USING LOCOMOTIVE BASIC 2
- on the Amstrad PC1512

Robert Ransom

Sigma Press, Wilmslow

This book is dedicated to my grandfather, Sidney Mellett, who at the age of 89 has shown that it is never to late to learn about computers.

Preface

This book is a step-by-step guide showing how to use the Locomotive BASIC 2 language on the Amstrad PC1512 and other IBM compatible computers. It is not intended to replace the excellent manual supplied by Locomotive Software for the BASIC 2 system, but presents a rather less terse introduction to the BASIC 2 language. The author's intention is to present a clear and readable account of the essential elements of BASIC 2, accenting the differences between BASIC 2 and the more traditional implementations of the BASIC programming language that the reader may be used to. Knowledge of simple BASIC programming will be useful knowledge for the intending reader of this book. The reader seeking an introduction to general aspects of the BASIC language will find the *Locomotive BASIC 2 User Guide* very helpful.

BASIC 2 is a GEM-based version of BASIC. GEM, or Graphics Environment Manager, is an adjunct to the Operating System of the computer on which it is running. It gives the computer the ability to use the WIMP (Windows, Icons, Mouse and Pointer) facilities. A BASIC running on a computer with GEM would be rather inadequate if it did not allow the programmer to use the WIMP facilities from within BASIC. To the author's knowledge, BASIC 2 is the first version of BASIC that allows you to use WIMPs on an IBM compatible computer.

This book is laid out in three main sections, each consisting of several chapters. The first section introduces GEM and BASIC 2, and describes how to get started with programming in BASIC 2. The second section takes a more detailed look at some of the facilities available within BASIC 2. This section begins by comparing BASIC 2 with more traditional versions of BASIC, before concentrating on facilities like the windows, graphics commands and use of the mouse. Chapters on file handling and 'advanced' topics like error trapping are also included. Finally, a reference section offers a handy reference guide to the commands and structures of BASIC 2. The present book also introduces the relevent parts of the GEM interface as well as BASIC 2. The reader will therefore not have to constantly refer back to the GEM user guide.

The book is intended for all users of BASIC 2, and not just for the owners of the Amstrad PC1512 who have BASIC 2 and GEM supplied with their machines. It is hoped that BASIC 2 will become the standard GEM-based BASIC language on

IBM PC compatible computers. Certainly there is no present competitor to this aim. I hope that you find BASIC 2 as useful and flexible as I have done, and that this book provides the springboard for many inspired and interesting programming projects.

A number of people have made this book possible, not least Graham Beech of Sigma Press, who must be the most amiable publisher around. The staff at Locomotive Software have also been very helpful, notably Howard Fisher and Chris Hall who have read and commented on the manuscript. However, any remaining errors and omissions are due to me, and not Locomotive Software. This book has been prepared using an Amstrad PC1512 with many programs also tested on an IBM PC (thanks to Geoff Einon, Mike Stewart, and Richard Austin from the Open University, and Dave Huckle of Lansdowne Educational Supplies for hardware and software). Dave Stanford of the Open University kindly transferred the program examples from P.C to Macintosh. The book was in fact typeset directly using a Macintosh Plus computer with the MacWrite word processor and MacDraw/MacDraft graphics packages. Camera ready copy was obtained from an Apple LaserWriter.

The final thanks must go to my wife Anne and children James and Paul, who make it all worthwhile, and without whose encouragement this book would not have been possible.

Robert Ransom

November 1986

CONTENTS

LIST OF ILLUSTRATIONS

Chapter One
Introduction

1.1 The visual interface

Until the advent of the Apple Macintosh computer, heralding the appearance of the WIMP (Windows, Icons, Mouse and Pointer) interface on relatively low cost computers, the BASIC programmer was limited to use of the keyboard for the input of information. The characters inputted via the keyboard appeared on the monitor screen in a single 'window' that was literally the full monitor screen in size. This mode of interaction between computer and user is still commonly found on many computers, the main advantages being simplicity for the system designer and economy of computer memory use. For the user there are several drawbacks, most notably the mixing of various types of process on the screen at the same time. Writing and running a program involves a complex interplay between typing the program into the computer, editing errors, running the program, and observing the output.

```
*  LIST
* 10 FOR X=1 TO 4
* 20 PRINT X
* 30 NEXT X
* RUN
1
2
3
4
```

Figure 1.1 Commands, program listing and output all on the same screen area

Using a single screen area in which to do all these things is clumsy and inconvenient. A major inconvenience is that running a program with a single screen area usually means losing the program listing from view, if any output is sent to the screen.

This 'traditional' way of programming is also usually associated with a separation in the way text and graphics are treated on the screen. The earliest microcomputers allowed text-only output, and the screen was divided into many

640 Pixels

200 Pixels
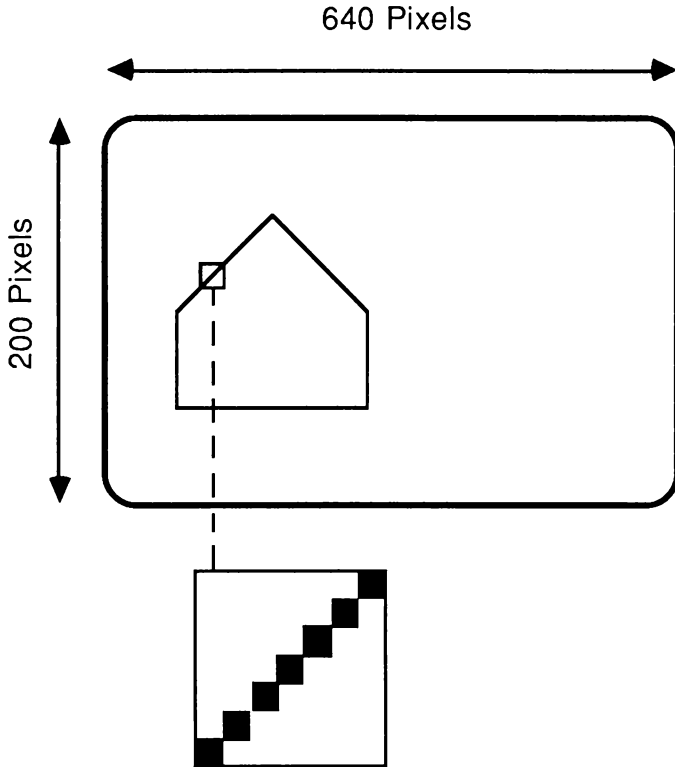
Figure 1.2 Relationship between bitmapped screen and an object drawn on the screen The lit pixels are here shown as black squares in the enlarged portion of the picture.

'textcells', characteristically 80 characters across by 25 characters down the screen. Later on, graphics facilities were developed, these using the concept of a

'bit-mapped' computer screen. In the simplest terms this means that the screen is divided into a large number of dots or 'pixels', typically 640 across by 200 or 400 down the screen. Each dot can be 'on' (lit) or 'off' (unlit). The computer draws graphics shapes made up of lines and points by lighting sequences of pixels on the screen.

Bit-mapped graphics introduced flexibility in the sense that graphics and text could be displayed on the same microcomputer, but there were difficulties in mixing the two at the same time, text characters using the textcell format rather than being bitmapped. The appearance of the Apple Macintosh computer offered a solution to the problem. Text and graphics were both processed on the same bitmapped screen. This novelty did not just allow simpler mixing of text and graphics. It also paved the way for the use of multiple fonts and text point sizes on the same screen. If text is manipulated in the same way as pictures, it seems logical that it can be treated with the same degree of flexibility.

The Macintosh was designed from the 'ground up' as an integrated software and hardware system. All Macintosh software uses the same visual interface via calls to the Macintosh Toolbox: a series of routines in ROM that control the behaviour of the interface. The IBM PC was not designed with a visual interface in mind, and therefore software to handle the WIMP facilities had to be developed independently. Several such interfaces are now available. Most prominent of these are Microsoft Windows and Digital Research GEM. BASIC 2 is at present based on GEM, although a Microsoft Windows version may be available in the near future. We will concentrate on the GEM interface here.

1.2 GEM

GEM can be studied at a number of levels, depending on the requirements of the computer user. At the most sophisticated level, it is necessary to see how GEM interacts with the operating system of the computer on which it is used. At this level of understanding it is crucial to know the complex details of the interplay between GEM and the host computer. At an intermediate level, the computer user may wish to use a number of different applications within the GEM shell. The user must find out whether or not the programs he or she wishes to use can be run under GEM, and if so, how they need to be tailored to allow smooth running of the application.

Fortunately, the BASIC 2 user only needs to scratch the surface of GEM understanding. Once the GEM Desktop (see Chapter 2) is loaded, the user is immediately able to use the comforting visual interface of windows and icons, and much of the bugbear of using a computer can be forgotten.

1.3 BASIC 2

BASIC 2 has its origins in the prototype BASIC language developed at Dartmouth College in the USA in the 1970s. The most obvious advantage of BASIC is that it is a 'high level language' closer to English than the machine code understood by microcomputer processors. Many other high level languages exist, like Fortran, C and Pascal. The most significant advantage of BASIC to the computer novice is that it is easy to learn and teach. Although BASIC has received a lot of 'bad press' in computer magazines in recent years, modern versions of the language are much more acceptable than the first BASICs, which were designed to run on computers with very small memories. These early BASICs offered only the most rudimentary structures (think of computer language structures as similar to structures in the grammar of, say, the English language). Advanced BASICs lack some of the sophistication of, say C or Pascal, but allow the computer user enough flexibility for most programming tasks.

The real power of BASIC 2 over earlier BASICs is the use of the GEM visual interface. The use of GEM allows easy access to commands and facilities through a system of menus, and more flexible control over the display is made possible by the use of multiple windows on the screen. BASIC 2 also features a comprehensive range of graphics facilities and sophisticated handling of disk files. In the next chapter we will see how to load up BASIC 2 and will start to investigate the range of features that it offers.

1.4 How to use this book

The novice reader should work carefully through the rest of the book from Chapter 2 onwards, whilst the reader with a rudimentary knowledge of GEM or BASIC will be able to quickly skim through Chapters 2 and 3. If you have used BASIC on a Macintosh you should have little difficulty mastering BASIC 2, and could start reading at Chapter 4. Chapters 4 - 9 present a more detailed look at the facilities within BASIC 2, and should be useful for any computer user

intending to use BASIC 2.

The appendices at the end of the book provide essential information about BASIC 2 command and function syntax, screen maps, error messages and character codes.

As you read the following chapters, you will find that BASIC 2 keywords are printed in bold, like this - **IF THEN ELSE**. Other items of program text (variables, comments, constants and so on) are shown in the same typeface, but not emboldened. The names of the four BASIC 2 windows (**Dialogue**, **Edit**, **Results-1** and **Results-2**) are shown in bold type.

One of the most puzzling aspects of learning a new computer language is the mystery of the language syntax, the way in which the language is formally described. In the present book, discussion of syntax has been kept to the minimum, as real examples are so much easier to understand. When command and function syntax is given, the general rule is that items in square [ ] brackets are *optional*, and the square brackets are *not* used in the program. Items using round ( ) brackets *are* used in the programs directly.

Chapter Two

Welcome to BASIC 2

2.1 Getting BASIC 2 up and running

Operating systems and GEM

Starting BASIC 2 is a simple procedure, and the first step is to enter the GEM environment on your computer. As we saw in the last chapter, GEM is not an operating system, so the first step is to 'boot up' the computer with an appropriate operating system. For all IBM compatible computers, including the Amstrad PC1512, you can choose to use the MS DOS operating system, and this will load in automatically if you switch on your computer with the MS DOS disk in Drive A.

An alternative operating system is Digital Research's DOS Plus, also supplied with the Amstrad PC1512. **If you have an Amstrad, you are advised to use DOS Plus to boot up your computer as GEM will load automatically from the version of DOS Plus on your GEM startup disk (Disk 2).** You will be prompted to insert the GEM desktop disk (Disk 3) during the GEM startup procedure. If you have a PC1512 with a hard disk you should follow the instructions in your User Manual for mounting GEM on the hard disk.

If you do not have an Amstrad PC1512 you should follow the instructions for loading GEM on the version of GEM supplied for your computer. The installation procedure varies from machine to machine.

Once the GEM Desktop is displayed, you should make sure that the root directory of Drive A (A:\) is visible. You can do this by clicking the mouse button in the small box at the top right of the upper window on the desktop until the correct directory is visible in the title bar across the top of the window (see Figure 2.1). Alternatively, if the icon of disk A is visible in the window, you can open the root directory by double clicking (ie clicking twice quickly in succession) on the disk icon.

The GEM Desktop disk supplied with the PC1512 contains the BASIC 2 application program, and some sample BASIC 2 programs. The folder containing the BASIC 2 application should be visible as in Figure 2.1 below.

Versions of BASIC 2 for other computers may come on a disk without the GEM Desktop. If BASIC 2 is not on your GEM Desktop disk, it will not be visible in the Desktop root directory window, and you should put the BASIC 2 disk in the second disk drive (B). If you have a single drive machine you will need to transfer BASIC 2 to the GEM Desktop disk to avoid unnecessary disk swapping. The BASIC 2 application is under 90K in size, and so it should be possible to move it without too much trouble unless the target disk is full up. Your GEM manual will explain how to copy programs between disks.

Opening BASIC 2

Use the mouse to move the pointer to the folder marked BASIC 2, and double click the lefthand mouse button.
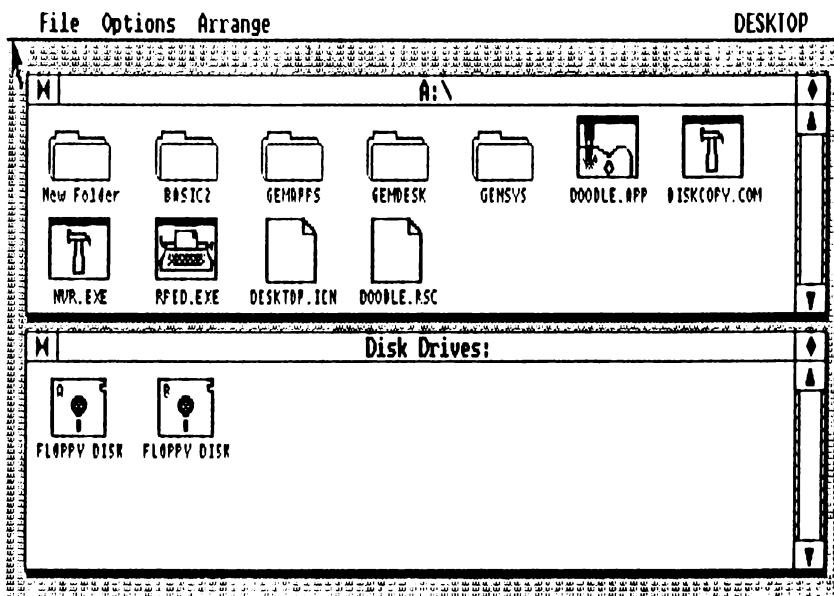
Figure 2.1 The GEM Desktop showing contents of the Amstrad PC1512 GEM Desktop disk in drive A

A:\BASIC2\

New Folder   EXAMPLES   PROGRAMS   BASIC2.APP   DEMO.BAS   TEST.DAT   ROM1.GEM

ROM1.IMG   BASIC2.RSC

Disk Drives:

FLOPPY DISK   FLOPPY DISK

File   Program   Edit   Fonts   Colours   Patterns   Lines   Windows   BASIC2

Dialogue   Results-1

print "100"   100
Ready

Edit
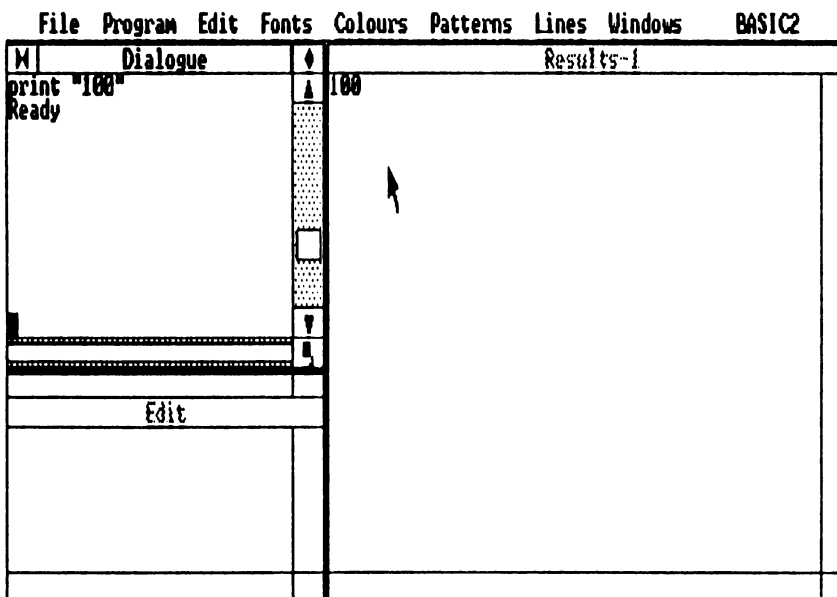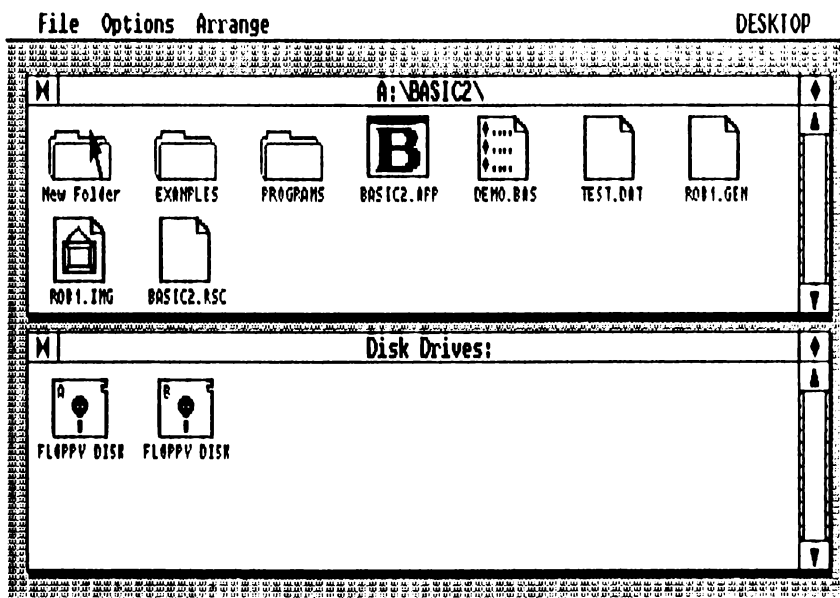
Figures 2.2-2.3 Contents of the BASIC 2 folder (top); the BASIC 2 screen showing some text output in the **Results-1** window (bottom)

You should now see the contents of the BASIC 2 folder in a separate window on

21

the screen. If the BASIC 2 window did not open, try double clicking the mouse button again. The window should look like the representation in Figure 2.2.

The next step is to enter the BASIC 2 application itself. To do this you need to move the pointer to the BASIC 2 application icon, and to double click the lefthand mouse button over the icon. If you are unfamiliar with the use of the mouse, you will already be learning a fundamental procedure in its use: double clicking is equivalent to opening something on the screen, be it a folder, an application, or a program.

If you double clicked the BASIC 2 icon at the right speed, you will now see that the pointer is replaced on the screen by an egg-timer. This merely shows you that the computer is busy doing something: in this case, reading the BASIC 2 software into memory. When this process is complete, you will see a screen like the one on the bottom of the next page. Congratulations. You are ready to use BASIC 2.

2.2 Finding your way around BASIC 2

If you are used to using more primitive versions of BASIC, say on a Sinclair, Commodore or BBC machine, you will find BASIC 2 a little strange at first. The main differences are the provision of the mouse, and multiple screen windows.   Within BASIC 2, the mouse is used mainly for control operations, the main ones being:

Selection of menu options
Movement and scrolling of screen windows
Cursor movement
Editing functions

Most of the text and data entry will take place via the keyboard. You will also see that you can type in many of the menu-driven commands also accessible with the mouse. Some commands have keyboard 'shortcuts' that can be quicker than selecting menu options, but of course you have to take the trouble to learn them!

