be obtained without having to write directly in the binary numbers of machine code.

Assembly Language Statements
These are statements which have a one-to-one mapping onto machine code instructions. They consist of mnemonics such as CMP, BEQ followed by the operands associated with them. An assembler is required to convert them into machine code which the CPU can execute.

BASIC
BASIC stands for Beginners All-purpose Symbolic Instruction Code. It is a computer language which is used extensively in home micros because it is easy to learn and yet powerful enough to allow a wide variety of applications to be produced using it.

Binary
Binary is the base two number system in which all numbers are represented by just the digits 0 and 1. Whereas in decimal a one in a particular column represents a power of ten, in binary it represents a power of two. Therefore, the first ten binary numbers are: 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010.

Bit
Bit stands for binary digit. It takes one of two values, either 0 or 1. Inside the computer, bits are used to represent the two possible states of switches, either off or on.

Blitter
Blitter stands for 'bit blatter'. It is a co-processor which specialises in copying and manipulating areas of memory very quickly. This enables rectangular areas of the screen or 'bobs' to be moved around to create animated scenes.

Bob
Bob stands for blitter object block. Bobs are one of the types of moveable objects which the Amiga provides (the other being sprites). Each type is handled by a different co-processor, bobs being looked after by the one known as 'Agnes'. Bobs have certain advantages over sprites. Their size is unlimited; they can contain up to 32 colours depending on the screen mode in operation and any number of them can be displayed at once. However, they move only slowly and have a tendency to flicker.

Buffer
A buffer is an area of memory inside the Amiga which is used as a temporary storage space for information being transferred between different devices. For example, the keyboard buffer stores characters entered at the keyboard until the CPU is ready to handle them. Similarly, the disc buffers store data to be saved to disc so that characters can be written out in blocks rather than individually which would be very slow.

Bug
A bug is an error in a program. It can take one of many different forms. For example a syntax error is a mistake in the format used when writing a BASIC statement. This sort of error will be found as soon as you try to execute the program. Logic errors are problems caused by the BASIC you have written not doing what you intended it to. The syntax of the program is correct so the program will run but the results will not be as expected. These are both examples of bugs.

Byte
A byte is eight bits. It can be used to represent integers between 0 and 255 (00000000 to 11111111 in binary notation). Hence bytes can be used to store characters using the ASCII notation. However most bytes in memory form part of larger structures called 'words'.

CPU
CPU stands for Central Processing Unit. This is the chip inside the machine which performs the main functions of the computer. The CPU of the Amiga is a Motorola MC68000 which can execute millions of machine code instructions per second.

Co-processor
A co-processor is one of the microprocessors inside the computer in addition to the CPU. The Amiga contains three such co-processors which specialise in displaying graphics, producing sound, performing input/output operations etc. They perform these functions for the CPU leaving it free for other tasks.

Cursor
A cursor is a marker which shows you the area of the screen at which you are 'located'. This is the position at which characters you type will appear or it identifies the item which will be selected if you press the

mouse button etc. The actual use depends on the type of cursor and the environment it is being used in. One of the commonest ones which you will come across is the cursor in the AmigaBASIC List window. This is a thin orange line which marks the position in the program where you are working.

Data
Data is the information which a program operates on. This data can be supplied as part of the program or it can be read from disc whilst the program is running or it can even be typed at the keyboard when required.

Decimal
Decimal is the base ten number system in which all numbers are represented by just the digits 0–9. It is the system which is in use in our everyday lives. In decimal numbers, each digit in a column represents a power of ten. For example, 123 represents 1*10*10 + 2*10 + 3.

Directory
A directory is a structure on a disc. The space available on discs can be divided into a hierarchy of directories to allow the individual files to be split up into related categories and hence located easily. Directories are also referred to as 'drawers'.

Error messages
An error message is the text printed in response to the occurrence of a problem which the Amiga does not know how to handle. These can be given for a wide variety of reasons. For example, issuing a command which BASIC does not recognise, calling a function with the wrong number of arguments and trying to save a program to a disc which is already full are all errors which will result in an error requester appearing containing an appropriate error message. To continue after an error, you must first click on the OK gadget in the requester.