Using the menus

The menus make the job of learning commands very easy. On a traditional computer you will have to work out the correct sequence of commands to type in to, say, save a program on disk or to perform editing functions. The menus in

BASIC 2 do away with this drudgery.

Move the pointer across the line at the top of the screen. This line is called the **menu bar**. As the pointer crosses each title, a menu is 'pulled down' like this.

```
 Colours  Patterns  Lines  ▓Windows▓      BASIC2
                        Res  Show Results-1
 ┌─────────────────────      Hide Results-1  ──────┐
 │ 100                                             │
 │                          Show Results-2         │
 │                          Hide Results-2         │
 │                    ▶                            │
 │                          Show Edit              │
 │                          Hide Edit              │
 │                                                 │
 │                          Show Dialogue          │
 │                          Hide Dialogue          │
 │                                                 │
 │                                                 │
 │                                                 │
 │                                                 │
 │                                                 │
 │                                                 │
 └─────────────────────────────────────────────────┘
```
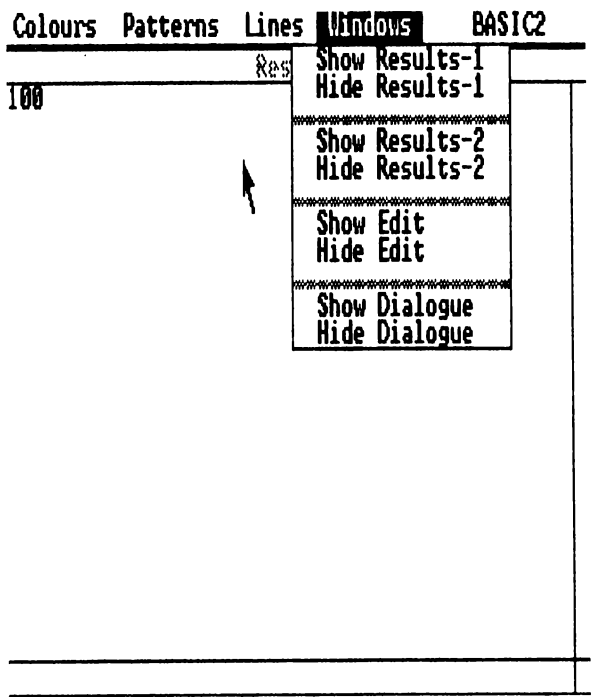
Figure 2.4 View of the BASIC 2 screen showing the Windows menu 'pulled down'

Try pulling down the other menus by pointing at their keywords on the menu bar. If you run the pointer down a menu when it is pulled down you will in turn select the options available on that menu. Don't press the lefthand mouse button yet! This action will activate the option. Note that some options (and sometimes menu titles) appear in lighter text. This indicates that the facility is not available to you at that time. Note also that once you have pulled down a menu, the menu will stay down until you either click on a menu item, activate another menu, or click outside the menu bar.

The screen windows

When you first enter BASIC 2 (Figure 2.3) you see the opening screen with three windows. These windows perform different functions. The top left window is the **Dialogue** window. This is the window in which commands may be directly entered. The **Edit** window is the part of the screen in which programs are listed and edited. The **Results-1** window is where program output will be observed. (There is actually a fourth window called **Results-2** that can also be used for text output: as this window is initially hidden from view we will deal with it later.) You will learn how to use the windows for program construction in the next chapter, but first we will see how the windows can be manipulated.

Move the pointer to the **title bar** along the top of the **Edit** window. Press the left mouse button down, and keeping your finger on the button move the mouse. You will see the **Edit** window moving. Move it to the centre of the screen and release the button.

Now move the pointer to the small box at the bottom right hand corner of the **Edit** window. This box is called the **size box**. Press the left hand mouse button down while it is inside the size box. Keeping the button pressed down, move the box diagonally towards the bottom right of the screen. You will see the **Edit** window enlarging. Release the mouse button to reform the window to the size you require.

The small box at the top left of the window has a small 'bow tie' in it. This is the **go-away box**. Try clicking inside it: the window will disappear! If you get lost after doing this, you can reselect the required window from the **windows** menu. At the right hand top corner of the current window is a diamond shape in a small square called the **full size box**. If you click inside this box the window will be expanded to fill the whole screen.

Along the bottom and right hand edges of the active window are **scroll bars**. The right hand one contains a square indicating the vertical position of the displayed part of the window on the whole virtual screen (this term will be explained fully in Chapter 6). You can scroll the square up and down within the scroll bar by putting the mouse pointer inside it, holding the left button down, and sliding in the required direction. The squares with triangles at the top and bottom of the scroll bar are used to scroll the window up and down fractionally.

The flexibility of window function, sizing and position makes the programming process easier and quicker in BASIC 2. You will appreciate the truth of this statement more and more as you gain proficiency in using the language.

Text and graphics

You may remember from the last chapter that we described how computers using visual interfaces have developed systems that treat text and graphics on screen in the same way, adding to the flexibility of the display. Four of the menus offered in BASIC 2 demonstrate the advantages of such a system. These are the **Fonts**, **Colours**, **Patterns**, and **Lines** menus. If you select the Fonts menu, you will see the range of text fonts and styles available to you. The range will vary depending on what fonts are installed on your disk (see Chapter 7).

Colours can be used for both drawing text and lines and for filling in shapes on the screen, while patterns are used for shape filling. A range of line thicknesses and styles is offered by the lines menu.

You can immediately see the effect of choosing the various options in these menus. Experiment by changing the text colour, size, and font - try printing a few words in the **Results-1** window. Enter the following command into the **Dialogue** window.

PRINT "Sample text variations"

Notice as you select menu options that there is no change in the **Dialogue** window, but the chosen menu options are adopted by the output.

The **Patterns** and **Lines** menus have no effect on straight text, and are used with graphics commands.

Fonts

▸ 1 System	
2 Swiss	
3 Dutch	
4 Loadable	

7 Point	18 Point
8 **Point**	20 **Point**
▸ 10 **Point**	24 Point
12 Point	28 Point
14 Point	36 Point
16 **Point**	72 Point

Thickened
Lightened
Skewed
Underlined

Colours

Text
0	8
▸ 1	9
2	10
3	11
4	12
5	13
6	14
7	15

Graphics
0	8
▸ 1	9
2	10
3	11
4	12
5	13
6	14
7	15
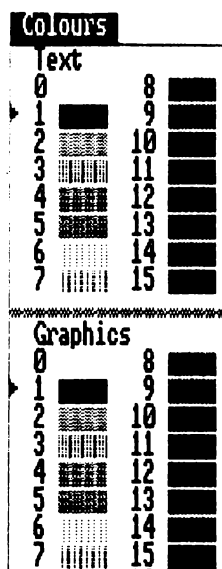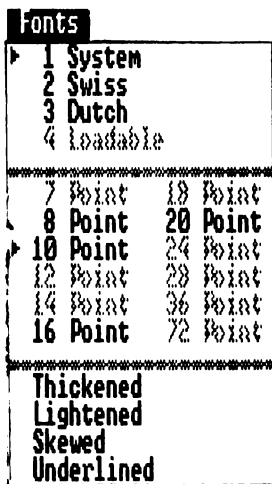
Figure 2.5-2.6 The Fonts and Colours menus in BASIC 2. Note that some fonts and colours may not be available on particular computer systems.

Here are two commands in BASIC 2 for generating a straight line and a square

respectively.

`**LINE** 100;100,200;200`

`**BOX** 100;100,100,100 **FILL**`

Try changing some of the **Patterns** and **Lines** menu options, and then enter one or other of the above commands into the **Dialogue** window. You should be able to vary the thickness of the lines drawn, and also vary the pattern with which the square is filled.

Besides being accessible through the menus, the text and graphic options available throught the menus may also be set via commands in BASIC 2. The relevant commands are described in Chapters 6 and 7 and are listed in Appendix 1.

'Cheap and dirty' approach

This short section is really for programmers with a little experience of a BASIC dialect, and who want to jump straight into BASIC 2 without working through the next chapter. These people will want to use the edit and output windows as quickly and easily as possible. You can use BASIC 2 direct commands to switch directly between the edit and output windows using function keys.

From the BASIC 2 entry screen, activate the **Edit** window by clicking inside it, and then click in the full size box for this window. The **Edit** window will then fill the whole screen, and you can use it to write your test programs. Try toggling the **f10** key. This selects and deselects the **Dialogue** window. From the **Edit** window you can run your program using the **f9** key. When the program stops, control will be transferred to the **Dialogue** window, and again you can get back to your full-screen editor using the **f10** key. If the program gets stuck for some reason (it happens to us all from time to time), use **Ctrl-C** to abort back to the **Dialogue** window.
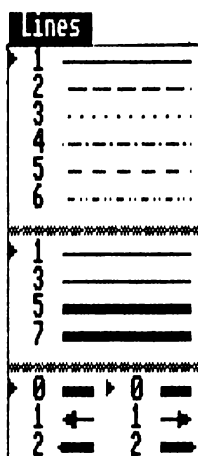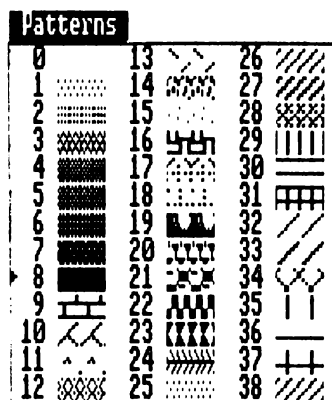
Figure 2.7-2.8 The Patterns and Lines menus available in BASIC 2.

Chapter Three

The BASIC 2 Environment

3.1 About this chapter

If you have not used BASIC 2 before, the combination of windows and mouse presents a rather foreign environment to the BASIC programmer used to the more traditional interface used on most home and business computers. In fact, the beginner with no programming experience may have an advantage in learning BASIC 2, as he or she will have no preconceptions about how the language should look on screen. Whatever your background, this chapter is designed to get you using BASIC 2 with the least trouble.

You will probably find the best way to use the chapter is to work through it page by page, with the book open beside your computer. By the time you have completed this chapter you will be able to use the most important components of the BASIC 2 visual interface: you will not be an expert programmer, but you will be familiar with the environment in which you will be doing your programming.

3.2 The Dialogue window

Load in BASIC 2 using the procedure described at the beginning of Chapter 2. If you have done any programming before, the window at the top left of the screen will look familiar, as it contains a copyright message about the version of BASIC in use. It also has a couple of other items of comfort to the programmer: a rectangular **cursor** and the message 'Ready'. The cursor is a device that indicates where the input entered from the keyboard will appear. 'Ready' indicates that the BASIC software is loaded and is 'ready for action'. Don't confuse the cursor with the **pointer**, the arrow that moves around the screen as the mouse moves around your desk.

The scroll bar in the active window. Slide the rectangular white box up and down by pressing the mouse button down whilst the pointer is inside the box.

Cursor

This is the main output window for text and graphics. The Results-2 window is only for text and is hidden under the other windows on entering BASIC 2

This is the active window, with scroll bars highlighted

| Dialogue | Results-1 |

Ready

Slide here for left-right movement within the active window

Edit

Program listing and editing occurs in this window

Figure 3.1 An annotated view of the BASIC 2 opening screen

Now type the following on your keyboard.

```
PRINT 2+2 <ret>
```

(`<ret>` indicates that you should press the return key at the right of the keyboard)

You should now see the following output in the **Results-1** window.

4

Your computer can add, and the result is the sum of 2+2. You have asked the computer to perform an **immediate** action, and the **Dialogue** window is the place where you communicate 'directly' with the computer (or at least the BASIC 2 software that is currently controlling the machine). There are a number of things that you can do via the **Dialogue** window - most of the BASIC 2 commands can be executed in immediate mode. You can `RUN` or `LIST` a program, or execute one-line 'miniprograms', the only stipulation being that when you press `<ret>` the 'miniprogram' will be run. The calculation above

was an example of such a 'miniprogram'.

Now enter the following commands on the same line.

```
x=10  :  PRINT x <ret>
```

The computer should then print:

```
10
```

in the **Results-1** window as the answer.

If you have never programmed before, you have seen one of the most powerful elements of computer programming in action: the use of a **variable** to represent data in the computer. In this case, x is a variable, and the **statement** x = 10 means that X is assigned the value of 10. If you have no experience in programming, this is the point at which you should make a note to buy a BASIC primer (or carefully read Part II of the *Locomotive BASIC 2 User Guide*) before embarking on the following chapters of this book.

To some extent the **Dialogue** window is the least important window in the BASIC 2 user interface. Why is this so? Well, the more use you make of the menus and keyboard 'shortcuts' (of which more later), the less you will find yourself doing the things that BASIC programmers on other computers have to do, like typing the full names of commands as they need to be performed. An experienced BASIC 2 programmer will mostly use the **Edit** and **Results-1** windows for routine programming and program execution, so our next job is to look in some detail at the way that these windows function.

3.3   The Edit window

Elementary editing

You should still have the **Dialogue, Edit**, and **Results-1** windows in the same configuration to one another as you did when BASIC 2 was first loaded in this session. The **Results-1** window will still have some text in it from your experiments in section 3.2, so you may want to clean up this window before proceeding. Type:

```
CLS  <ret>
```

and the **Results-1** window will clear itself. (The command `CLS` means 'clear screen').

We now want to use the **Edit** window, but as you will have realized, text entered from the keyboard is going to the **Dialogue** window. We need to transfer control to the Edit window. This is done using the mouse.

Move the pointer on the screen to anywhere over the **Edit** window and click the left-hand mouse button. You will see that the title bar at the top of the **Edit** window becomes emphasized, whilst that of the **Dialogue** window becomes less bold.

You have now *activated* the **Edit** window and *deactivated* the **Dialogue** window. Your **Edit** window is now ready for editing. The **Edit** window as it first appears on entering BASIC 2 is rather small and cramped. To expand it you can do one of two things.

The first way to expand the **Edit** window is to click on the diamond shaped **full box** at the top right of the window, in which case the **Edit** window will expand to fill the screen. If you still want to see the other windows, you can alternatively move the **Edit** window upwards by clicking the mouse button on the title bar, and moving the mouse towards the top of the screen. You can then expand the window to the size required by clicking in the **size box** at the bottom right corner and dragging the box with your finger still on the mouse button.

Here is a very simple BASIC program that we can use to demonstrate the use of the Edit window. Type the program into the computer, remembering to press **<ret>** at the end of each line.

```
REM sample BASIC 2 program
WINDOW  OPEN
REM to calculate area of a circle
     INPUT"input radius";ra
     area=PI*(ra*ra)
     PRINT"area=",area
END
```

Note that the BASIC 2 keywords in the program are highlighted in upper case. In fact, you do not have to type then in like this. Each time you press the

<ret> key, the BASIC 2 software checks to see if it recognises a combination of letters to be a keyword. If it does, the word is made upper case. This is a useful check to see if you have used a keyword as a variable by accident. You might, for example, be tempted to use the word CIRCLE as a variable. This would not be allowed, and on running the program you would find that an error would be generated, because BASIC 2 expects CIRCLE as a graphics command to draw a circle. Variables will be forced to lower case, even if you type them in upper case.

After you have entered your program you may like to study it for a few seconds. If you have programmed before, you will see one fundamental difference between BASIC 2 and many other BASICs: there are no line numbers! In fact, you can use line numbers in BASIC 2, but they are not obligatory - you just use them for labelling lines that need to be accessed in the program. The second difference is the use of lower and upper case text, a feature that makes BASIC 2 easier to read. Finally, lines may be indented to help show the structure of the program more easily.

So far the **Edit** window has shown its ability to enter text, but how can the text be manipulated? Your computer has a number of special keys that can be used in the **Edit** window. You may have experimented with these if you made a mistake when typing in the above short program.

Did you make a mistake in the program? You can move the cursor to the point of the mistake by using the cursor keys, or by moving the pointer to the required point with the mouse and then clicking the lefthand button.

Once you have got to the position where you want to make changes, you will find a problem. Everything you type moves the existing text to the right. This is because you are in **insert** mode. In order to change to **replace** mode press the key marked **<Ins>**. Now any text you type will replace the text over which the cursor moves. To return to **insert** mode, press the **<Ins>** key again.

You can delete characters by pressing the keys marked **<Del** (on the IBM PC keyboard the key is marked ←) or **Del**, deleting to the left and right respectively.

Experiment with these keys before reading further. Make sure that you can delete, insert and change parts of the program you have typed. You might like to try running the program to check that you have entered it correctly. To RUN a program, you can select RUN from the Program menu, or reselect the **Dialogue** window and type RUN into this window. Note that the output will

go to the **Results-1** window.

Advanced editing

You may recall from the end of the last chapter that there is a shortcut that may be used to move between the **Edit** window and the **Dialogue** window. If you press the **f9** key while you are in the **Edit** window, the program will automatically **RUN** and the **Dialogue** window will be activated when the program ends. If you press the **f10** key from the **Dialogue** window you will enter the **Edit** window and vice versa. If you look at the entries under the Program and Edit menus you will see that several other operations have function key equivalents.

Once you are familiar with these operations you can attempt more complicated editing procedures. Unlike more rudimentary versions of BASIC, BASIC 2 allows you to use several 'wordprocessing' operations within the editor. You can delete, copy and move around whole blocks of the program by using simple commands.

In order to perform operations on parts of the program, you first need to highlight the required program block. BASIC 2 calls the parts of the program that you manipulate **areas**. You define areas by marking the **start area** and the **end area**.

To select the start area, move the cursor to the beginning of the area you want to move, delete or copy. Using the mouse, move to the beginning of the line of our test program containing the first **INPUT** command and click the mouse button to place the cursor in position. You will see a small box appear at the chosen position. In order to define the start area you can select the command **Start area** from the Edit menu (by moving the mouse to the Edit menu and running down it until **Start area** is reached, and then pressing the mouse button). Alternatively, you can press the **f1** key (remember that the keyboard shortcuts for menu items are listed alongside the items on the menus to aid you).

Now move the cursor to the end of the area required, again using the mouse. In this case, we have decided perhaps to copy the main part of the program so that the program calculates two circle radii before stopping. The cursor therefore needs to be at the end of the line containing the second **INPUT** command. Note that as you move the cursor, the region between the start position and the current cursor position is highlighted in black. You can select larger areas by scrolling down the program using the mouse and the scroll bars. When you have

located the end of the appropriate region to be edited, **End area** can either be selected from the Edit menu, or alternatively by using the **f1** key again. You will of course find the keyboard shortcuts of great help, and it will pay you to learn them quickly! The end position is marked with a small box, like the start position.

Figure 3.2 The Edit menu

Now that the area has been selected, you can do what you want with it. Select the Edit menu. You will see options to **Copy area, Move area**, and **Delete area**. There is even a command to **Cancel area** if you make a mistake during area selection. Don't do anything yet. First put the cursor at the beginning of the line containing the command **STOP** using the mouse or the function keys. Now press the **f3** key or select **Copy area** from the Edit menu. You should see the three selected lines copied, so that the program now reads as follows.

```
REM sample BASIC 2 program
WINDOW OPEN
REM to calculate area of a circle
     INPUT"input radius";ra
     area=PI*(ra*ra)
```

```
            PRINT"area=",area
            INPUT"input radius";ra
            area=PI*(ra*ra)
            PRINT"area=",area
      STOP
      END
```

You can remove the lines again by pressing the **f5** key or selecting **Delete area** from the Edit menu.

The final operation that we have to consider in the **Edit** window is vertical movement around the program. Although our circle program is only a few lines in length, you may be intending to write programs that are hundreds of lines of BASIC 2 statements. Clearly, the **Edit** window is not large enough to display the whole of such a program at one time. If you want to move up or down a few lines, the **scroll box** at the right hand side of the window can be used (move the bar up or down while you are holding the mouse button). The up and down cursor keys will also scroll for you in the same way albeit more slowly: you may find these more convenient to use.

If you need to move to the beginning of the program, you can use the **Home** key (or drag the scroll box to the top of the scroll bar). To get to the end of the program use the **End** key (or drag the scroll box to the bottom of the scroll bar). You can also move up or down a full window of information by using the **Pg Up** and **Pg Dn** keys. These commands are of no use with the short circle program, so you may like to load in one of the demonstration programs on your BASIC 2 disk at this point so that you can experiment with movement around a larger program.

To save the program that you are currently working on you, select **SAVE** from the File menu. A dialogue box appears and you are prompted to enter the name of the name under which you wish to save the program.

To load in a new program, select **LOAD** from the File menu, at which point a dialogue box appears on the screen. You can select the name of any program on the disk using this box. Before the new program is loaded you are prompted to save the existing program (if you haven't already done so), and when the new program loads, it overwrites the existing program in memory (goodbye circle program). You will then be able to look around the new program from within the **Edit** window.
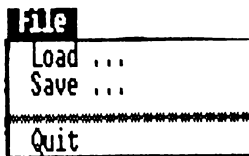
Figure 3.3 The File menu

3.4   The Results-1 window

We have already been using this window for program output, and its use is really very straightforward. Any text or graphic output that is *not specified to go elsewhere* will go to the **Results-1** window. For most simple programs this is the only output window that you will use. There is a second output window, the **Results-2** window, that can also be used for output, and this window differs from the **Results-1** window mainly in that it is a *text* window rather than a graphics window. The difference between text and graphics windows will be explained in Chapters 6 and 7. If you need to specify the **Results-1** window for output, it is stream #1. The concept of streams will be introduced in more depth in Chapter 5.

3.5   Handling the windows

The main problem that you will encounter when first using BASIC 2 will probably be getting to grips with using the windows. If you are used to a different version of BASIC using a single window, you are sure to get into difficulties when moving between the windows, especially if you have expanded one window to cover the others. 'Where has the output gone?' is a common question to ask when learning to use BASIC 2. Try to keep part of the **Results-1** window visible when developing programs, until you have gained enough confidence to switch between **Edit** and **Results-1** windows.

HINT: If you are having trouble finding your output, try putting the following lines at the top of your trial programs.

```
CLS
WINDOW  OPEN
WINDOW  FULL
```

These commands will clear and activate the **Results-1** window, ready for your output.

If you get really stuck and need to locate a particular window you can always use the windows menu to open the required menu.

3.6 Controlling program activity

We saw how to load in a new program using BASIC 2 in section 3.3, and the File menu is the easiest route by which you can **LOAD** and **SAVE** programs. The Program menu offers several facilities for controlling program execution. These are the commands **RUN, STOP,** and **CONTINUE**, with the keyboard shortcuts **f9, Ctrl-C**, and **f7** respectively. There are also menu items concerned with editing, these being the command **LIST**, which lists the program to the system printer, together with **NEW**, erasing the stored program from memory. As we have seen, **Edit** (keyboard shortcut **f10**) activates the **Edit** window to edit the current program in memory, and **Angles in ...** switches the units in which angles are calculated from angles to radians or *vice versa*. You can also change the angle units between degrees and radians from within your program using the **OPTION** command (see Chapter 6)
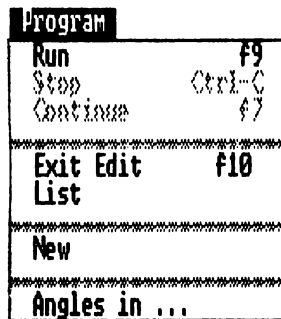
Figure 3.4 The Program menu

Loading programs from the Desktop

When entering BASIC 2 for the first time in a session, you may wish to load in

a pre-existing BASIC 2 program. You can do this automatically from the GEM Desktop by double clicking on the icon representing the program, instead of double clicking on the BASIC 2 icon. BASIC 2 will be loaded in as usual, together with the chosen program. Note also that a program loaded in this way will be automatically **RUN** when it is loaded. If you load programs directly from BASIC 2, this does not happen. You must make sure, however, that a copy of the program you wish to run is actually on a disk in the computer, but it need not be on the BASIC 2 disk itself.

Chapter Four

Numbers and Text

4.1 BASIC 2 numbers

One of the most noticeable differences between BASIC 2 and other computer languages (BASIC dialects included) is the use of only a single type of number. If you have programmed before you will be used to dealing with 'integer' and 'real' numbers (ie whole numbers and fractional numbers). You do not have to worry about number types with BASIC 2, as the language itself works out when integer and real numbers are needed.

You can type in numbers in any of the following forms:

```
123
-123
123.123
-123.123
+123
+123.123
```

You can also used scaled numbers and binary and hexadecimal numbers. Scaled numbers are most commonly used in technical areas and mathematics for dealing with very large or very small numbers. The chances are that if you do not already use scaled numbers, you will not find them useful in your BASIC 2 programs. A typical scaled number is:

$$3.4 \times 10^{-6}$$

and this is written in BASIC 2 as:

```
3.4E-6
```

If you need to use hexadecimal or binary numbers, they are specified by using the forms **&** and **&X** respectively. Again, if you haven't used them before you will probably not need to use them in BASIC 2. Here is the number 255 represented

in BASIC 2 using the hexadecimal and binary forms.

```
&FF
&X11111111
```

4.2 Variables

A variable is an entity within a computer program that possesses a name and a value. The name is decided by the user, and stays the same throughout the life of the variable. The value however may change. In some ways, variables are the life blood of computer programming, as they keep track of change within programs. There are two types of variables, *numeric* and *text strings*. As we will see, variable names in BASIC 2 have to obey certain rules, as do variable names in other BASIC dialects. Examples of simple numeric variable names are

```
cat
dog
maxvalue
```

and these can all be given numeric values

```
cat = 3
dog = 17.656
maxvalue = 10004567
```

Variables can be manipulated in the computer, so

```
maxvalue = dog * cat
```

is a nonsensical English statement, but is fine in BASIC!

Whatever form your number is entered into a BASIC 2 program, it will end up in the same internal format. Whether you define a variable as the number &FF, &X11111111, or 255, the number will be printed as 255 if you PRINT the variable, so

```
dog = 255
cat = &FF
PRINT dog,cat
```

will print the result

```
255  255
```

Variable names in BASIC 2 can be any combination of between 1 and 40 characters, as long as the first character is a letter, and the name is not a BASIC 2 keyword. Although a number of Greek characters can also be used, most programmers use only letters, numbers, and the underscore (_) character. The underscore is useful in making more readable variable names, and is used like this.

```
the_first_variable = 3

number_of_clients = 123
```

Variables in BASIC 2 can be entered in upper case, but they are automatically changed to lower case by the editor, as each line is entered.

4.3 Text strings

Text is handled in computers in the form of 'strings' of characters, and in BASIC 2 the length can be anything from 0 to 4096 characters. Variables are assigned text strings in a statement of the form:

```
name$ = "ANYTHINGYOULIKE"
```

where **name$** is the variable to hold the string (text string variables always end with the character $) and `"ANYTHINGYOULIKE"` is the string itself. Note that the string begins and ends with quotation marks.

A string of length 0 is called a null string, and is defined as **""**, that is two sets of quotation marks with nothing between them. **" "** is **not** a null string, but is a string of length 1 containing a space!

Text strings and numbers are handled quite differently within the computer, so you cannot write a statement like this:

```
name$ = 3
```

as a string variable cannot have a numeric value. You can however write:

```
name$ = "3"
```

making the variable `name$` contain the *character* 3.

4.4 Array storage

So there are only two types of variables in BASIC 2, namely *numeric variables* and *string variables*. There are two further ways in which variable information can be stored, however, and these are termed **arrays** and **records**.

Arrays are the most familiar of the two storage systems, and you may well be aware of their use in other versions of BASIC. An array is a collection of variables each with the same name, but with an individual numeric *index*. For instance think of a group of ten soldiers. Each could be called by the variable name `soldier`, and they could be referenced like this:

```
soldier (1)
soldier (2)
soldier (3) ... and so on.
```

`soldier` is the name of the array to which all the individual soldiers belong. As each *element* of the array (that is, each soldier) is a variable in its own right, the elements can all be made to hold some information, for example the age of the soldier concerned, so:

```
soldier (1) = 40
soldier (2) = 25
soldier (3) = 34 ...
```

Arrays can be one dimensional or multidimensional. The array `soldier` is a one dimensional numeric array. Perhaps we want to use a two dimensional array to reference the squares on a chessboard. We could represent such an array pictorially as shown on the next page.

Each location on the chessboard can be referenced by a row and a column number, so the black knight is shown in the array element:

```
chessboard (3,6)
```

where the location is given in (row, column) format.
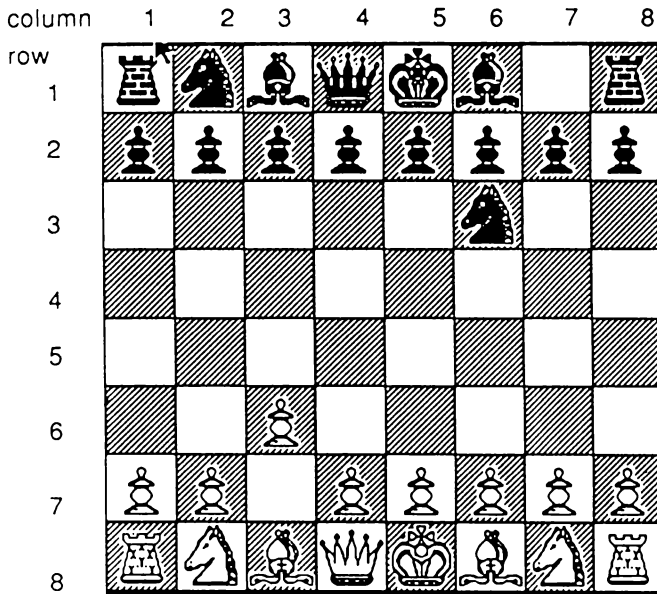
Figure 4.1 A chessboard as an example of a two dimensional array

You can have three or even more dimensions in an array, but ordinarily you will probably stick to one or two dimensions. In common with all BASIC versions, arrays are automatically defined with bounds from 0 to 10, for as many dimensions as are used. In order to use arrays with more elements than 10 in one or more dimensions in your programs you usually have to *define* your arrays before you use them.

To do this you use a statement of the form:

```
DIM soldier (20)
```

or:

```
DIM chessboard (8,8)
```

(In fact you would not theoretically have to define the chessboard array as it

does not exceed 10 elements in either dimension).

If you have already defined a variable, say n in your program, you can use the variable when defining the array, so:

```
DIM soldier (n)
```

is allowed.

Arrays are normally set up so that they are numbered from 0 upwards (our first soldier would therefore really be 0 in the example above). If you want to start array numbering at any other number you can specify the lowest and highest elements of the array in the following form:

```
DIM soldier (1 TO 20)
```

or more generally as:

```
DIM arrayname (first TO last)
```

A last point concerning arrays is that an array can hold character strings instead of numbers. Such an array is called a *string array*, and, like simple string variables, has a $ sign at the end of the variable name. Here is an example.

```
DIM soldier$ (1 TO 20)
    soldier$(1)="Jones"
    soldier$(2)="Smith"
    soldier$(3)="Brown"
```

4.5 Array classes and Records

Arrays eat up a lot of storage. If you have a multidimensional array of thousands of elements, the amount of space taken up by the array in your program may be quite critical. BASIC 2 gives you the option to reduce this storage requirement by defining the *storage class*.

If you do not define the class of storage for a numeric array, the 'ordinary numeric' class will be automatically set for you. This default size occupies 5 bytes for each element. If you are dealing with small numbers, you can cut the space needed for the array by one half or even more. Here are the space

allowances for the various classes:

Class	Range	Space occupied (bytes)
BYTE	-128,127	1
UBYTE	0,255	1
WORD	-32768,32767	2
UWORD	0,65535	4
INTEGER	-2147483648,2147483647	4
default		5

Compare the storage requirements of the following arrays:

```
DIM WASTEFUL (1 TO 100)

DIM COMPACT (1 TO 100) WORD
```

WASTEFUL occupies 500 bytes (= 5 bytes/element). COMPACT on the other hand occupies only 200 bytes (=2 bytes/element). Note that all storage classes *except* the default hold *integer* values.

Arrays are the standard form of *data structure* in most variants of the BASIC programming language. BASIC 2 offers you a more sophisticated form in which to store information, and this is called a RECORD. A record is really a kind of array that can hold mixed data types, and you will find out more about records in Chapter 7.

4.6   Handling strings

We have already seen that string variables can be used to store strings of characters. Here is a BASIC 2 program that sorts a list of string data, printing it in alphabetical order.

```
REM string sorting program
'first set up the output window
CLS
WINDOW OPEN
WINDOW FULL
'now set up a string array to hold the data
```

```
DIM a$(16)
'set dummy variable at end
a$(16)="zzzz"
'read in the data
  FOR i = 1 TO 15
  READ a$(i)
  NEXT i
LABEL top
f=0
i=1
LABEL next_step
  IF a$(i)<=a$(i+1) THEN GOTO jump
  t$=a$(i+1)
  a$(i+1)=a$(i)
  a$(i)=t$
  f=1
LABEL jump
  i=i+1
  IF i<=15 then GOTO next_step
  IF f=1 THEN GOTO top
'now print the data
FOR i=1 TO 15
PRINT a$(i)
NEXT i
WHILE BUTTON=-1:WEND
STOP
END

DATA BIRD,DOG,COW,ELEPHANT,MOOSE,ANEMONE,DOGFISH,CAT
DATA CATFISH,BUDGERIGAR,FINCH,DANDELION,DAISY,BULLDOG
DATA OCTOPUS
```

This simple program illustrates several things about handling string variables. Firstly, if you read string variables in as data in a **READ** statement, you do not use quotation marks " " to define each variable. Secondly, you can treat string variables 'numerically', ie "CAT" is less than "DOG"! BASIC is an excellent language for handling string data, and contains a variety of string handling commands.

Don't worry if you find this program difficult to understand. You may wish to consult a general text on BASIC, but much of the rest of this book will be

accessible without extra help. You can always come back to the more difficult program sections as you become more expert.

It is often necessary to perform operations on strings of characters. For instance, we may wish to find the number of characters in a string, or to find which character is in a specific position in a string. BASIC 2 contains a range of string-handling functions found in many BASIC dialects (see Appendix 2 for the full list). String functions are very useful in database-type programs where lists of names or objects are to be manipulated.

The short program below illustrates the use of the various string handling functions.

```
REM Program to demonstrate string handling
  INPUT "Input your name as christianname space
surname";name$
  PRINT"Very nice ..."
    length = LEN(name$)
      FOR i = length TO 1 STEP -1
      emas$ = emas$ + MID$(name$,i,1)
      IF MID$(name$,i,1) = " " THEN save = i
      NEXT i
    PRINT"Your christian name is ";LEFT$(name$,save)
    PRINT"Your surname is ";RIGHT$(name$,length-save)
    PRINT"Your name is",length-1,"characters long"
    PRINT"Your name backwards is",emas$
  STOP
  END
```

This program first works out the length of the name, using the function LEN$. It then reverses the name character by character using the function MID$. MID$ can be used to pick out single or multiple characters within a string using the syntax:

```
part$ = MID$ (whole$,start_position,substring_length)
```
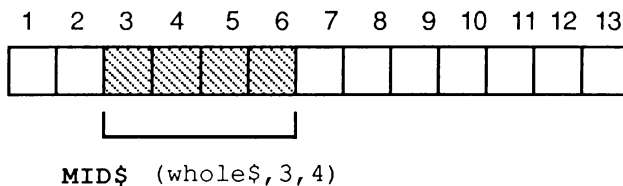
Figure 4.2 shows an example of the MID$ function.

MID$ (whole$,3,4)

Figure 4.2 The MID$ function in operation

In order to locate the christian name and surname, the space between the two names is located. The names can then be separately printed by using the functions **LEFT$** and **RIGHT$**. These both use the same syntax:

```
left_part$ = LEFT$(whole$,substring_length)

right_part$ = RIGHT$(whole$,substring_length)
```

In the following case, the five leftmost characters in a string are located.

LEFT$ (whole$,5)

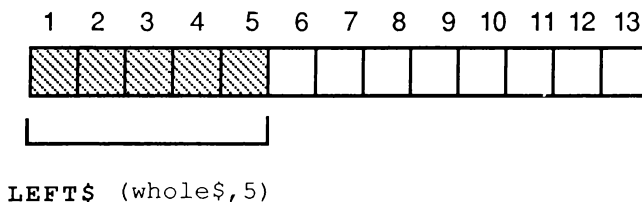**Figure 4.3** The LEFT$ function in operation

4.7 BASIC 2 Functions

A function is a rule that relates one set of values to another. In computer terms, a function is an operation performed on a constant or variable to obtain a *result*. We have already seen examples of string functions, and here we will concentrate on functions involving numbers. For example:

```
log_number = LOG(10)
```

or:

```
hex_value$ = HEX$(dec_value)
```

The full range of BASIC 2 functions are given in Appendix 2. Besides the 'normal' range of functions available in most BASIC dialects, a number of WIMP-specific functions are available for reading the mouse position, or for obtaining information about the windows.

As there are around 100 BASIC 2 functions, they will not all be described in this short section. Here instead are several short program segments that illustrate the use of a few functions. Note that these segments are not complete programs in their own right, as no window-handling commands are included. You should however be able to embellish these short program snippets without too much trouble.

```
REM use of the ROUND function
'rounds a real variable to the nearest 32 bit signed
integer
INPUT"interest rate (%)";interest_rate
interest_day = interest_rate/365.25
INPUT"how much money do you have";cash_in_hand
INPUT"how long will you have it";days
profit = interest_day*cash_in_hand*days
profit = ROUND (profit)
PRINT"your interest will be:;profit;"£"


REM use of the UPPER and LOWER functions
'to calculate upper and lower bounds of an array
DIM example(5 TO 50)
LABEL retry
INPUT"which array element do you wish to
change";number
IF number <LOWER(example) OR number>UPPER(example)
THEN PRINT"no such element":GOTO retry


REM use of the TRUNK, FLOOR and CEILING functions
INPUT "type in a positive or negative decimal
number";real_num
PRINT"your number is";real_num
PRINT"rounded towards 0 it is",TRUNC(real_num)
```

```
PRINT"rounded towards - infinity it
is",FLOOR(real_num)
PRINT"rounded towards + infinity it
is",CEILING(real_num)
```

You may like to note that **FLOOR** and **TRUNC** are equivalent to the functions **INT** and **FIX** respectively. Both the latter functions may also be used in BASIC 2, and these names are retained for compatibility with older BASIC versions.

Chapter Five

Programming in BASIC 2

5.1 BASIC 2 vs most other BASICs

Up until now we have been concerned with the mechanics of using BASIC 2, and a number of features of the language have been described. In fact if you have already had some contact with a BASIC dialect you will already be able to do a lot of simple programming operations in BASIC 2 by this stage. The main areas that require specialized knowledge are graphics programming and the handling of disk files, and these topics are covered in Chapters 6-8 below.

The present chapter is something of a half-way stage in the book. We'll look at a number of simple BASIC techniques and see how they translate to BASIC 2. Hopefully, this chapter will give you some ideas on how to improve your own programming skills with the facilities offered by BASIC 2. You already know something about the GEM based user interface, windows, mouse and so on. Let us start this chapter by considering the fundamentals of BASIC 2, the commands seen in any BASIC 2 program.

The first thing that you will have noted about BASIC 2 programs is that they do not have line numbers on every line. In other BASIC dialects, line numbers are an essential part of the language as they enable the BASIC interpreter to keep track of the various parts of the program. If you are writing a program with more than a few lines however, they are a nuisance: the only real value of line numbers is to *label* lines. Look at the following two short programs, one in BASIC2, and the other in a cruder version of BASIC.

```
REM BASIC2 PROGRAM          REM BASIC PROGRAM
10  PRINT "hello there"      10  PRINT "hello there"
        FOR i = 1 TO 10      20  FOR I = 1 TO 10
            PRINT i          30  PRINT I
        NEXT i               40  NEXT I
    INPUT"more?";i$          50  INPUT"more";I$
IF i$="Y' THEN GOTO 10       60  IF I$="Y" THEN GOTO 10
```

There will obviously be no prizes awarded for the quality of these programs, but you can immediately see something of the flexibility offered by BASIC 2. The only line labelled is the line that **needs** to be labelled. BASIC 2 forces keywords into upper case characters whilst forcing variables as lower case. Indenting of the lines is allowed. These facilities make for much more readable programs than the old style illustrated in the right hand example. Labels do not even have to be *numbers*. So long as the label name is defined, you can use meaningful names for the labels, so:

```
LABEL loopstart
GOTO loopstart
```

are the label and a jump to the label respectively.