Event
An event is the occurrence of a particular action. For example a mouse button being pressed, a collision occurring between two objects or a certain period of time passing are all events. All events can be trapped by a BASIC program so they may be acted upon by parts of the program called 'event handlers'. Once you have set up the event

handler, you can leave the rest of the work to the computer. AmigaBASIC checks after each statement to see if any of the events which are being trapped have occurred and, if so, automatically moves to the event handler and executes it before carrying on executing the main program.

Expression
An expression is a sequence of constants and variables together with 'operators' which act on them. For example in the statement:

```
triarea = height * width / 2
```

the variable 'triarea' is assigned the value of the expression on the right-hand side of the '=' sign. This expression contains two operators '*' and '/' which act on the variables 'height' and 'width' and the constant '2'.

File
A file is a just a sequence of bytes which can be held in memory or stored to disc. These bytes can represent any type of data. Therefore files can contain numeric data, names and addresses, screen pictures, BASIC programs etc.

Floating Point Numbers
Floating point numbers are also known as 'reals'. They are numbers which can contain a decimal fraction as well as a whole number part. For example 11.2, 0.00456 etc. AmigaBASIC supports two different types of floating point numbers, single precision and double precision. Double precision numbers cover a greater range and hold values to a greater precision.

By default, variables are treated as being single precision reals. However, they can be defined explicitly as being single precision or used for double precision reals by either using the definitions DEFSNG (single precision) or DEFDBL (double precision) or by giving the variable name a terminating '!' (single precision) or '#' (double precision). Double precision variables take up more space than single precision ones and operations on them are slower.

Function
A function is a routine which returns a result. Functions subdivide into two types: intrinsic functions which are those provided by AmigaBASIC and user defined functions which you can write yourself as part of a program. Either sort of function can return a result of any type, ie numeric or string. For example the intrinsic function ASC returns a number which is the ASCII code of a character whereas CHR$ returns a string containing the character represented by the ASCII code given as an argument.

Gadget
A gadget is a temporary box which appears when the Amiga wants you to enter information. In some cases, gadgets require you to type text into them (these are known as 'string gadgets'). In other cases all you have to do is click on them to select them.

Heap
The heap is an area of memory shared by BASIC and all the other applications running on the Amiga. It is used for holding, amongst other things, the information for controlling the screen display. Therefore, a program which creates new screens and windows requires a lot of heap space and runs the risk of generating the error message 'OUT OF HEAP SPACE'.

Hex
Hex is short for hexadecimal. It is the base sixteen number system in which all numbers are represented by the digits 0–9 and the letters A to F. Whereas in decimal, a one in a particular column represents a power of ten, in hex it represents a power of sixteen. Therefore, 123 in hex represents the decimal number 1*16*16 + 2*16 + 3.

Integer
An integer is a whole number, ie one without a fractional part. For example 27, –44, etc. AmigaBASIC supports two different types of integers: short integers which are numbers between –32768 and +32767 and long integers which lie in the range –2147483648 to 2147483647.

Variables can be defined as being integer variables by either using the definitions DEFINT (short integer) or DEFLNG (long integer) or by giving the variable name a terminating '%' (short integer) or '&' (long

integer). Long integer variables take up more space than short integer ones and operations on them are slower.

Interface
An interface is a connection which allows a peripheral device such as a printer to be connected to the Amiga. The Amiga has several different interfaces: a serial connector to allow it to be connected to other computers, a parallel connector for attaching certain makes of printer, two joystick connectors etc.

Interlacing
This is a technique used to double the vertical resolution of the screen. The display on a TV or monitor screen is refreshed 50 times a second. When the screen is interlaced, two different images are held, one containing the odd rows and the other containing the even ones and these are refreshed alternately. The disadvantage of interlacing is that the screen flickers because it has to change between two different images.