Note that *numeric* labels (line numbers) do not have to be defined in a **LABEL** statement. This facility is important, as it maintains compatibility with other, line number oriented BASIC dialects. So long as you use the appropriate window-handling commands you should be able to run many text-based BASIC programs with very little alteration to BASIC 2. However, you may well find that you want to embellish such programs with the extra features that are available within BASIC 2.

Loops

Many programmers advise caution in the use of labels and jumps to labels. This is to prevent programs from becoming undebuggable masses of GOTO statements. In early versions of BASIC, it was difficult to avoid using GOTO statements, because the language was rather primitive and lacked enough *control structures* to allow any other way of getting around the program. BASIC 2 is not alone in rectifying this situation, but the language is especially rich in control structures. Here are the possibilities.

BASIC 2 offers the standard **FOR NEXT** loop, in common with other BASICs, and this was demonstrated in the first programming example of the chapter. Several alternatives exist, and are useful when a program is being interacted with using the keyboard or mouse. These are **WHILE WEND** and **REPEAT UNTIL**. They may be used like this:

```
WINDOW    OPEN
WINDOW    FULL
```

```
WHILE BUTTON (1) = -1
PRINT"please press the left button"
WEND
```

which will continually print the message until you get fed up and press the button!

```
WINDOW    OPEN
WINDOW    FULL
REPEAT
PRINT"please press the Y key"
UNTIL INKEY$ = "Y"
```

You may have noticed the subtle difference between the two sorts of loops. The **WHILE WEND** loop will not be executed at all if the condition tested for is not true on entry to the loop. The **REPEAT UNTIL** loop on the other hand is always executed once, because the test is at the **end** of the loop!

Conditional statements

Conditional statements are those that are only executed if certain conditions are true, and the two types of loops that we have just looked at are conditional loops.

As with other BASICs, BASIC 2 offers the **IF THEN** structure:

```
IF x=3 THEN y=4
```

```
IF x=3 THEN y=4:z=5
```

and BASIC 2 also suffers from the disability that the structure must be on one program line. The option **ELSE** is also available.

```
IF x=3 THEN y=4 ELSE y=5
```

It is possible to nest **IF THEN** statements, but they must all be on the same program line. You can get over the restriction to one program line by using the **GOSUB** command. Subroutines are defined as a block of the program that starts with a label and ends with a **RETURN** statement. The following short program illustrates the use of the **GOSUB** command in conjunction with a conditional structure.

```
REM GUESS A NUMBER
    CLS
    WINDOW   OPEN
    WINDOW   FULL
    LABEL top
    PRINT "I'm shaking a dice..."
    INPUT"What number came up?";num
    GOSUB dice_shake
IF num=dice_no THEN PRINT "well done!" ELSE PRINT
"bad luck, it was",dice_no
    INPUT"Another go? Type Y or N";trial$
    IF trial$="y" THEN GOTO top

END
LABEL dice_shake
    dice_no = INT(RND(6))+1
RETURN
```

The **RND** function used in this program is a random number generator in the range 0 - 1. To get a number in a specific range, it is therefore necessary to multiply the **RND** number by the maximum number required and then to add one to it. Random numbers are used extensively in simulations and games (they are used for example to set where the 'space invaders' appear in the archetypal computer game), and we will consider more complex programs using them later in this book.

Operators

There are two further classes of command that should be dealt with in this first section. The first group are not really commands as such, but are more commonly termed *operators*. There are two groups of operators. The first are called *relational* operators, for instance < (less than) and >= (greater than or equal to. These operators are always used in conjunction with **IF** statements, for instance:

IF pay < cost **THEN GOTO** nobuy

IF credit = 0 **THEN PRINT** "that won't do nicely!"

The other operator group comprises the *logical operators*. These are **AND**, **NOT, OR** and **XOR.** Again, they are often used in conditional statements:

```
IF pay=ok AND credit=good THEN PRINT"that'll do
nicely!"

IF name$ = "Charles" OR name$ = "Andrew" THEN PRINT
"Prince!"
```

Logical operators can also be used in evaluating expressions in the same way as the arithmetic operators (+, -, /, *). Examples of this type of use include:

```
x = 1 AND 1

y = 2 OR 2
```

The values of x and y become 1 and 2 respectively.

Functions

Besides the functions already available within BASIC 2 (described in Chapter 4 and listed in Appendix 2), you can define your own functions within a program. The command

```
DEF FN name (parameters) = expression
```

is used for this purpose. The name can be any variable name, and the expression is the operation to be performed on the parameters. Here is a simple illustration of the use of a user-defined function.

```
DEF FN percentage (portion,whole) = (portion/whole) *
100

PRINT FN percentage (30,50),"%"
```

and the result that will be printed will be **60%**.

Note that there must be the same number of parameters listed when you call the function as when it was first defined. You must also remember to define the function before you try to use it!

Remarks

Like other dialects of BASIC, BASIC 2 implements the **REM** keyword, and all other characters on the same line are ignored by the BASIC 2 interpreter. BASIC 2 offers a shorthand version of **REM** that offers a more elegant solution to sprinkling comments throughout your program than does **REM** itself. This is the use of the single quote (') as a substitute for **REM**. The **REM** command should be used with caution, because it cannot be used after the following BASIC 2 reserved words. (These keywords take the rest of the command as the parameter and so should always be used on a single line.)

DATA
DEL
DIR
ERASE
REN
TYPE

Machine code programming

BASIC 2 users coming from other versions of BASIC may be used to either embedding machine code subroutines into their programs, or directly acccessing memory locations using the **PEEK** and **POKE** commands. You will be interested (or annoyed) to find that BASIC 2 deliberately offers no access to machine code. The reasons for this are probably that the combination of GEM and the computer operating system are more complex to access than the operating environment on most small computers, and if you are an expert machine code programmer you will probably not be using BASIC 2 for program development. As BASIC 2 is a very fast version of BASIC, the need for speeding up program execution (the main reason for using machine code) is of less importance than it would be if you were using an 'inferior' BASIC dialect.

Graphics

You will learn the major principles of BASIC 2 graphics in the next two chapters. If you are used to BASIC games programming you may be sorry to find that no provision for 'sprites' (moveable object blocks) is available within BASIC 2. This means that you will be unable to define and animate space invaders or 'matchstick men' as easily as you can on, say a Commodore 64 or

other home computer. This will only be of worry to you if you are primarily interested in writing games on your computer, and you will find that BASIC 2 is so fast that the advantages far outweigh the limitations of having no sprite manipulation commands. Several versions of BASIC, notably BASICA running on IBM PCs and Microsoft BASIC v2.0 on the Apple Macintosh allow the user to 'capture' parts of the screen and to subsequently recall the captured images. This facility is not offered in BASIC 2.

In the simplest sense, BASIC 2 graphics commands are similar to those on many computers, including the Amstrad 6128 and IBM PC. You can draw lines, circles, fill shapes with colour, and so on. In addition to this, commands for handling the windows and other components of the GEM interface are included. Some information for translating graphics between BASIC 2 and other BASIC dialects may be found at the end of this chapter.

Sound

Games fanatics will also be upset to find that BASIC 2 allows no use of sound at all via specialized commands. The lack of sound, even a humble beep or click, is a definite disadvantage, as there are occasions on which the user needs to be alerted that something has happened in the program. You can to some extent overcome this problem by use of the BEL character.

```
REM this line will beep at you!
PRINT CHR$(7)
```

Multiple 'beeps' can be obtained by printing more than one BEL character. There is no facility within BASIC 2 for setting volume, so if you have a computer without a volume control you will have to consult your user manual for details of how to change sound levels.

5.2 Program input

BASIC is a very easy language for input and output of data to and from a program. Those of us who have wrestled with input and output on mainframe computers running languages like Fortran can especially appreciate the advantages offered by BASIC. BASIC 2 has a full range of the more 'traditional'

BASIC input commands, so let us briefly look at examples of how to use these.

```
INPUT x

INPUT "type in x please";x
```

The first example will print a ? on the screen, and the computer will wait for the data to be typed in. Nothing will happen until a `<return>` is typed after data entry. The second statement above shows that a prompt can also be included in the `INPUT` statement. Such information can be very useful in helping the user, especially if he or she is not the person that wrote the program!

Both these uses of the `INPUT` statement read data from the keyboard. They can be used to input any data type: integer, real number or character string. In the latter case, the variable to be input must be defined as a string variable, or else an error will result if you press a non-numeric key.
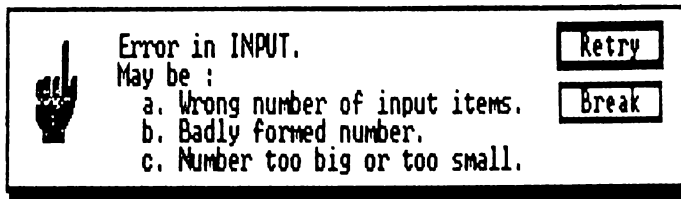
Figure 5.1 'Error in Input' alert box

You can use `INPUT` to take data from a disk file, in this case you need to set the stream number. (Disk file access is discussed in Chapter 8).

A handy variation of the `INPUT` command is the function `INPUT$`, and this function is used to input a fixed length string from the keyboard. It inputs the set number of characters and then continues with the program. No `<return>` has to be entered. The input characters are not displayed on the screen, making `INPUT$` ideal for entering information like passwords into the computer.

```
REM password routine in BASIC 2
REM first set the password
   PRINT"Set your password"
   INPUT pass$
   pass_length = LEN(pass$)
10 PRINT"You cannot proceed without typing the
```

```
password"
    trial$ = INPUT$(pass_length)
    IF trials$ <> pass$ THEN GOTO 10
```

You could of course make this routine as complex as you like. For instance, you could only allow the user to make a set number of attempts to enter the password before the program is aborted. The password could also be stored on disk. A BASIC program is not the most secure type of program, but sometimes even a little hardship is enough to deter potential troublemakers!

You can also input information without waiting for the data! Why would you want to do this? Consider a program that, say printed the current time according to the system clock every 10 seconds (a very boring program). The program is set to cycle indefinitely. How would you stop it? One rather inelegant way is to press the **Ctrl-C** keys, but this will break into the program. A neater way is to get the program to scan the keyboard at each loop of the program to see if a key has been pressed. A conditional statement can then be used to break out of the loop on the pressing of a specific key. The functions to read the keyboard are called **INKEY** and **INKEY$**.

```
REM program to print the system time
    WHILE INKEY$<>"S"
        PRINT TIME
    WEND

END
```

INKEY$ is easy to use, as you just specify the relevant key within quotes. **INKEY** on the other hand returns a numeric result depending on the numeric code of the key being pressed (see Appendix 3). **INKEY** is useful if you need to specify a key not represented by a character, for instance the **f9** key.

5.3   Using the mouse

The most novel form of input in BASIC 2 is undoubtedly the mouse. In the simplest case, the mouse buttons can be used as substitutes for keys on the keyboard. The press of a mouse button can be detected by the computer, and such an activity can be used in similar situations to the **INKEY$** function. The function for reading the mouse buttons is aptly enough called **BUTTON**, and it

is used in the following way:

```
result = BUTTON (button)
```

where button is an integer number in the range 1 to 16. If you have two buttons on your mouse, the lefthand button will be 1, and the righthand button will be 2. Buttons are numbered from left to right.

In most cases, the mouse will also be used as a pointing device. As you will have found out by now, movement of the mouse around a horizontal surface is mirrored by movement of the pointer around the BASIC 2 screen. The horizontal and vertical screen coordinates of the mouse position are stored in the computer in memory locations accessed by the **XMOUSE** and **YMOUSE** functions.

The ability to locate the mouse position on screen adds enormously to the ease of use of your computer. The manipulation of the GEM environment depends on the mouse: as proof of this, try using the cursor keys instead of the mouse for a few minutes and you will quickly see the value of your little friend!

HINT: When writing programs you will find that the **Dialogue** window is reactivated as soon as the program ends, covering the current output window. This can be very annoying, especially if the program has created some graphics output that you want to keep on the screen. To avoid this problem, put the following line just before your **END** statement. It will 'hang' the program until a mouse button is pressed.

```
WHILE BUTTON = -1:WEND
```

You should get into the habit of using the mouse in your BASIC 2 programs, and not just use it as a handy way of accessing the various windows and menu items whilst you are developing programs. We have not yet looked at how to measure our way around the screen, so specific examples of mousemanship will be left to the next chapter, but in the next section we will look at a pictorial example of how the visual, mouse-driven interface scores over its text-based rival.

Alert boxes

Let us imagine that we have arrived at the point in a program where a simple yes/no choice has to be made by the user. In most forms of BASIC, the choice

would be handled on screen like this:

```
PROCEED? (Y/N)...
```

to which the user types:

```
N
```

and the program branches accordingly.
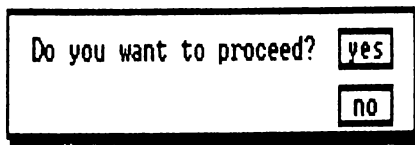
Here is the BASIC 2 equivalent.

Figure 5.2 User defined alert box

and the user points at the required choice and clicks the mouse button.

The box shown in the above figure is called an *alert* or *dialog* box. Dialog boxes are a common feature of computer visual interfaces, but in order to avoid confusion with the BASIC 2 **Dialogue** window, we will refer to them as alert boxes.

Alert boxes are handled very easily within BASIC 2. The command syntax is:

```
ALERT icon number TEXT textstring [,textstring ..]
BUTTON textstring [,RETURN][textstring..]
```

The *icon number* on the Amstrad 1512 can be:

0   no icon
1   note "finger"
2   question mark
3   stop "full hand"

Other icons may be available on other versions of GEM. If **RETURN** is specified with a **BUTTON** textstring, that button is shown highlighted within the alert box.

Here are two varying box styles. The accompanying program specifies the various alert boxes. The maximum number of comment lines that can be used is 5, and the maximum number of characters on each line is 40 (longer text strings are truncated). Up to three buttons may be specified within an **ALERT** box, and the maximum number of characters within a button is 20.
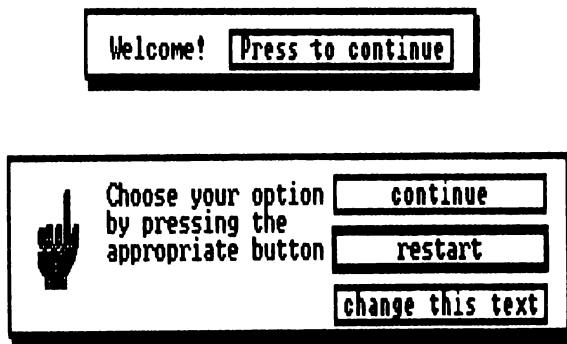
Figure 5.3 Two different alert box styles

The size of the **ALERT** box is self regulating, so a box containing a few short words will be much smaller than a verbose box. As the **ALERT** command temporarily suspends program execution until a button is pressed, only one **ALERT** box can be used at a time.

```
REM Alert box demonstration program
    'set up text for changeable button
    t$ = "change this text"
    LABEL top
    'first clear screen
    CLS
    WINDOW   OPEN
    WINDOW   FULL
```

```
    'now set up a box
    ALERT 0 TEXT "Welcome!" BUTTON "Press to
continue"
    'now set up a full blown alert box
    'note the use of the RETURN option on the centre
button
    alert_value = ALERT 1 TEXT "Chose your
option","by pressing the","appropriate button" BUTTON
"continue",RETURN "restart",t$
    '
IF alert_value = 1 THEN PRINT "continuing program
..."
IF alert_value = 2 THEN GOTO top
IF alert _value = 3 THEN INPUT "type in new
text";t$:GOTO top
    '
WHILE BUTTON = -1 :WEND
```

Note how the **ALERT** command is used in the above program. **ALERT** is used in the *assigned command* form, so that in each case the variable alert_value contains a result depending on which **ALERT** button was pressed. The program can then be made to branch depending on the returned value. The value is 1,2 or 3, with the topmost button being 1. If the **RETURN** option is specified, pressing the **return** key on the keyboard has the same effect as clicking the button marked **RETURN.**

5.4 Translation from other BASIC dialects to BASIC 2

Many BASIC 2 users will be familiar with other BASIC dialects, and in Chapter 5 we considered many similarities and differences between BASIC 2 and its elder brethren. If you have programs written in other BASICs you may want to carry out quick and easy translations into BASIC 2. As with most computer programming, translations are fairly simple unless graphics are involved. It is in the graphics area that most differences usually occur. A simple program such as:

```
10  FOR i = 1 to 100
20  PRINT i
30  PRINT LOG(i/50)
40  NEXT i
50  INPUT "do it again";a$
```

```
60  IF a$ = "y" THEN GOTO 10
70  STOP
```

will work on practically any microcomputer running BASIC.

Windows

The most important point to remember when doing program translations to BASIC 2 is that you will have to set up your output window(s). The good old standby:

```
CLS
WINDOW  OPEN
WINDOW  FULL
```

will suffice for most output, because most BASIC dialects will be expecting one full screen size window.

Text input and output

Text input and output using commands like **PRINT**, **INPUT**, **INKEY$** and **LOCATE** are similar in many BASIC versions. You may find that simple programs using such commands run with little conversion needed. The exception to worry about is **LOCATE** because the **Results-1** window only allows 20 rows of text in default mode. Many micros have 24 or 25 rows on screen, and unless you want a rude error message you will have to amend the translated program accordingly. By using the appropriate **SCREEN** command (see next chapter) you can increase the number of lines on screen to 22.

Graphics coordinates

Graphics commands like **MOVE, LINE** and **CIRCLE** in BASIC 2 employ user coordinates to specify locations on screen. Many BASIC dialects also use similar commands, but you will need to first check the syntax of the equivalent command in BASIC 2, and in addition you must translate the coordinates from those applicable to the machine on which the program originally ran. Translating coordinates is a simple task. All you have to do is to find the screen dimensions on the chosen machine, and then to multiply x and y coordinates by

an appropriate constant for the BASIC 2 screen. Here are some example screen dimensions with the appropriate conversion constants to BASIC 2:

IBM PC and compatibles (BASICA or GWBASIC)

640 x 200   12.5x   25.0y

Amstrad 464, 664 and 6128 (Locomotive BASIC)

640 x 400   12.5x   12.5y

Commodore 16, Plus 4 and 128(BASIC v3.5, BASIC v7.0)

320 x 200   25.0x   25.0y

Apple II series (Applesoft BASIC)

280 x 192   28.57x   26.04x

Commodore, Apple II and IBM screens both *reverse the Y axis*, so the 0;0 origin is at the top left of the screen, whereas Amstrad computers use the much more sensible bottom left corner as the origin. Let us consider two sample translations from IBM BASICA, and Amstrad Locomotive BASIC to BASIC 2.

Example 1

```
LINE (100,100)-(200,200)  (in IBM BASICA)
```

to translate this to BASIC 2, we first need the appropriate BASIC 2 line drawing command, which is:

LINE 100;100,200;200

This will of course be a very short line as the user coordinates in BASIC 2 use a larger range of units. First we must multiply the BASICA coordinates by the appropriate constants given above.

LINE (100*12.5);(100*25.0),(200*12.5);(200*25)

Because the IBM y coordinates start at the top of the screen, we must also reverse them, or else the graphics will be drawn upside down. This operation is

performed merely by *subtracting the translated y coordinates from the maximum y value on the BASIC 2 screen, ie 5000*. The final translated command is therefore:

LINE
`(100*12.5);5000-(100*25.0),(200*12.5);5000-(200*25)`

Rather than perform these calculations in the program, you may well find it easier to work them out using a calculator as you rewrite the program. The choice is yours.

Example 2

```
MOVE 100,100
DRAW 200,200    (in Amstrad Locomotive Basic)
```

Although this line uses two commands rather than the single command used in BASIC 2, the translation is relatively simple.

LINE `(100*12.5);(100*12.5),(200*12.5);(200*12.5)`

As all Amstrad computers have the origin at the bottom left corner of the screen, you will not have to worry about the subtraction step necessary with IBM conversions. You can in this case use a BASIC 2 shortcut to program translation by using the **SCREEN** and **USER SPACE** commands to adjust the BASIC 2 coordinate space to that used in the computer from which you are translating the program. The use of these commands is described in the next chapter.

So long as you understand what each graphics instruction in your program is doing, you should have little difficulty in translating graphics based programs to BASIC 2.

5.5   Defaults

With more restricted dialects of BASIC, the 'changeable' elements of the interface between user and computer are rather limited. On a colour computer, the colour of text, graphics and background can be changed, and the output can be routed to screen or to printer. A sophisticated version of the language such as BASIC 2

offers a wide variety of options to change such elements as the measurement of angles in degrees or radians, the font being used, the font size, the font style, fill pattern, window to use for output and so on.

The options set on entry to BASIC 2 are known as the *defaults*. You cannot permanently change these default values, as they are preset within the BASIC 2 application. However, you can change them within a program, and it is possible to 'reset' the default values temporarily outside of a program, until you quit from BASIC 2 in a session.

There are several ways in which you can change the default values. These are:

Global change (either from a menu or by a command within the program)
Change temporarily within a command
Change temporarily for *part of* a command

Let us consider a typical default and look at the ways in which it can be changed. The default colour for text output is black (code number 1). We can change this colour by selection of a different colour from the Colours menu. If you select red (code number 2) from this menu, all output text will be in red until you quit BASIC 2 or redefine the colour. We have *globally changed* the text colour.

An alternative way of globally changing the colour is by a command in BASIC 2. Such a command can be directly entered in the Dialogue window, or can be embedded within a program. To change the text colour to red, the command will be

`SET COLOUR 2`

and all text will be subsequently output in red.

You may wish only a particular output line to be in red. Say you want to emphasize something (perhaps you are writing an accounts program and want to literally mark someone 'in the red'! To do this, you can temporarily change the default thus

`PRINT COLOUR(2) ;"WARNING - CUSTOMER OVERDRAWN`
`BY",amount`

Only the specified line will be highlighted in red. If you want to have the message only in red, you could change the colour of only part of the line.

```
PRINT COLOUR(2)  ;"WARNING - CUSTOMER OVERDRAWN BY";
COLOUR  (1)  ;amount
```

Note that this command has to specify the colour for the variable amount as it is within the same **PRINT** command as the message in red. You could alternatively have typed

```
PRINT COLOUR(2);  "WARNING - CUSTOMER OVERDRAWN BY";
```

```
PRINT amount
```

In this case amount will automatically appear in black, as the default colour will take over after the first line is completed.

5.6   Streams

Defaults are also used for the output stream in BASIC 2. Before considering what the defaults are for streams, let us list the available streams.

Stream	Window	Type	Function
1	**Results-1**	graphics	default text and graphics output
2	**Results-2**	text	second text window
	Edit	text	
	Dialogue	text	

Note that each stream is equivalent to a screen window. The default output stream is #1, **Results-1**, and the great majority of programs will use only this window. The **Results-2** window may be of use in certain instances where you want to have text output in a separate window to the main part of the output. An example of the use of the **Results-2** window will be found in Chapter 9. It *is* theoretically possible to use the **Edit** and **Dialogue** windows (ie, windows 3 and 4) for output, but they will revert to their 'normal' functions when the program stops, so their use is not advised. Any output that is not asssigned to a particular stream will go to stream #1 (ie the **Results-1** window). You can change the output stream globally by using the command **STREAM.**

for example will switch the output to stream 2, the **Results-2** window.

You will find out more about the equivalence between streams and windows in the next chapter. For the moment, note that switching output to a different stream will not necessarily make it visible on your computer screen! You will have to open the window, and place it in a suitable position for your needs. As with other defaults (see last section), the default can be temporarily overridden within a command, for instance

PRINT #2,"this should appear in the Results-2 window"

Besides the four screen windows, other streams may be defined. Stream 0 is the system printer, and GEM defaults for other types of output may be found listed in Chapter 9. Streams 5-15 are usually available for disk file use.

Chapter Six

Graphics and BASIC 2

6.1   Computer graphics and BASIC 2

Text and graphics output can be displayed in a variety of ways in BASIC 2. In the simplest case, output is routed to the **Results-1** window. In this chapter we will concentrate on output to this window, and so we can conveniently avoid worrying about output to other windows, printers, plotters and so on. These complications are best left until the general principles have been covered.

Computer graphics covers a wide area of application within computer programming, and to do it justice many whole books have been written. Although study of graphics techniques can involve a lot of hard work, the BASIC 2 GEM environment will enable you to produce some sophisticated and polished-looking graphics effects for minimum effort. If you want to draw complex three-dimensional shaded pictures or design programs for computer-aided design you will need more than this book, it is true but for the moment let us look at the building blocks that BASIC 2 offers the graphics programmer.

BASIC 2 graphics come in three flavours. Firstly there are text handling facilities. The most elementary of these can hardly be called graphics: the BASIC 2 statements:

```
X=100
PRINT X
```

will output the number '100' to the **Results-1** window, but is rather unambitious. BASIC 2 enables you to do more than this with text. You can change fonts, or styles, colours or emphasis of the text. You can even change the size at which the text is displayed. We will look at the options available for text manipulation within BASIC 2 in the next chapter.

The second form of BASIC 2 graphics is the generation of lines, curves, shapes, filling patterns and so on that are usually associated with the term 'graphics'. You will not be surprised to learn that BASIC 2 has a large variety of such

commands, and as an example consider the statement:

```
BOX  1000;1000,  2500;2500  ROUNDED  FILL  WITH  5
```

This command will output the following shape to the **Results-1** window.
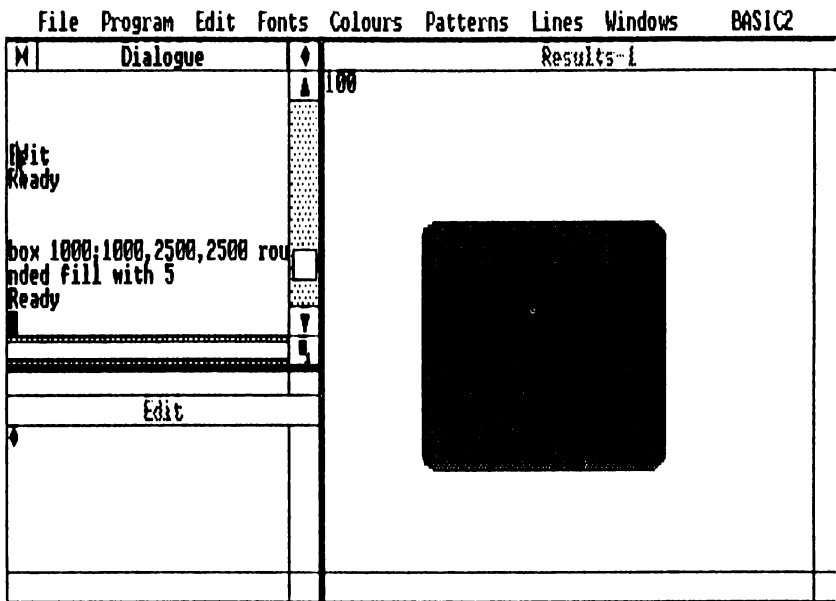
Figure 6.1 Simple graphics output generated by direct commands in the **Dialogue** window

The final group of graphics commands are those for generating 'turtle graphics'. Turtle graphics are most commonly found in the computer language Logo, developed to help children grasp the complexities of communicating with a computer. The central feature of turtle graphics is the 'turtle', a small pointer marking the graphics pen position. A novel feature of BASIC 2 turtle graphics is that you can mix turtle graphics and other graphics forms, although some care is necessary, as the same cursor will be used in each case.

To draw a square with turtle graphics commands in BASIC 2 you could use the following statements.

```
FORWARD   20
LEFT   90
FORWARD   20
LEFT   90
FORWARD   20
LEFT   90
FORWARD   20
```

(The numbers after the **FORWARD** commands are distances in user coordinates, those after the **LEFT** commands are angles in degrees.)

6.2   Graphic facilities

We will now move on from this general introduction to consider how to generate graphics using BASIC 2.

The graphics world

Whereas the text screen consists of a number of character cells on the screen (the text screen will be discussed in Chapter 7), the graphics screen has a finer reference grid. This grid can be referenced in a similar way to the character cell system, and in fact there are two different ways of addressing points on the graphics screen. The first of these uses the **resolution** of the screen in **pixels**: the physical number of pixels (points) on the computer monitor. If you have a standard monitor there are likely to be 640 such points across the screen, and 200 down the screen. That makes 128,000 points in all. However, if you wrote programs directly addressing the points, it would make it difficult to, say, take advantage of a monitor with a higher resolution.

BASIC 2 gets around this problem neatly by specifying **user coordinates** in which the screen is measured. User coordinates allow you to set the screen units (within certain limits) as you wish. Although user coordinates have defaults that are preset when you enter BASIC 2, you can redefine them if you wish. This redefinition is especially useful if you are translating coordinates from a program running on a different computer. We have already mentioned this possibility at the end of the last chapter, and will discuss details later in this section.

There are two main commands that are used to set up a graphics screen and to change the scaling of the screen. These are the commands **SCREEN GRAPHICS** and **USER SPACE**.

The general form for the **SCREEN GRAPHICS** command is

```
SCREEN [#stream] GRAPHICS width [FIXED] height
[FIXED] [MINIMUM w,h] [MAXIMUM w,h] [UNIT w,h]
[INFORMATION switch]
```

This rather complex looking command may be broken down as follows.

FIXED sets the dimension of the window to the same as the dimension of the virtual screen. The maximum size of the virtual screen is the same as the actual screen size in pixels.

MINIMUM and **MAXIMUM** set the minimum and maximum width and height for the window (the units are again in pixels).

The units by which the window size may be altered is specified by the **UNITS** option. The default is a single pixel. **INFORMATION** is a switch (either **ON** or **OFF**) specifying whether or not the window should have an information line.

The **SCREEN GRAPHICS** command clears the window and virtual screen, and the origin, text and graphics colours and other values are reset to the default.

The **USER SPACE** command takes as its argument the width and height (in units) that you wish the screen to be. If you only specify a single argument, BASIC 2 takes this as the shortest axis and automatically calculates the unit length on the other axis to maintain a 1:1 aspect ratio.

```
USER [#stream,] SPACE width, height
```

Most of the programs in this chapter will use the default **USER SPACE** of 5000 x 8000 user coordinates and the default **SCREEN GRAPHICS** area of 640 x 200 pixels, and you will only need to change these values if you wish to 'import' graphics data from other computers. Lets consider a BASIC 2 programmer moving from an Amstrad 6128 to a PC1512. He is used to an area of 640 x 400 units for graphics programming. In order to reference the same area within BASIC 2 to fill the whole screen, a single command is needed

```
USER SPACE 400
```

that is all there is to it!

If this sounds a little confusing, try the following simple test. Start up BASIC 2 and without disturbing the initial window positions enter the following in the **Dialogue** window

`LINE 1;1,400;400`

A very short line should appear extending diagonally upwards from the origin in the lower left hand corner of the **Results-1** window. The default user coordinates setting the shorter axis to be 5000 units are in force, so the line is very short. Now enter the next command in the **Dialogue** window.

`USER SPACE 400`

and next type in the command to draw the line again

`LINE 1;1,400;400`

You will now see that the line extends neatly from the lower left hand corner to the top right corner of the **Results-1** window. The **USER SPACE** command has rescaled the **Results-1** window from 5000 x 5000 units to 400 x 400 units.

The next concept that must be introduced is that of a **virtual screen**. You can think of a virtual screen as being a two dimensional world over which various windows can be placed. Because your computer has a large amount of memory, up to four virtual screens can be handled at one time, although there are limitations to the feasibility of running four virtual screens at once.

A map of the coordinate space in which a virtual screen sits is shown in Figure 6.2. The small filled square shown in this Figure was drawn using the graphics command:

`BOX 2000;3500,1000,1000 FILL WITH 6`

As you can see, only part of the box is visible within the output window on the screen. So far, we have only looked at output that was directly visible within the output window. Windows are not static entities but can be expanded or moved as required either by using the mouse or directly from BASIC 2 by using the commands **WINDOW SCROLL**, **WINDOW SIZE**, **WINDOW FULL** and **WINDOW PLACE**. The placement and size of the window is specified in pixels and not user coordinates because they deal with the physical placement of the window on the screen, rather like tacking a notice to a

noticeboard. **WINDOW SCROLL** on the other hand controls the scrolling of the window across the virtual screen, and is more useful from the graphics point of view. **WINDOW SIZE** and **WINDOW FULL** will allow you to 'cover or uncover' more of the virtual screen, however.

Here is a short program that shows you how the physical and virtual screens are related. First, the position and size of the **Results-1** window are set, and then a circle is drawn on the virtual screen to the left of the window. The window is then scrolled leftwards towards the circle, and stops moving when the circle is centred in the window. Note that no movement of the window 'frame' occurs across the screen.

```
REM program to demonstrate window scrolling
CLS
WINDOW  OPEN
WINDOW  PLACE  240;50
'
CIRCLE  1000;4000,500
FOR i = 240 TO 2000 STEP 200
WINDOW  SCROLL  i;50
NEXT i
```

If you are drawing on the screen you must not let the coordinates of the object or text lie outside the coordinates of the screen. For most cases you can assume the screen to be about 5000 x 5000 units: the default user space of the **Results-1** window as it first appears when you enter BASIC 2.

The functions **XVIRTUAL** and **YVIRTUAL** allow you to find the size of the screen you are working with, and hence to get a better idea of the scale to use for your graphics. They are used to return the required values, in the following way.

```
PRINT "width of screen = ",XVIRTUAL
PRINT "height of screen = ",YVIRTUAL
```

virtual screen

(0;5000)　　　　　　　window　　　　　　(8000;5000)

(0;0)　　　　　　　　　　　　　　　　　　(8000,0)
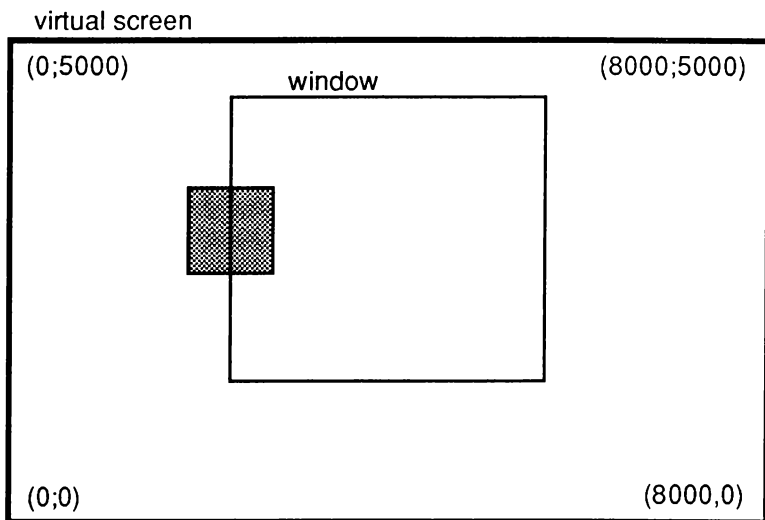
Figure 6.2 Map of the virtual screen

An important point to note about user coordinates is that they 'correct' for any asymmetry along the X and Y axes of the screen. We have already seen that a typical screen resolution is 640 x 200 pixels. If the screen were in 1:1 proportion along both axes, it should therefore be 3.2 x as wide as it is high. It is unlikely that your screen looks like this!
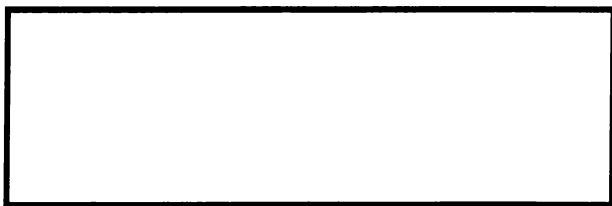
Figure 6.3 A truly symmetrical 640 x 200 pixel screen.

BASIC 2 user coordinates take away the drudgery of calculating aspect ratios, as they always correct for the display being used. The results of the functions **XVIRTUAL** and **YVIRTUAL** therefore represent a true 'map' of the available screen area as it would appear on normal graph paper, and you can assume that a length of, say, 100 units along the X axis is the same physical length as 100 units down the Y axis.

Drawing lines and shapes with BASIC 2

Let us first limber up with some simple BASIC 2 graphics instructions to be outputted to the **Results-1** window. Look at the output in Figure 6.4 and study the following program that generated it.

```
REM  multicolour overlapping circle program
CLS
    WINDOW  #1  OPEN
    WINDOW  #1  FULL
    col_value = 1
    n = -1
    y_coord = 500
    FOR x_coord = 500 TO 8000 STEP 200
       n = n + 1
       IF n = 16 THEN n = 0
       y_coord = y_coord + 100
       CIRCLE  x_coord;y_coord,500 COLOUR n FILL
    NEXT  x_coord
WHILE  BUTTON  =  -1:WEND

END
```

The command **CIRCLE** was used to draw the circles. Like **BOX** that you met earlier in this chapter, **CIRCLE** is one of the armoury of graphics commands available to you directly within BASIC 2. These commands will allow you to draw points, straight lines, rectangles, polygons, circles, arcs of circles and ellipses. Different line thicknesses and colours may be used. Coupled with the options for filling the two dimensional objects you create with colours and patterns, these commands make BASIC 2 very powerful for handling graphics.