Interpreter
An interpreter is a program which reads the statements in a program, one at a time, and analyses them. If they are legal, then it carries out the operations which are necessary by using machine code routines contained within itself. A 'compiler' is a different means of achieving the same end result. It converts the entire program into machine code without executing any of it. Once it has all been converted, the resulting machine code can then be executed.

The advantage of compilers is that the work of analysing the code is done before the program starts and therefore less work has to be done during the execution phase so programs run faster. The disadvantage is that the machine code which is run has no knowledge of the BASIC which produced it. This means that you cannot step through a BASIC program which has been compiled one statement at a time to investigate the values of variables etc.

AmigaBASIC is an interpreter. In fact most home computers are supplied with BASIC interpreters because they are easier to use. Often, BASIC compilers are produced in addition by third party suppliers for serious software developers.

Intuition
Intuition is a library of routines contained in the Amiga Operating System. These routines handle the manipulation of windows, menus and the mouse etc.

Keyword
A keyword is a word which AmigaBASIC recognises and treats in a special way. Every statement or function which AmigaBASIC provides is a keyword, for example CIRCLE which is a statement used to draw a circle and ASC which is a function which returns the ASCII value of a character. The full list of keywords is given in the Reference Section of this book. When you type a program into the List Window, any keywords will be converted to upper-case when you move onto a different line to distinguish them. You are not allowed to use keywords as variable names.

Library
A library is a group of routines which are related in some way. The operating system of the Amiga is divided into libraries. Each library contains a list of addresses for each of the routines within it. These addresses can be used to access the routines from within your own programs.

List Window
The List window is one of the two windows which appear when you enter the AmigaBASIC system. It is used for entering and editing BASIC programs.

Logical Operators
Logical or 'boolean' operators can be looked at in two ways. The first is to regard them as acting on logical operands, ie those which have the value true or false. They then produce true or false as their result. The second way is to regard them as acting on sequences of bits in which case each bit of their result is obtained by acting on the corresponding bit(s) or their operand(s).

The six logical operators are 'NOT', 'EQV', 'AND', 'OR', 'XOR' and 'IMP'.

Menus

Menus are lists of items. You can see the titles of the menus available at a particular time by pressing the right-hand mouse button. To see the contents of one of these menus, point at the appropriate title with the mouse button still held down and the menu will appear beneath it. Then you can select one of the options which it contains by moving the mouse pointer down to point to it and then releasing the mouse button whilst it is highlighted.

AmigaBASIC has four menus: the Project Menu for dealing with program files, the Edit Menu for entering and editing programs, the Run Menu for controlling program execution and the Windows Menu for displaying the AmigaBASIC windows.

Multitasking

Multitasking is a word used to describe the Amiga Operating System. It means that the Amiga is able to run several tasks at once. The CPU divides its attention between each of the tasks which is running at a particular time. Therefore, the more tasks you try to run, the slower each will be. However, multitasking is a very useful feature to have since it allows you to leave one application, enter another to perform some action and then return and continue where you left off. It also means that you can be getting on with other things whilst something else is happening as a 'background task'. For example you can start printing something and, instead of having to wait for it to finish, you can be writing a BASIC program whilst the printing is going on.

Octal

Octal is the base eight number system in which all numbers are represented by the digits 0–7. Whereas in decimal a one in a particular column represents a power of ten, in octal it represents a power of eight. Therefore, 123 in octal represents the decimal number 1*8*8 + 2*8 + 3.

Opcode

Opcode stands for operation code. It is the part of a machine code instruction which instructs the CPU what operation to perform. The remainder of the instruction is the data to be used.

Operand
An operand is a piece of data which is to be operated on. For example, the addition operator requires two operands which are the two numbers to be added together.

Operating System
An Operating System is the main program supplied with a computer which is responsible for performing the fundamental actions of the machine. Without the Operating System, no software would run. For example the Amiga Operating System handles all the standard interface inputs using the mouse and menus, the displaying of screens and windows, the production of sound etc.

Output window
An output window is the window into which you type BASIC commands for loading and saving programs etc. In addition, it is where output from your programs can be displayed. You can create several output windows; however only one of these can be 'current' at a given time. Graphics and text produced by your programs will always be sent to the current output window.

When you enter the AmigaBASIC system, a default output window is created for you.

Parameter
A parameter is a piece of data which is passed between the main program and a function or subprogram which it is calling. The value which is passed is known as the 'actual parameter' and the variable which is used in the function definition to receive it is the 'formal parameter'.

Pixel
A pixel is a rectangular dot on the screen. It is the smallest element into which the display can be subdivided in a given screen mode. The pixel in the top left-hand corner of the output window is defined to be at position (0,0). Increasing the x-co-ordinate by one moves a point across by one pixel and increasing the y-co-ordinate by one moves it down by one pixel. Higher resolution modes contain more pixels; therefore each of the pixels is smaller and so the picture is clearer. The number of bits used to represent each pixel determines the number of colours which are available for the pixel to be displayed in.

Program
A program is a series of instructions which you enter into the computer and then tell the computer to carry out.

RAM
RAM stands for Random Access Memory. This is the memory inside the computer which you can both write to and read from. However, the contents of the RAM are lost when the computer is switched off. To preserve them they have to be saved to disc and read in again next time they are required.

Register
A register is a special memory location within the CPU. The Amiga's CPU contains 16 registers which you can use when writing assembly language programs. Eight of these are for holding data, the other eight for addresses. Each is 32 bits in size.

Relational Operator
A relational operator is an operator which takes two operands and returns one of two alternate results: either true or false. For example:

```
height1 > height2
```

Relational operators tend to be used mainly as tests in IF or WHILE statements. The full list is '=' (equals), '<>' (not equals), '>' (greater than), '<' (less than), '>=' (greater than or equals) and '<=' (less than or equals).

Requester
A requester is a temporary box which appears to allow the Amiga to communicate with you. These always contain gadgets asking for input. For example, when you ask to save a program, a requester may be given showing the previous name used and asking you to either supply a new name or go ahead with the one suggested.

Another example is an 'error requester'. These are used to display error messages when the Amiga has come across a problem with what you have asked it to do. These give you no option about the input you give – they contain just one gadget which you have to click on when you have read the message and want to continue.

ROM

ROM stands for Read Only Memory. The ROM is a permanent bank of memory which cannot be changed. The data within it is not lost when the computer is switched off, however it cannot be altered. Some computers keep the whole of their Operating System in ROM so that the machine is ready to be used once it has been turned on. However, the Amiga contains just a small core system in ROM – enough to enable it to start up the machine and load the main Operating System from disc plus a few extras depending on the actual model.

Sprite

Sprites are one of the types of moveable objects which the Amiga provides (the other being bobs). Each type is handled by a different co-processor, sprites being looked after by the one known as 'Denise'. Sprites are limited in certain respects. They can be a maximum of 16 pixels wide, they can contain only three colours and the number which can appear on the screen at a given time is restricted. Their advantage over bobs is that they move quickly and don't have such a tendency to flicker.

Stack

The stack is an area of memory which is used internally by BASIC to hold information about the flow of control of a program. For example, it uses it to store the position in the program at which a subroutine call is made so that it knows where to return to at the end of the subroutine. When the end of the subroutine is reached, the return address is removed from the stack. Hence, while you are within a subroutine, there will be one entry on the stack. If this subroutine calls a second subroutine, then the call address of this will be added as well and, during execution of the second subroutine, there will be two entries on the stack and so on. Therefore the amount of stack space used by a program depends not on its size but on how deeply you nest structures. Normally, the stack space required is not great.

String

A string is a sequence of characters. Constant strings should be enclosed in brackets and can be any length up to 32767 characters. For example "Hello" or "Joe Bloggs, 20 High Street, Scunchester".

Variables can be defined as being string variables by either using the definition DEFSTR or by giving the variable name a terminating '$'.

Subprogram
A subprogram is a block of code which starts with the keyword SUB followed by the name of the subprogram and ends END SUB. To execute the statements it contains, the main body of the program just needs to issue the command CALL followed by the name of the subprogram.