The full list of graphics drawing commands is as follows:

BOX
CIRCLE
ELLIPSE
PIE
ELLIPTICAL PIE
LINE
PLOT

Additional parameters are used to define the stream number, size and/or shape of the graphics object, be it a line, circle or more complex shape. User coordinates are used to measure in all the graphics drawing commands.
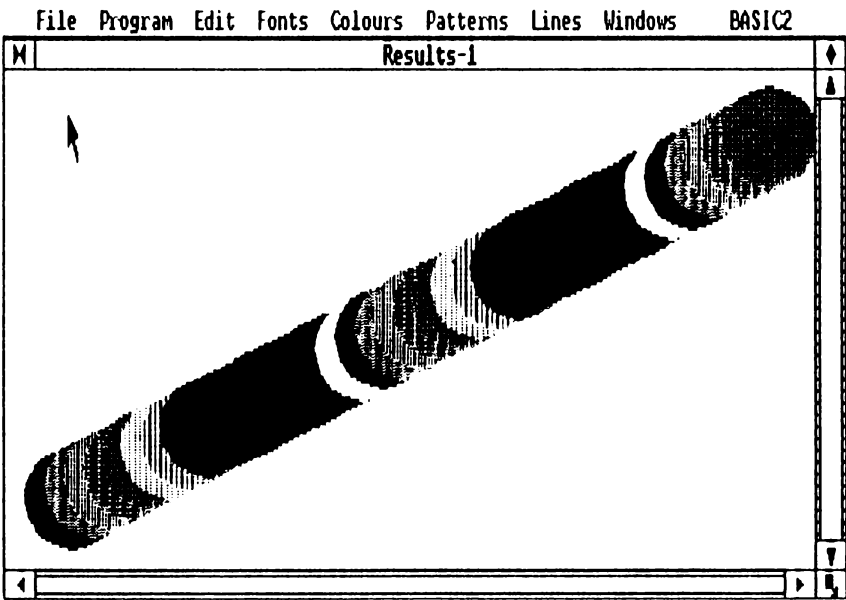
Figure 6.4 Overlapping circles

The **SHAPE** command is a useful way of drawing a polygonal (ie many sided) shape in a single command. A list of the x;y coordinates making up the points defining the shape is given, and straight lines are then sequentially drawn between the points. The shape is always 'closed off' by drawing from the last defined point to the starting point. Here is an example of the use of the **SHAPE** command.

| H | Dialogue | ♦ | | Results-1 |

```
Ready
shape 1000;1000,1500;1500,
2000;500,1000,1000 colour
7 fill with 4
Ready




Edit
```

Edit
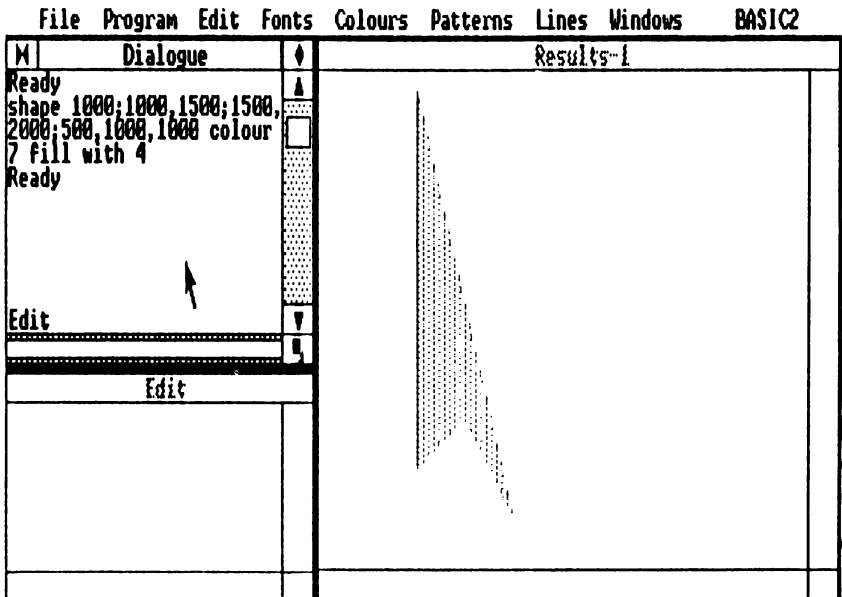
Figure 6.5 The SHAPE command, used here in direct mode from the **Dialogue**
window

SHAPE can be especially useful in two ways. Firstly, if you want to draw the
same shape at various points on the screen, you can put the relevent SHAPE
command into a subroutine, like this.

```
REM program demonstrating multiple use of the SHAPE
command

CLS
WINDOW OPEN
WINDOW FULL
WINDOW TITLE "multiple shape demonstration"
    FOR i = 500 TO 7000 STEP 500
    FOR j = 500 TO 4000 STEP 700
    GOSUB image_draw
    NEXT j
    NEXT i
WHILE BUTTON = -1:WEND
```

```
LABEL image_draw
SHAPE i;j,i+400;j,i+200;j-400 COLOUR 1 FILL
SHAPE i+200;j+200,i+400;j-200,i;j-200 COLOUR 5 FILL
RETURN
```
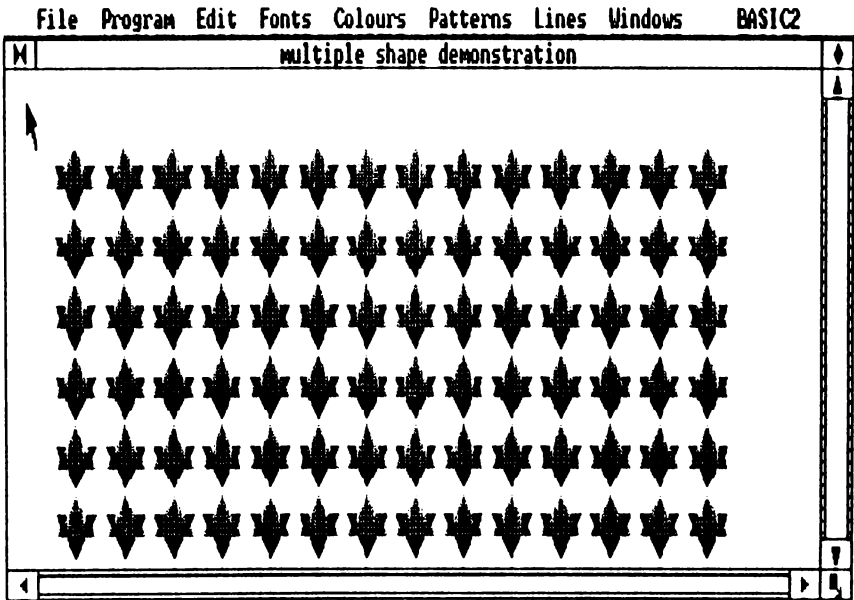
Figure 6.6 Multiple shapes drawn using SHAPE commands inside a subroutine

You can obtain the full syntax of the various graphics commands from Appendix
1 and the *BASIC 2 User Guide*. It is often easier to see how to use commands
from specific examples, and here are two programs designed to demonstrate the
graphics drawing facilities in BASIC 2. The first program draws a graph from
data supplied within the program (this could be easily amended to read in data
from a disk file if required). The second program draws a house and garden.

```
REM graph generation program
CLS
WINDOW  OPEN
WINDOW  FULL
WINDOW  SCROLL 0;0
WINDOW  PLACE 260;50
```

```
'define number of points on the graph
   DATA 5
'define data for x and y axes
   DATA 50,50,100,100,200,200,300,300,400,400
'define title data
   DATA "x distribution"
   DATA "y distribution"
'define maximum x and y values
   DATA 400,400
'
   READ npts
DIM x(npts),y(npts)
FOR i = 1 TO npts
   READ x(i),y(i)
NEXT i
   READ n$,m$
   READ mx,my
'draw axes
LINE 2000;3800,2000;800
LINE 2000;800,5000;800
'now put in the scale marks
FOR i = 1 TO 11
   LINE 1900;(i*300)+500,2000;(i*300)+500
   LINE (i*300)+1700;700,(i*300)+1700;800
NEXT i
'now label the axes
'position the x axis label first
   ax = (3500-(EXTENT(n$)/2))
   MOVE ax;500
   PRINT n$
   ay = EXTENT(m$)/2
   MOVE 1200;ay
'note y text is rotated by 90 degrees
   PRINT ANGLE (90) m$
'now plot the points
FOR i = 1 TO npts
    MOVE 1940 + (3000 * (x(i)/mx));860 + (3000 *
(y(i)/my))
   PLOT MARKER 3
NEXT i
```

```
WHILE  BUTTON=-1:WEND

END
```

Patterns Lines Windows   BASIC2
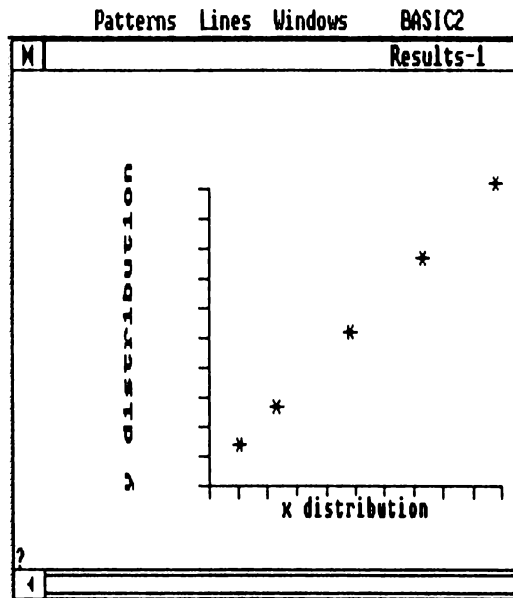
Figure 6.7 Output from GRAPH program

```
REM   house and garden program
WINDOW PLACE 300,50
WINDOW OPEN
CLS
DIM x(25),y(25): ' x,y coordinate arrays for points
    'draw horizontal line for horizon
    LINE x(1);y(1),x(2);y(2)
    'draw tree foliage
    CIRCLE x(11);y(11),1000 FILL WITH 8 COLOUR 3
    'draw apples on tree
    FOR i = 2 TO 10
        CIRCLE x(i);y(i),40 FILL WITH 8 COLOUR 2
    NEXT i
    'draw house front
```

85

```
BOX x(12);y(12),2000,1000 FILL WITH 8 COLOUR 6
'draw windows
BOX x(13);y(13),400,400 FILL WITH 8 COLOUR 6
BOX X(14);Y(14),400,400 FILL WITH 8 COLOUR 6
'draw tree trunk
BOX x(15);y(15),100,800 FILL WITH 8 COLOUR 14
'draw roof
SHAPE x(16);y(16),x(17);y(17),x(18);y(18),x(19);
y(19) FILL WITH 20 COLOUR 2
'draw path
SHAPE x(20);y(20),x(21);y(21),x(22);y(22),x(23);
y(23) FILL WITH 8 COLOUR 8
'draw chimney
BOX x(25);y(25),350,600 FILL WITH 8 COLOUR 4
'coordinate data
    DATA 0,2000,5000,2000
    DATA 3600,4000,3700,3100,3800,3200,4500,4200,
         3700,3900
    DATA 3800,4000,3900,4100,3900,3600,3950,3520
    DATA 500,2000,700,2500,1800,2500,3900,2000
    DATA 500,3000,2500,3000,2000,3500,1000,3500
    DATA 1540,2000,1600,2000,2000,500,1000,500,
         1250,2000
    DATA 1250,2040
```

Both the GRAPH program and the HOUSE program are very simple and their structures require little explanation. You should be able to work through the programs using the comments within the programs to help you.

Graphics modes

There are four different modes with which graphics objects can be written to the screen. The default is **replace mode**, in which the new object replaces anything under it at that screen location. This means that anything drawn in *foreground* or *background* colour will overwrite the screen. The second mode prints only the foreground colour, and is called **transparent mode**. These are the most common modes that you will use most of the time, and are illustrated in Figure 6.9. One of the less common modes is **XOR mode**, in which the foreground colour only is drawn after **XOR**ing with the existing colour at that location. **Reverse transparent mode** is used to give a negative effect.
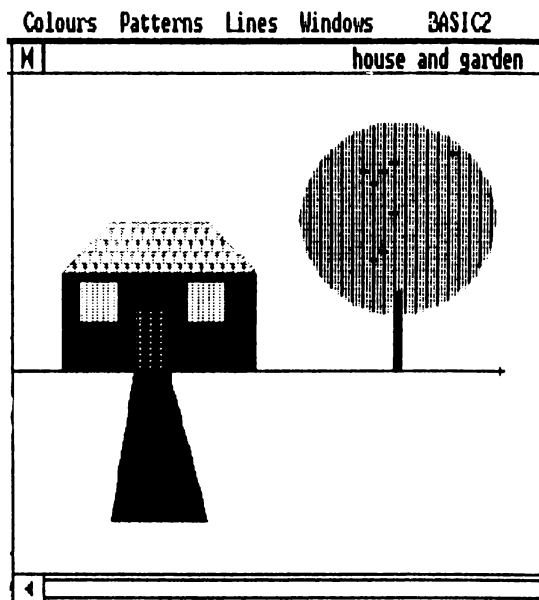
Figure 6.8 Output from HOUSE program

The following program shows the effect of the different write modes when plotting graphics.

```
REM graphics write mode program
CLS
WINDOW OPEN
WINDOW PLACE 250;50
WINDOW SCROLL 0;0
   BOX 500;500,1500,1500 MODE 1 COLOUR 1 FILL WITH
3
   BOX 1000;1000,1000,1000 MODE 1 COLOUR 2 FILL
WITH 4
MOVE 200;280
PRINT "mode 1 default"
MOVE 500;2500
   BOX 500;2500,1500,1500 MODE 2 COLOUR 1 FILL
```

```
WITH 3
    BOX 1000;3000,1000,1000 MODE 2 COLOUR 2 FILL
WITH 4
MOVE 200;2300
PRINT "mode 2 transparent"
MOVE 2000;3000
    BOX 2500;500,1500,1500 MODE 3 COLOUR 1 FILL
WITH 3
    BOX 3000;1000,1000,1000 MODE 3 COLOUR 2 FILL WITH
4
MOVE 2300;280
PRINT "mode 3 XOR"
    BOX 2500;2500,1500,1500 MODE 4 COLOUR 1 FILL
WITH 3
    BOX 3000;3000,1000,1000 MODE 4 COLOUR 2 FILL
WITH 4
MOVE 2300;2300
PRINT "mode 4 reverse transparent"
WHILE BUTTON=-1:WEND
```
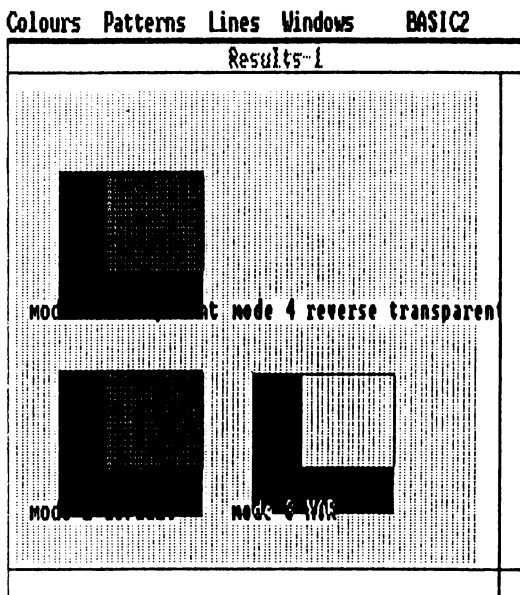

Figure 6.9 Different graphics write modes

Drawing circular objects

Although several of the programs already described in this chapter have drawn circles, BASIC 2 has a very flexible set of circle drawing commands that are worthy of further study. The first thing to note is that there are two options for specifying angles in BASIC 2. The default is *radians*, but you can specify angles to be in degrees, either from the **Program** menu, or alternatively by using the `OPTION DEGREES` command.

Angles are measured counterclockwise, with 0 degrees as the orientation towards the right hand side of the screen. You can draw full circles or arcs of circles (both of which may be outlines or filled areas). The same options are available for ellipses. A word of advice. BASIC 2 has a rather bewildering duplicity of commands for drawing parts of circles. In order to draw an arc, you can use:

```
CIRCLE x;y, radius PART start angle, end angle
PIE x;y, radius, start angle, end angle
```

```
REM pie chart program
CLS
WINDOW    PLACE 250;50
WINDOW    OPEN
PRINT "welcome to the pie chart program"
INPUT "main title";m$
INPUT "how many segments for display";number
DIM segs (number+1),h$ (number), cum
(number+1),mark(number)
total = 0
cangle = 0
'now input segment data
'note that your segment titles should be of sensible
'length - up to 5 or 6 characters maximum
   FOR i = 1 TO number
     INPUT "input this segment title";h$(i)
     INPUT "input segment value";segs (i)
     total = total + segs (i)
   NEXT i
   segs(number+1)=segs(1)
```

89

```
'set angles for each segment
   FOR i = 1 TO number
      cangle = cangle + ((segs(i)/total)*(2*PI))
      mark (i) = cangle - (((segs(i)/2)/total)*(2*PI))
      cum(i) = cangle
   NEXT i
   cum(number+1)=cum(1)
'now draw the pie using this data
CLS
WINDOW #1 TITLE m$
radius = 1500
xcentre = 2500
ycentre = 2500
   FOR i = 1 TO number
      PIE xcentre;ycentre,radius,cum(i),cum(i+1)
COLOUR i FILL
   NEXT i
GOSUB text_place
WHILE BUTTON=-1:WEND

END

LABEL text_place
'this routine works out where each text label should
go
'and then inserts the label at the correct point
   FOR i = 1 TO number
      text_x_pos = xcentre+(radius/2)*COS(mark(i))
      text_y_pos = ycentre+(radius/2)*SIN(mark(i))
      displace = EXTENT (h$(i))
      IF text_x_pos>xcentre THEN displace =
displace-200
      MOVE text_x_pos-displace;text_y_pos
      PRINT h$(i);
   NEXT i
RETURN
```
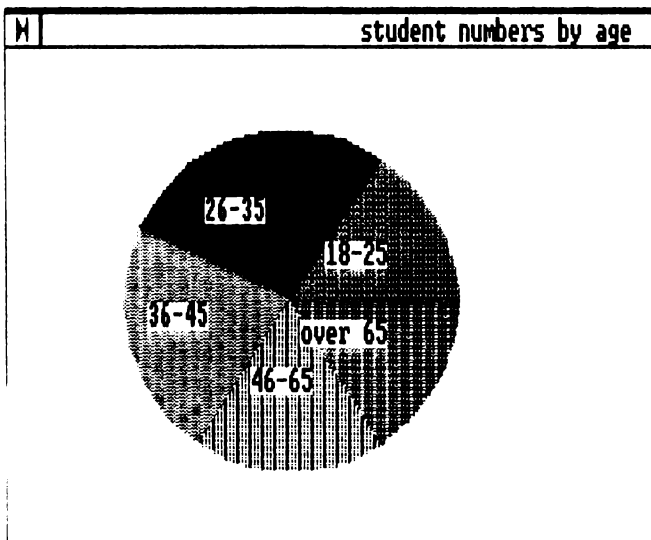
Figure 6.10 Output from the PIE program

6.3 Animation in BASIC 2

One of the most common forms of computer graphics is graphic animation. There is often a problem in using BASIC to handle animation, as many dialects of BASIC are slow in operation, due to the fact that BASIC is usually an *interpreted* rather than a *compiled* language. In an interpreted language, the program in effect has to go through a compilation step each time it is run, so introducing a time overhead.

The more efficient the interpreter, the faster the version of BASIC. Locomotive Software have a history of producing fast BASIC interpreters, and BASIC 2 is no slouch! You can therefore use BASIC 2 to do many things that would be impossible for the user of a slower BASIC (Microsoft BASIC on IBM compatible computers is an example of a slow BASIC dialect). There is a drawback however, and this is that BASIC 2 maintains a list of the items to be drawn in each window. When the list gets full up, it is periodically 'cleared'. You will find that BASIC 2 performs animations at hyperspeed for a few iterations, but when the clearing operations start it slows to just plain fast!

Animation is the art of simulating movement. Before the days of computers,

animation was the province of the artist, and of course the most flexible animation is still done manually. Computers are now capable of sophisticated animation effects, particularly when the animated images are describable in simple mathematical terms, for example rotation of a polygonal three dimensional shape.

To generate complex images capable of movement in real time you will need more than an IBM compatible PC, and even the Amstrad PC1512 with its fast 8086 processor would be pushed, even in machine code, to animate images made up of more than a few tens of lines. Although this may sound a dismissive statement, there are still interesting things that can be done in BASIC 2, and in this section we will look at a few elementary animation techniques.