By default, the variables used within a subprogram are entirely independent of those of the main program. However, they can have values passed to them which allow them to vary their actions each time they are called.

Subroutine
A subroutine is a block of code which starts with a line number or label and ends with the keyword RETURN. A subroutine should be kept separate from the main body of the program. To execute the statements it contains, the main body of the program just needs to issue the command GOSUB followed by the relevant line number or label. Then all the statements in the subroutine will be executed until the RETURN is reached at which point control will pass back to the main body.

Token
A token is a code which is used to represent an AmigaBASIC keyword. Each token is just one byte long. When an AmigaBASIC program is saved to disc in the normal way, each keyword is replaced by the appropriate token in order to save room on the disc.

Variable
A variable is something which has a name and a value associated with it. The name allows it to be identified and its value to be either set or read. You can set the value as many times as you wish so that it 'varies' throughout the program – hence the name. For example, you can have a variable called 'radius' which is assigned the value for several different circles in turn and used to calculate the perimeter and area for each. Using variable names instead of explicit values in expressions makes them easier to understand.

Word

A word is 16 bits or two bytes. It can be used to hold numbers between 0 and 65535 (0000000000000000 and 1111111111111111 in binary). However, when used to represent short integers, the usual convention is to interpret the top bit as the sign bit in which case a word represents numbers between -32768 and +32767. Long integers require two words to represent them.

Other Dabs Press Books

Dabs Press publishes a wide range of books on computer topics. There follows a list of some of our recent and forthcoming titles.

If you are interested in any of these books, details of how to obtain them are given at the end of the list.

Commodore Amiga

AmigaDOS: A Dabhand Guide by Mark Burgess

ISBN 1-870336-47-X. Price £14.95. Available NOW.

This is a comprehensive guide to the Commodore Amiga, and its disc operating system, covering releases 1.2 and 1.3 of AmigaDOS/ Workbench. It provides a unique perspective on this powerful system in a way which will be welcomed by the beginner and experienced user alike.

Rather than simply reiterating the Amiga manual, this book takes a genuinely different approach to understanding and using the Amiga and contains a wealth of practical hand-on advice and hints and tips.

Among the many features in this book are:

- Full coverage of AmigaDOS functions
- Filing with and without the WorkBench
- The Amiga's hierarchical filing system
- Pathnames and device names
- The Amiga's multitasking capabilities
- The AmigaDOS screen editor
- AmigaDOS commands
- Batch processing
- Amiga Error code descriptions
- Use of the RAM discs
- Using AmigaDOS with C

C: A Dabhand Guide by Mark Burgess

ISBN 1-870336-16-X. Price £14.95. Discs £7.95-£9.95 inc. VAT. Available NOW.

This is the most comprehensive introductory guide to C yet written, giving clear, comprehensive explanations of this important programming language.

The book is packed with example programs, making use of all C's facilities. Unique diagrams and illustrations help you visualise programs and to think in C.

Assuming only a rudimentary knowledge of computing in a language such as C or Pascal, you are provided with a grounding in how to build up programs in a clear and efficient way.

The differences between various compilers are acknowledged and sections on the popular compilers for the Amstrad/IBM PC, Acorn machines including BBC and Archimedes, Atari ST and Commodore Amiga are included, with notes concerning the ANSI and Kernighan and Ritchie standard.

Features of the book include:

- Compatible with all popular ANSI and K&R compilers
- Sections for PCs, Atari, Amiga and Acorn
- Diagrams to help you think in C
- Arrays and string handling
- Data structures
- Mathematical programming
- Recursion
- Discs available for many machines

Mark Burgess writes computer programs in many languages of which C is his favourite. He is an honours graduate in Theoretical Physics.

"I wish this book had been available when I was learning C" Personal Computer World. *"...will give even relatively inexperienced programmers a clear understanding of programming in C."* Elektor Magazine (December 1988).

Acorn Archimedes

Archimedes First Steps: A Dabhand Guide by Anne Rooney

ISBN 1-870336-73-9. Price £9.95. Available NOW.

This book is the ideal starting point for first-time users of the Archimedes, taking you through the first few days and months of owning and using the machine.

There is an abundance of software provided with the Archimedes, and Anne goes through the programs, telling you how to get them started and how to get the most out of them.

Many hints and shortcuts for using the RISC OS Desktop are also discussed, as are many third-party commercial software packages in such fields as art, music and so on.

Budget DTP for the Acorn Archimedes & A3000: A Dabhand Guide by Roger Amos

ISBN 1-870336-11-1. Price £12.95. Available Now.

Every Archimedes and BBC A3000 owner receives copies of the !Draw, !Paint and !Edit software, along with the RISC OS operating software. This book shows how these applications can be used to produce high-quality documents without the need for an expensive desktop publishing package.

The book includes detailed descriptions of the various applications, and helpful tips are given on how to get special effects such as drop shadows and pie charts. There are also sections on fonts, clip art and page layout as well as printers and reproduction of your finished work.

Roger Amos is a technical journalist and public relations consultant, who has used Acorn computers since the appearance of the BBC Model B. He has a long-standing professional interest in typography and has worked on Beebug's DTP package Ovation and their Outline font creation programme.

Archimedes Operating System: A Dabhand Guide
by Alex & Nic Van Someren

ISBN 1-870336-48-8. Price £14.95. Programs disc £9.95 inc.VAT. Available NOW.

For Archimedes users who take their computing seriously, this guide to the Operating System gives you a real insight into the micro's inner workings. This book is applicable to any model of Archimedes.

The Relocatable Module system is one of the many areas covered. Its format is explained and the information necessary for you to write your own modules and applications is provided. This tutorial approach is a common theme running throughout the book.

The sound system is explained and the text includes much information never before published. The discerning user will revel in the wealth of information covering many aspects of RISC OS such as:

- The ARM instruction set
- Writing relocatable modules
- VIDC, MEMC and IOC
- Sound
- The voice generator
- SWIs
- Vectors and Events
- Command Line Interpreter
- The FileSwitch Module
- Floating Point Model

Throughout the book, programs are used to provide practical examples to use side-by-side with the text, which go to make this publication the ideal table-side companion for all Archimedes users.

A programs disc is also available containing all the listings from the book, and some extra useful programs as well.

"Here is an essential book for Archimedes programmers" Micronet 800 (April 1989). *"A jolly good read. Lots of really useful information presented in an accessible and readable manner...this is a clearly written, well presented book. It is up to the usual high standards we have come to expect*

from Dabs Press, and I wholeheartedly recommend it to all who want to know more about their machine's operating system." Archive magazine March 1989.

Archimedes Assembly Language: A Dabhand Guide by Mike Ginns

ISBN 1-870336-20-8. Price £14.95. Programs Disc £9.95. Available NOW.

Learn how to get the most from the remarkable Archimedes micro by programming directly in the machine's own language, ARM machine code. This is the only book that covers all aspects of machine code/assembler programming specifically for the entire Archimedes range.

For those new to assembler programming, this book contains sections which take you step-by-step through new and exciting areas of Archimedes programming, including many examples using the features of the RISC OS Operating System, including the co-operative multitasking environment.

- Practical tutorial approach with example programs
- Descriptions of all the processor instructions
- Using the Operating System, WIMPs and Vectors
- Co-operative multitasking explained
- Assembler equivalents of BASIC commands
- Sound and graphics in machine code

"The contents make the book a welcome addition to the manual provided with the computer, and will, no doubt, be an invaluable source of information for many owners of an Archimedes" Everyday Electronics (December 1988)

BASIC V: A Dabhand MiniGuide by Mike Williams

ISBN 1-870336-75-5. Price £9.95. Available NOW.

This is a practical guide to programming in BASIC V on the Acorn Archimedes. Assuming a familiarity with the BBC BASIC language in general, it describes the many new commands offered by BASIC V, already acclaimed as one of the best and most structured versions of the language on any micro.

The book is illustrated with a wealth of easy-to-follow examples.