We will start with the simplest example, that both gives the flavour of animation and, incidentaly, provides an illustration of the use of the XOR graphics mode described earlier in this chapter. The following program draws triangles at random over the screen. Before drawing each triangle, the previous triangle is erased by drawing over it in XOR mode. The effect is therefore that of a single triangle rotating around in three dimensional space. A circle drifts across the screen from left to right at each triangle redraw step. Any reader upgrading to an Amstrad PC1512 from the Amstrad 6128 might like to note that a version of this program appeared in *Amstrad Graphics: The Advanced User Guide* (also available from Sigma Press). The poor 6128 could however only animate movement of a single straight line in the time the PC1512 takes to animate both triangle **and** circle!

```
REM line moving program
WINDOW  OPEN
WINDOW  FULL
WINDOW TITLE "line and circle animation demo"
CLS
'set circle start coordinates
n = 500
m = 2000
LABEL jump
'first calculate end points of line at random
x1 = INT (RND*5000)+200
y1 = INT (RND*3000)+200
x2 = INT (RND*5000)+200
y2 = INT (RND*3000)+200
x3 = (x1+x2)/2
```

```
y3 = (y1+y2)/2
'draw the circle
CIRCLE n;m,300 MODE 3 COLOUR 2
'draw the triangle
SHAPE x1;y1,x2;y2,x3;y3 MODE 3 COLOUR 6
'now wipe triangle out
SHAPE x1;y1,x2;y2,x3;y3 MODE 3 COLOUR 6
'now wipe out the circle
CIRCLE n;m,300 MODE 3 COLOUR 2
n = n + 50
IF BUTTON = -1 THEN GOTO jump

END
```

Besides animation of lines and other geometric shapes, you can animate characters on a text screen. A similar technique is used to the line drawing situation, but in this case you simply print a space over an existing character to erase it. The program moves a character around the screen in a square, changing to the next character in sequence at the end of each cycle.

```
REM character animation program
'flips a character around the screen using the LOCATE
command
CLS
WINDOW  OPEN
WINDOW  FULL
WINDOW TITLE "animation example"
n = 48:'set first character code
LABEL jump
n = n+1:'increment character code counter
   FOR j = 3 TO 18
      LOCATE j;5:PRINT " " + CHR$(n);
   NEXT j
   FOR i = 5 TO 17
      LOCATE 19,i:PRINT" ";
      LOCATE 19;i+1:PRINT CHR$(n);
   NEXT i
   FOR j = 19 TO 3 STEP -1
      LOCATE j;18:PRINT CHR$(n) + " ";
   NEXT j
```

```
FOR i = 18 TO 6 STEP -1
    LOCATE 3;i:PRINT " ";
    LOCATE 3;i-1:PRINT CHR$(n);
NEXT i
IF BUTTON = -1 THEN GOTO jump

END
```

6.4 Turtle graphics

If you want to draw sophisticated graphics, or even if you only need a few squares and circles, you will probably be best off using the graphics commands described so far in this chapter. In fact, any reader of this book will not need to routinely use turtle graphics commands. Why are they there at all? The answer to this question is that turtle graphics are very useful for introducing concepts of computer programming to absolute beginners, especially children. Turtle graphics are often used in school for this purpose, and the turtle graphics commands in BASIC 2 are essentially a subset of those in LOGO, the computer language devised by Seymour Papert at the Massachussetts Institute of Technology some years ago. Turtle graphics are also used in PILOT, another language used to teach children.

A turtle-moving session

The first thing that you need to use turtle graphics with BASIC 2 is a turtle. You can in fact use the existing cursor for this purpose, but the turtle looks more interesting, and we will assume that you want to get into the spirit of turtle graphics! To get the turtle you use the cursor changing command:

```
GRAPHICS CURSOR 3
```

Try it?

The idea of turtle graphics is that the programmer is in control of the turtle, which itself represents the 'pen' which is doing the drawing. The pen can be 'up' in which case it will not draw anything as the turtle is moved, or 'down', causing a line to be traced along the path of the turtle as it moves. The following short program invokes the turtle, places it in the centre of the **Results-1** window, and draws a triangle from that point.

```
REM TURTLE GRAPHICS TRIANGLE PROGRAM
    GRAPHICS  CURSOR 3
    LEFT 45
    FORWARD  50
    LEFT 90
    FORWARD  50
    LEFT 45
    FORWARD  50
```

You can try drawing a number of simple shapes in this way.

The power of turtle graphics is that it offers an intuitive feel for what otherwise might seem an exercise in pure mathematics. Here are two more short programs that draw simple shapes. The first draws a spiral, by 'walking round' in a circle, with each step getting shorter and shorter.

```
REM turtle spiral program
SCREEN  GRAPHICS
WINDOW  OPEN
WINDOW  FULL
CLS
GRAPHICS  #1 CURSOR  3
OPTION  DEGREES
head = 1
line_length = 200
number = 100: 'set reiteration number
increment = 20:'set angular increment
MOVE 2000;2500
FOR i = 1 TO number
    head = head + increment
    line_length = line_length - 1
    IF head > 360 THEN head = 1
    POINT head
    FORWARD line_length
NEXT i
WHILE  BUTTON=-1:WEND
```
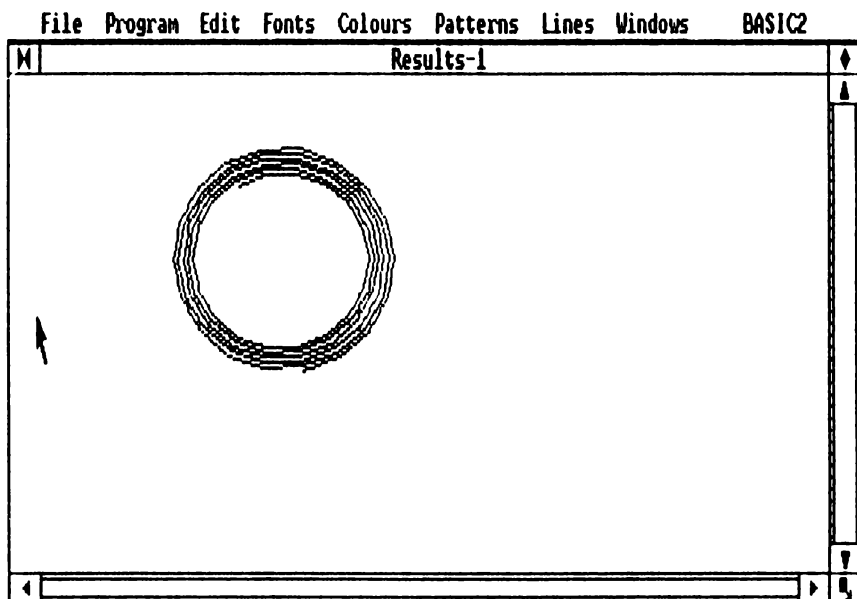
Figure 6.11 Output from SPIRAL program

The next short program is a variation of the spiral program, but in this case a routine is called at each increment to draw a square. A spirograph type pattern results.

```
REM turtle squares program
SCREEN  GRAPHICS
WINDOW  OPEN
WINDOW  FULL
CLS
GRAPHICS  #1  CURSOR  3
OPTION  DEGREES
head = 1
line_length = 200
increment = 20:'set angular increment
number = 20
MOVE 5000;2500
FOR i = 1 TO number
    head = head + increment
    line_length = line_length - 1
```

```
    IF head > 360 THEN head = 1
    POINT head
    MOVE FORWARD 10
    GOSUB square
NEXT i
    END
  LABEL square
    FD 1000: LT 90
    FD 1000: LT 90
    FD 1000: LT 90
    FD 1000: LT 90
RETURN
```
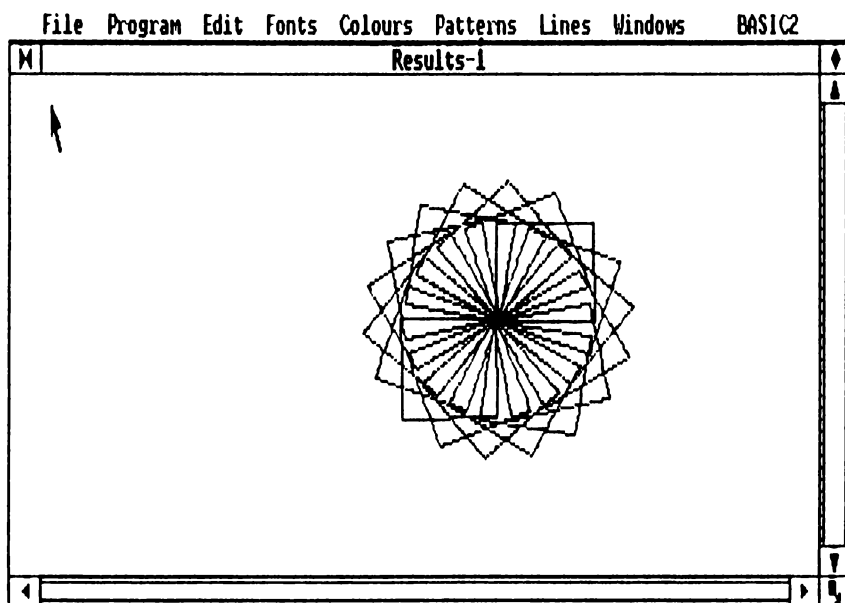

Figure 6.12 Output from SQUARES program

6.5   Business graphics

We have already looked at a simple graph program earlier in this chapter. Although this book is intended to introduce the concepts of BASIC 2 rather than

the fundamentals of computer graphics, it would seem unfair to leave the topic of graphics without giving an example of a typical business graph: a yearly sales report. Besides this graph we will look at a very simple modification of the technique used that will allow a histogram (or bar chart) to be drawn with the same data.

```
REM chart program
'draws a labelled graph for a twelve month period
'for example sales figures, currency fluctuations etc
'the data is at the end of the program, but the
'program could be easily modified for interactive
'input
CLS
WINDOW #1 OPEN
WINDOW #1 FULL
'set up data arrays
DIM xp(12),yp(12)
'input label information
    INPUT "main title max 80 characters";t$
    INPUT "side title max 10 characters";s$
    INPUT "side (sub) title max 10 characters";s1$
'now calculate title positions
t1 = LEN(t$)
t2 = LEN(s$)
t3 = LEN(s1$)
xt = (t1/2)+10
xs = 5-(t2/2)
xs1 = 5-(t3/2)
CLS
LOCATE xt;2:PRINT t$
LOCATE xs;10:PRINT s$
LOCATE xs1;11:PRINT s1$
GOSUB month_legends
'now draw axes
LINE 1450;3700,1450;1100
LINE 1450;1100,5500;1100
'make y axis gradations
    FOR y = 3670 TO 1170 STEP -20
        LINE 1420;y,1470;y
    NEXT y
'now insert y axis scale values
```

```
      READ y_value
      MOVE 950;3600:PRINT y_value
      MOVE 950;2300:PRINT y_value/2
'now plot the graph
xf = 1310
FOR i = 1 TO 12
      READ value
      value = ((value/y_value)*2500)+1320
      xf = xf + 325
      IF i = 1 THEN x2=xf:ys=value:y2=value
      x1=x2:y1=y2
      x2=xf:y2=value
      LINE x1;y1,x2;y2
      NEXT i
WHILE BUTTON = -1:WEND

END
LABEL month_legends
LOCATE 1;17
PRINT TAB(16);"|  |  |  |  |  |  |  |  |  |  |  |"
PRINT TAB(16);"J  F  M  A  M  J  J  A  S  O  N  D"
PRINT TAB(16);"A  E  A  P  A  U  U  U  E  C  O  E"
PRINT TAB(16);"N  B  R  R  Y  N  L  G  P  T  V  C"
RETURN

DATA 10
DATA 1.6,1.8,2.5,2.7,1.1,3.6,4.6,5.9,7.2,8.1,7.1,9.3
```

In order to modify this program to draw a bar chart rather than a graph, a very simple modification can be used. All you have to do is to remove the line:

```
LINE x1;y1,x2;y2
```

and replace it with:

```
BOX xf;1100,160,y2-1100 FILL WITH 4
```

The fill pattern can be varied as required, and the constant 160 is the width of each rectangle: this can be changed if wider or narrower 'bars' are required. The output from both the graph and bar chart variations of the program are shown on the next page.

Results-1

Sales Summary for 1986

Modems
(x 1000)

J F M A M J J A S O N D
A E A P A U U U E C O E
N B R R Y N L G P T V C

Results-1

Student enrolments

student
number

J F M A M J J A S O N D
A E A P A U U U E C O E
N B R R Y N L G P T V C
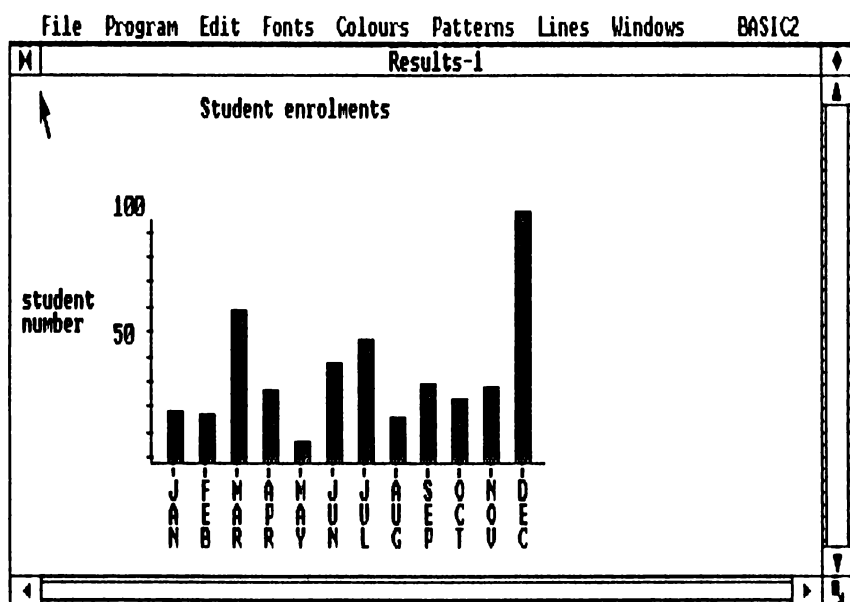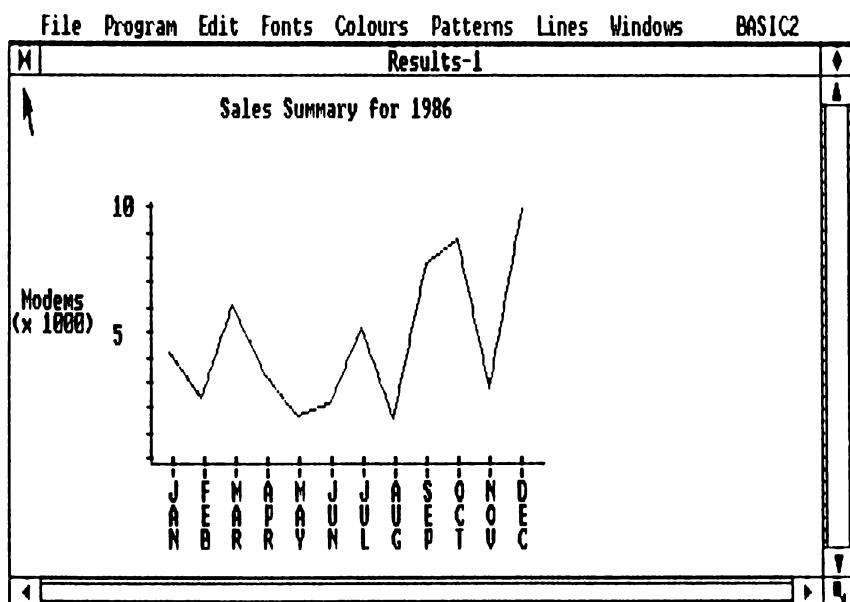
Figure 6.13-6.14 Output from program CHART (top) and program BAR (bottom)

100

BASIC 2 Text Output

7.1 Displaying character output

There are two ways of displaying characters in BASIC 2, depending on whether you are using a text or a graphics screen. In this section we will look at the more limited case applicable to a text screen. If you have used a computer before, this kind of simple character output will be familiar to you. On most computers, text is output to the screen as if the screen consisted of a number of regularly spaced boxes, each of which can hold a single character and has a row and a column reference, like this:
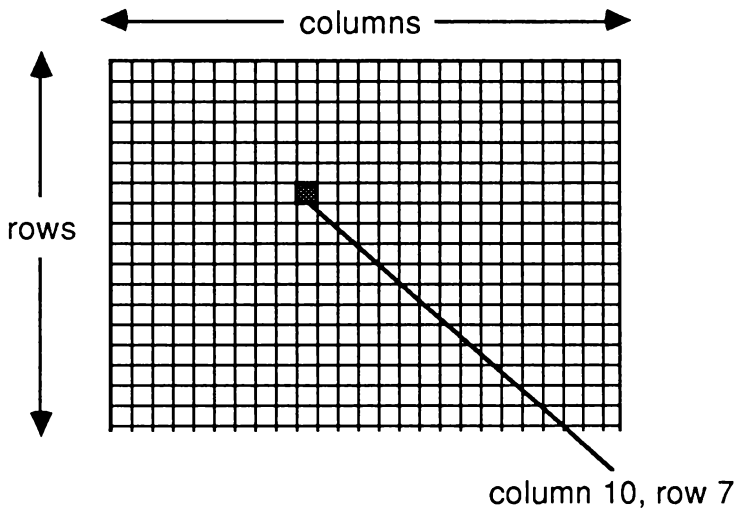
Figure 7.1 Text placement on the screen

To move the cursor to a particular place on the screen, the command **LOCATE** is used, so to move to the square highlighted in Figure 7.1, the command:

```
LOCATE  10;7
```

would be used, with the general format **LOCATE** `column;row`. The top left hand corner of the current window is defined as `1;1`.

Try the following short program to see how the **LOCATE** command works.

```
REM PROGRAM DEMONSTRATING LOCATE COMMAND

   SCREEN  TEXT
   WINDOW  OPEN
   WINDOW  FULL
      j=1
   FOR   i=2 TO 18
      j=j+1
      LOCATE  i;j
      PRINT "basic 2"
      LOCATE  20-i;j
      PRINT "basic 2"
   NEXT  i
   WHILE  BUTTON  =  -1:WEND
   STOP
   END
```

The output from this program is shown on the next page.

Sometimes it is not necessary to exert complete control over the vertical and horizontal cursor position: you may wish to output data to a series of tab positions along a line for example. To do this, the command **TAB** `(column)` can be used. Say we want to print a numeric total:

PRINT `"Total = ";` **TAB(12)** `number`

where number is a variable.

You see that **TAB** makes the cursor move along the required number of columns.

```
◆SCREEN #1 TEXT                         H|                          Results-1
◆WINDOW #1 OPEN          basic 2        basic 2
◆WINDOW #1 PLACE 250;50    basic 2        basic 2
◆j=1                        basic 2      basic 2
◆FOR i = 2 TO 18             basic 2    basic 2
◆    j=j+1                     basic 2 basic 2
◆    LOCATE i;j                 basic basic 2
◆    PRINT "basic 2"            basibasic 2
◆    LOCATE 20-i;j              babasic 2
◆    PRINT "basic 2"            basic 2
◆NEXT i                        basic 2 2
◆WHILE BUTTON = -1:WEND      basic 2ic 2
◆STOP                       basic 2asic 2
◆END                       basic 2 basic 2
◆                         basic 2    basic 2
◆                        basic 2       basic 2
          ▶            basic 2          basic 2
                      basic 2            basic 2

                               ◀
```

Figure 7.2 Text output using the LOCATE command

There is only one problem with the use of the **TAB** command, and this is that the numbers are always aligned by their first digit rather than by the decimal point. If you are a businessman, you do not want to have your figures printed in the following form.

```
100.00
5.00
25000.00
12.00
34.45
623.45
```

To correct matters, you need to use the BASIC 2 command **USING** in conjunction with a **template**. The template specifies how the number is to be printed, using the decimal point as a constant position across the screen or printer. The commands:

PRINT USING "######.##" 23.65

```
PRINT USING "######.##" 111234.42
```

will neatly output in the following form:

```
   23.65

  111234.65
```

You can also split the whole part of the number into groups of digits separated by commas - say if you wish to use the thousands notation like this:

```
 111,234.65
```

The corresponding **USING** command would be

```
PRINT USING "###,###.##" 111234.42
```

The following short program could be used to output a table of numbers from a program to the screen. This program is pretty dumb, but it shows you how the various derivatives of the humble **PRINT** statement can be coupled to produce output pleasing to the eye.

```
REM PROGRAM PRINTING TABLES OF NUMBERS TO SCREEN
   REM HERE IS THE DATA
   DATA 1,234.5,65.43,93
   DATA 2,12.64,34.21,45
   DATA 3,95.1,62,31.2,55
   DATA 4,34.3,23.4,12,22

   REM PRINT HEADINGS
   PRINT "SALES CODE     INCOME       COSTS
SALES"
   PRINT
   REM NOW DO 4 LINES OF OUTPUT
   FOR I=1 TO 4
     READ A,B,C,D
     PRINT USING "###.##       "a;
     PRINT USING "###.##       "b;
     PRINT USING "###.##       "c;
     PRINT USING "###.##       "d
```

NEXT i

END

H		Results-1					
File	Program	Edit	Fonts	Colours	Patterns	Lines	Windows	BASIC2

```
sales code    income         costs         sales
    1.00       234.50         65.43         93.00
    2.00        12.64         34.21         45.00
    3.00        95.10         62.00         31.20
   55.00         4.00         34.30         23.40
```

Figure 7.3 Tabular output

7.2   The text screen

We have already seen the use of the **SCREEN** command for setting up a graphics screen in BASIC 2, and the same command, with a little modification, is also used to set up the parameters for defining a text screen. The general form for the command is

SCREEN [#stream] **TEXT** dimensions [**MINIMUM** w,h]
[**MAXIMUM** w,h] [**UNIT** w,h] [**INFORMATION** switch]

This looks rather complicated, but like most other complex commands within BASIC 2, you need only specify the parts that you need.

dimensions may be either:

105

or

`columns [`**`FIXED`**`], rows [`**`FIXED`**`]`

You will have noticed that **`FLEXIBLE`** is not available for the graphics equivalent of the **`SCREEN`** command. The reason for this is that **`FLEXIBLE`** forces the width of the virtual screen to be the same as its window. If you want to see this effect in action, try resizing the **Edit** window with some text in it. You will see that the text rejigs itself to fit the window width. **`FIXED`** on the other hand sets the virtual screen to a set number of rows and columns, so that a window smaller than the virtual screen can be defined if required. The largest number of characters allowed on a virtual screen is 2048.