An essential aid for all Archimedes users, the book will also appeal to existing BBC BASIC users who wish to be conversant with the new features of BASIC V. Major topics covered include:

- Using the colour palette
- Use of mouse and pointer
- Operators & string handling
- Control structures
- Functions and procedures
- Extended graphics commands
- WHILE, IF and CASE
- Local error handling
- The Assembler
- Matrix operations
- Sound
- Hints and tips

Mike Williams has been working with computers for over twenty years. For the past five, he has been editor of Beebug and RISC User magazines, the latter being the largest circulation magazine devoted to the Archimedes.

BBC Micro & Master

Master 512: A Dabhand Guide by Chris Snee

ISBN 1-870336-14-3. Price £9.95. Programs Disc £7.95 inc.VAT. Available NOW.

This is a comprehensive reference guide for all users of the Master 512, Acorn's PC-compatible add-on for the Master 128 and BBC Micros, and the companion Volume to this book.

Master 512: A Dabhand Technical Guide by Robin Burton

ISBN 1-870336-80-1. Price £14.95. Program Disc £7.95 inc. VAT. Available NOW

This second volume on the Acorn Master 512 covers the more technical issues associated with the system and provides useful information on technical utilities provided with the system, such as EDBIN, the binary file editor.

Master Operating System: A Dabhand Guide by David Atherton

ISBN 1-870336-01-1. 272pp. Price £12.95. Program Disc £7.95 inc. VAT. Available NOW.

Now in its second edition, this is the definitive reference work for programmers of the BBC Model B+, Master 128, and Master Compact

computers. It also contains much material of interest to BBC Model B and Electron users. The book covers all the features of the Acorn machine operating system (MOS).

Mastering Interpreters and Compilers by Bruce Smith

ISBN 0-563-21283-7. 314pp. Price £14.95 incl. programs disc (incl.VAT). Available NOW.

This clear and comprehensive introduction to the often misunderstood topic of computer language interpreters and compilers emphasise the practical side of the art.

BBC Micro Assembler Bundle by Bruce Smith

ISBN 1-870336-08-9. Price £4.95 (inc.VAT). Available NOW.

This is a five part package of materials for anyone starting out learning assembly language/machine code programming on the BBC Micro/Master Series.

Mini Office II: A Dabhand Guide by Bruce Smith and Robin Burton

ISBN 1-870336-55-0. Price £9.95. Program Disc £7.95 inc.VAT. Available NOW.

Bruce Smith and Robin Burton have joined forces to write this official tutorial and reference guide to the award-winning and revolutionary Mini Office II software. This book covers the BBC Micro and Master versions of the program.

VIEW: A Dabhand Guide by Bruce Smith

ISBN 1-870336-00-3. Price £12.95. Program Disc £7.95 inc.VAT. Available NOW.

Now in its second edition, this is the most comprehensive tutorial and reference guide ever written about the Acornsoft VIEW wordprocessor, for the BBC Micro, and issued as standard (but without a manual!) on the BBC Master 128 and Compact computers.

ViewSheet and ViewStore: A Dabhand Guide by Graham Bell

ISBN 1-870336-04-6. Price £12.95. Program Disc £7.95 inc. VAT. Available NOW.

This is a complete tutorial and reference guide for the ViewSheet spreadsheet and ViewStore database manager for the BBC Micro model B/B+, Master 128 and Compact computers. Whether you wish to check your bank statement or run a million-pound business, this book is for you.

Z88

Z88 A Dabhand Guide by Trinity Concepts

ISBN 1-870336-60-7. Price £14.95. Available NOW

This is the most comprehensive guide for all users of the Z88 portable computer and is indispensable for anyone wanting to get the most out of their machine.

All of the standard built-in application programs, including (but by no means limited to) PipeDream, are covered and clearly explained using easy-to-follow examples, and many hints and tips are included *en route.*

Z88 PipeDream: A Dabhand Guide by John Allen

ISBN 1-870336-61-5. Price £14.95. Available NOW

In this detailed and authoritative book, John Allen explains how to get the most out of PipeDream, the standard business software supplied with the Cambridge Z88 portable computer.

Psion Organiser

Psion Organiser LZ: A Dabhand Guide by Ian Sinclair

ISBN 1-870336-92-5. Price £14.95. Available NOW

In this exciting book, Ian Sinclair, the UK's premier computer author delves into the new LZ Organiser from Psion, explaining how to use the various utilities and the built-in programming language.

IBM PC Compatibles

BASIC on the PC: A Dabhand Guide by Geoff Cox

Price £14.95. Available early 1992.

In this book, Geoff Cox provides a comprehensive tutorial and reference to the programming language provided free with most IBM-compatible machines. As well as a friendly and helpful tutorial in BASIC programming, the book contains a complete command reference, detailing every command in GW-BASIC with examples of its use.

The book is also suitable for programming with Microsoft QuickBASIC and Borland TurboBASIC.

Ability and Ability Plus: A Dabhand Guide by Geoff Cox

ISBN 1-870336-51-8. Price £14.95. Available early 1992.

In this book, Geoff Cox provides a no-nonsense comprehensive tutorial and reference to this popular integrated package for IBM compatible computers including the Amstrad range.

All aspects of all the modules are covered and, by the use of examples, you are shown how to perform a range of business tasks and how to use the programs in conjunction with each other, including transferring of data.

Amstrad PCW

PCW9512: A Dabhand Guide by F. John Atherton

ISBN 1-870336-50-X. Price £14.95. Available Spring 1992.

The Amstrad PCW9512 personal computer word processor and its accompanying software, the LocoScript 2 system, has revolutionised low-cost wordprocessing and introduced a whole generation of people to computer-based word processing for the first time.

In this easy-to-follow guide, John explains how to use the program starting from first principles, with no prior knowledge assumed, either of the Amstrad PCW system, the LocoScript program or even computers in general.

You are shown in practical detail how to set the system up to your own preferences and how to produce neatly laid out letters, reports, essays and so on.

Difficult subjects are not avoided; instead they are introduced in a painless and straightforward way. After you have read this book, you will, without knowing it, become a perceptive and sagacious word processor user!

F. John Atherton has used an Amstrad PCW machine for many years, and has trained dozens of beginners on the machine. He has used the most common questions and problems as the basis for many of the topics in this book.

General

Software

Dabs Press publish a range of software for the Acorn Archimedes and BBC Micro, including products as diverse as language compilers for BASIC and Pascal, and computer games. For a free catalogue of software, please contact us.

Obtaining Dabs Press Books and Software

You can obtain Dabs Press books and software from any good bookshop or computer dealer, or in case of difficulty directly from us.

Orders can be sent by post and payment can be made by cheque (drawn on a UK bank), postal order, credit card (quote number and expiry date), or official order (education/public sector/PLCs only).

Telephone or fax orders can be made with a credit card — this is the simplest and most popular method.

Our address, telephone number and fax number are on page 2 of this book.

Index

Aug 92

A Dabhand Guide

AmigaBASIC: A Dabhand Guide provides a fully structured tutorial to using AmigaBASIC on the whole range of Commodore Amiga computers.

Practical application is one of the many themes running through the pages and as such the many varied programs contained in its pages are both useful, and informative in programming technique. You are assumed to have a grounding of the way in which your Amiga works but no prior knowledge of BASIC itself is necessary. A graphical theme is applied to the many examples in the book so that the techniques described are visually reinforced.

The many features of this book include:

- Writing and editing a program
- Handling and understanding errors
- Communicating with the user
- Text handling
- Graphics and the palette
- Animation, sprites and collisions
- Sound, Voices and speech
- Structured programming
- File handling
- Writing large programs
- Debugging programs
- Memory and resource management

AmigaBASIC: A Dabhand Guide is one of the most comprehensive and informative books on this topic, and an indispensable reference to any AmigaBASIC programmer.

Paul Fellows is a professional computer programmer and writer with many years of experience in the field.

£15.95

ISBN 1 870336 87 9

9 781870 336871