On the Amstrad PC1512 you will find that the maximum usable text screen dimensions are about 77 columns by 21 rows. You can of course get more text on the screen by defining a small font size in graphics mode (see the next section for details of text and graphics).

`MINIMUM` and **`MAXIMUM`** set the minimum and maximum width and height for the window (the units are columns and lines respectively).

The units by which the window size may be altered is specified by the **`UNITS`** option. The default is a single character.

As with the **`SCREEN GRAPHICS`** command, the window and virtual screen are cleared, and the origin, text and graphics colours and other values are reset to the default.

7.3 Text and graphics

Text on the graphics screen offers a variety of options to enhance your output. These options are the ability to display different text fonts, to change text sizes, to angle text, and to write the text in different modes so that it can mix with the existing text and graphics on the screen in different ways.

Text fonts

The number of fonts available on your computer system will be seen by pulling down the **Font** menu at the top of your BASIC 2 screen. These fonts are numbered upwards from 1, the standard system font. The names of the other fonts can also be discovered by using the function **FONT$**.

```
fontname$ = FONT$([#stream,] fontnumber)
```

Given the number of a particular font, you can change output in a particular stream to print in that font. For example, to print all text in the default output window (**Results-1**) in font 2

SET FONT   2

and to print only specified items in a particular font:

PRINT FONT (2); "this is in font 2"

Font size

The sizes of text are measured in the printer's measure called points. A point is 1/72 of an inch, and as with the fonts, you can find out the sizes available from the **Fonts** menu, or by using a special function. The function **POINTSIZE** will unfortunately not tell you all the pointsizes available for a given font, but it returns instead the nearest available point size to the one specified in the function. Besides the point sizes in the system, you can also use a point size that is *exactly twice* that point size.

In order to print out all the point sizes available in a given font you will have to use a short BASIC 2 program like this:

```
REM PROGRAM TO GIVE FONTSIZES
INPUT "fontnumber"; fontnumber
   temp=256
FOR i = 1 TO 256
   old_size = temp
   temp = POINTSIZE(fontnumber, temp - 1)
   IF temp <> old_size THEN PRINT temp
NEXT i
```

Text Positioning

In order to get text to a specific position on a graphics screen you use the command **MOVE**

MOVE [#STREAM] x;y

or to move the graphics cursor to the point 100;50 on the **Results-1** screen:

MOVE 100;50

The **MOVE** command is a very accurate way of placing text in a graphics screen. You can also use the standard text screen command **LOCATE** to position graphics screen text, and this will result in text being placed using the row and column positions of the standard system font character cell size. The dimensions of this cell can be found by using the functions **XCELL** and **YCELL**. You can use the information returned by these functions to work out the relationship between positions in user coordinates and positions in row/column units. An example of this is shown in Figure 7.4 on the next page.

You can see that the cursor position is not at one of the corners of a text cell. The position of the cursor is given by the functions **POS, VPOS, XPOS** and **YPOS.** The first two functions give the position in row and column values, the latter functions give the cursor position in user coordinates.

BASIC 2 contains a useful function called **EXTENT** for working out the horizontal length of a text string. This avoids the tedious calculation of text widths based on multiplications of text-cell sizes, especially when different point sizes are to be displayed.

width = **EXTENT** ("how wide is the Ganges?")

or in the more general form:

width =
EXTENT([#stream,][print-functions,]stringexpression)

Besides calculating the horizontal extent of a text string, the vertical height is also of importance. Text of varying heights will always be printed on the same base line, as shown on the next page.
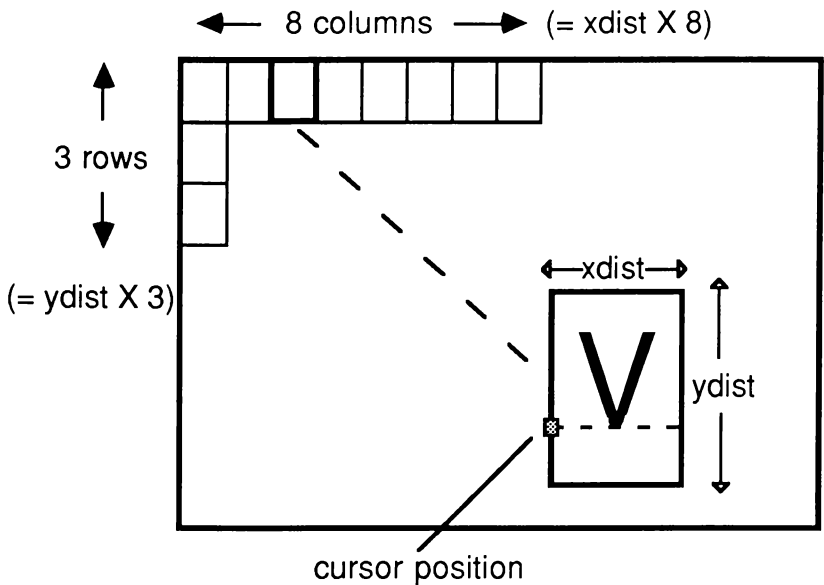
Figure 7.4 Relationship of text cell size and the screen measured in user coordinates
The size of each cell in user coordinates is given here by the variables xdist and ydist. The
lengths of the row and column of cells given here are calculated from these variables

Headline middleline bottomline

The separation between the lines is set according to the following formula:

Downward movement of cursor =

Baseline-bottom line distance (current font)
+ Top line-base line distance (default font)

Figure 7.5 demonstrates how this looks in diagrammatic form.

Example of text

on two lines

top line 1

base line 1

bottom line 1

top line 2

base line 2

bottom line 2

d
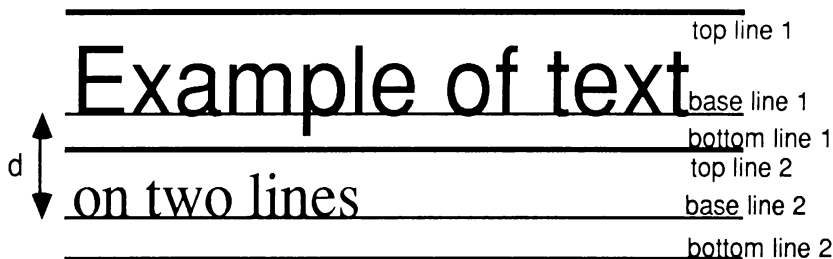
Figure 7.5 The cursor moves distance d to set the line separation

Figure 7.6 shows some examples of text effects generated with the following program.

```
REM text generation program
'generates text in different sizes and fonts
CLS
WINDOW OPEN
WINDOW FULL
WINDOW TITLE "Using fonts on the screen"
   SET POINTS 18
   SET FONT 1
   MOVE 0;4000
'different font sizes all use the same baseline
   PRINT "System font, size = 18 point";
   PRINT POINTS (8) FONT (12);" Swiss font, size =
8 point"
   MOVE 0;3000
   SET POINTS 10
'smaller text sets line spacing
   PRINT "System font, size = 10 point"
   PRINT POINTS (18) "System font, size = 18 point"
   MOVE 0;2000
'now use ADJUST to prevent overwriting
   PRINT "System font, size = 10 point"
   PRINT ADJUST (18) "System font, size = 18 point"
   MOVE 0;1000
```

110

```
   PRINT POINTS (72) FONT (3) "Dutch font, size = 72
point"
WHILE  BUTTON = -1:WEND

END
```

Note the use of the **ADJUST** print function to automatically adjust line spacing
when overwriting would otherwise occur.
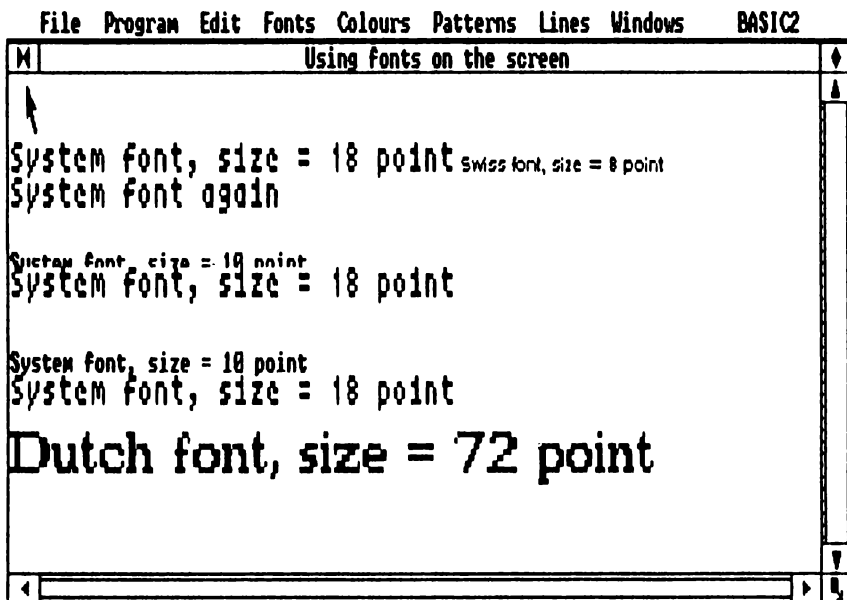
Figure 7.6 Mixing different font sizes on the screen

Angled text is obtained using the command **ANGLE**. Although BASIC 2
will theoretically allow you to print text at any angle, the choices available are
set by the version of GEM that you are currently using. You may well be
limited to multiples of 90 degrees,

```
PRINT ANGLE (90); "up the page"
```

or in general:

```
PRINT [#stream] ANGLE (angle); variable(s)
```
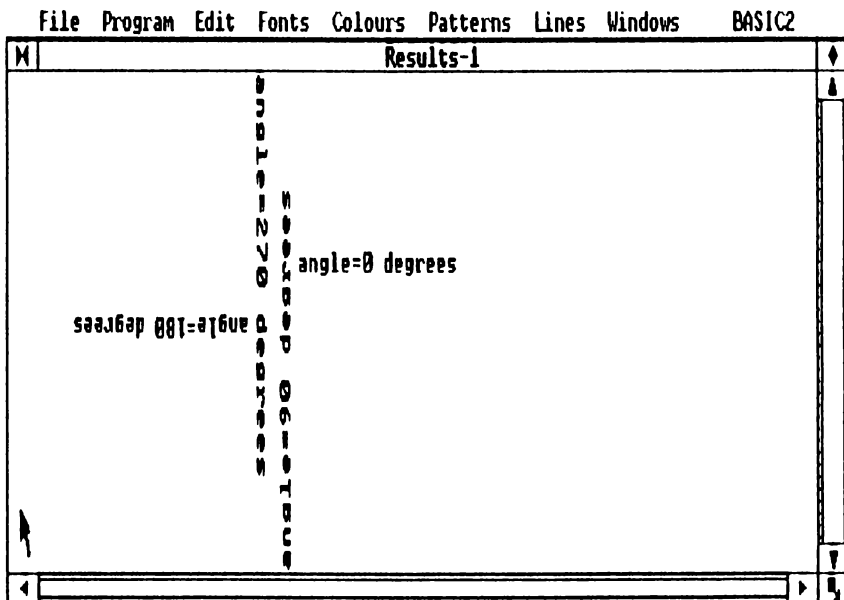

Figure 7.7 Example of angled text in BASIC 2

```
REM text angle demonstration program
CLS
WINDOW OPEN
WINDOW FULL
SET #1 POINTS 10
MOVE 3000;3000
PRINT ANGLE(0) "angle=0 degrees"
MOVE 2500;2500
PRINT ANGLE(180) "angle=180 degrees"
MOVE 2700;500
PRINT ANGLE (270) "angle=270 degrees"
MOVE 2800;5000
PRINT ANGLE (90) "angle=90 degrees"
MOVE 0;0
```

```
WHILE BUTTON=-1:WEND
STOP
END
```

In normal printing onto the screen, text appears in the chosen pen colour. This is fine if text is not to overlap with any other graphics, but you can see the problem that arises in Figure 7.8 below. If text is printed onto existing graphics it replaces *whole blocks* of the graphics. The reason for this is that the text printed on the screen is the whole character cell, and not just the text bit of the cell.

There are four different *modes* in which you can write text to the screen. These are analogous to the modes used to write graphics objects (points, lines and so on) to the screen. The default is **replace mode**, in which text is just pasted over anything underneath. The second mode prints only the characters, and is called **transparent mode**. These are the more common modes that you will use most of the time, and are illustrated in Figure 7.8. The less common modes are **XOR mode**, which is similar to transparent mode but the colour of each text pixel is **XOR**ed with the present colour of the pixel at that location. Although this might seem a rather esoteric option for text display it is extremely useful for *removing* text, as the next program example will show.

The fourth way of displaying text is called **reverse transparent mode**. This displays the text in 'negative' fashion, with the background coloured and the text characters uncoloured.

The write modes are coded:

1. Replace mode
2. Transparent mode
3. XOR mode
4. Reverse transparent mode

and they are set using the command:

SET [#stream,] **MODE** write-mode

Figure 7.8 also shows examples of the use of **XOR** and reverse transparent mode, and the program following it gives the code used to generate the figure.
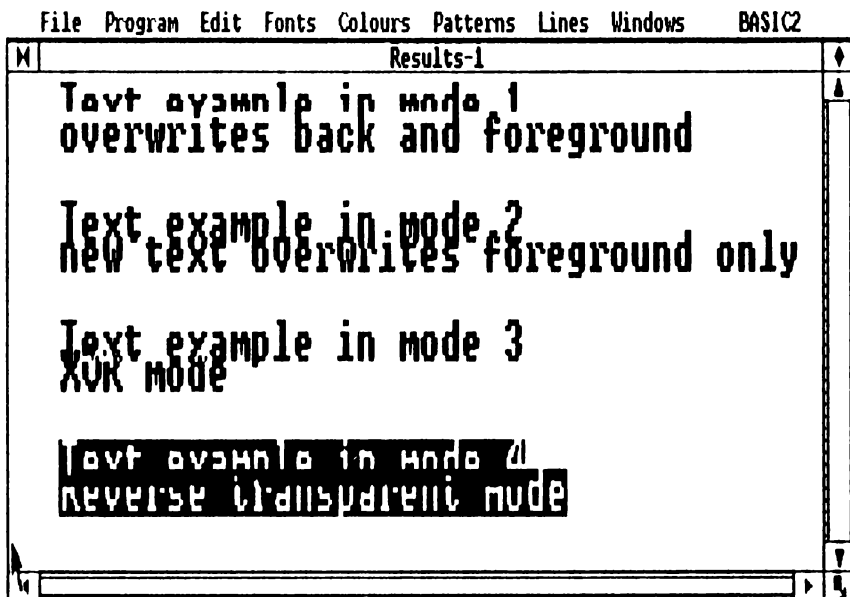
Figure 7.8 Different text modes in BASIC 2

```
REM program to demonstrate different text modes
CLS
WINDOW  OPEN
WINDOW  FULL
MOVE 500;4500
SET    MODE 1
PRINT POINTS(24) "Text example in mode 1"
MOVE 500;4200
PRINT POINTS(24) "overwrites back and foreground"
MOVE 500;3300
PRINT POINTS(24) "Text example in mode 2"
MOVE 500;3000
SET MODE 2
PRINT POINTS(24)  "new text overwrites foreground
only"
MOVE 500;2100
SET MODE 3
PRINT POINTS(24) "Text example in mode 3"
```

```
MOVE  500;1800
PRINT  POINTS(24)  "XOR mode"
MOVE  500;900
SET  MODE  4
PRINT  POINTS(24)  "Text  example  in  mode  4"
MOVE  500;600    ·
PRINT  POINTS(24)  "Reverse  transparent  mode"
SET  MODE  1
WHILE  BUTTON=-1:WEND

END
```

You may like to experiment with combinations of text modes other than those superimposed here. Note that the XOR option actually *removes* colour from those pixels already in the foreground, as shown in the diagram below.

An important application of the XOR text mode is to neatly erase text at a particular screen location (this is useful if you want to take off text without clearing the whole screen). If **MODE 3** is selected, and text is written over the *identical* existing text, the text will be cleared from the screen.
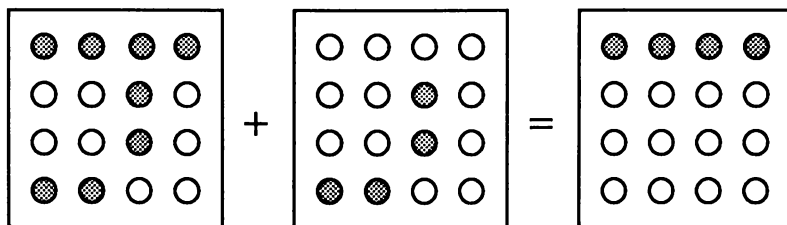
Figure 7.9 Operation of the XOR write mode. If the sequence of pixels on the left is overwritten by the pattern in the centre, the result is as shown on the right.

```
REM  example  of  MODE  3  text
CLS
WINDOW        FULL
WINDOW        OPEN
MOVE  500;3000
PRINT"The  text  is  here!"
WHILE  BUTTON  =  -1:WEND
```

```
MOVE 500;3000
PRINT"The text is here!"
MOVE 500;2000
PRINT"The text has moved here!"
WHILE BUTTON = -1:WEND

END
```

The use of the graphics screen to print text allows far more flexibility in the way in which you can display text output. The use of different print modes and ability to locate the text accurately to a single pixel width allows great scope for experimentation. Figure 7.10 shows a simple example. The shadowed effect has been obtained by using print mode 2 (overwrites foreground only), sequentially writing the same text 20 user coordinates along the x axis and down the y axis. In order to obtain the white text at the end of the sequence, the final text string is rewritten using XOR mode: this reverses the colour of each pixel (see the above figure).

Figure 7.10 Generation of shadowed text using print modes 2 and 3

```
REM text shadow demonstration program
'generates shadowed text by multiple drawing
CLS
WINDOW OPEN
WINDOW FULL
WINDOW TITLE "Shadowed text example
SET FONT (1)
SET MODE (2)
FOR i = 1 TO 200 STEP 20
   MOVE 500+i;4000-i
   PRINT "special text effects";
NEXT i
   SET MODE 3
   MOVE 700;3800
   PRINT "special text effects";
   SET FONT (2)
   SET POINTS (36)
   SET MODE (2)
FOR i = 1 TO 200 STEP 20
   MOVE 500+i;3000-i
   PRINT "can enhance the output";
NEXT
SET MODE 3
MOVE 700;2800
PRINT "can enhance the output";
WHILE BUTTON = -1:WEND

END
```

Chapter Eight

Working with files and devices

8.1 File and directory names

All permanently stored information on your computer system will normally reside on floppy or hard disks in the form of **files.** Files may be of several different types. They may be *program files*, or *data files*, depending on whether they control the computer's actions or provide data for a program.

The hard or floppy disks are referenced by the **drive** in which they reside. The disk will be segregated into one or more **directories**, each of which can hold a number of files. You can consider a disk as a filing cabinet, and a directory as a drawer in the cabinet.

filing cabinet = drive

folder = file

drawer = directory
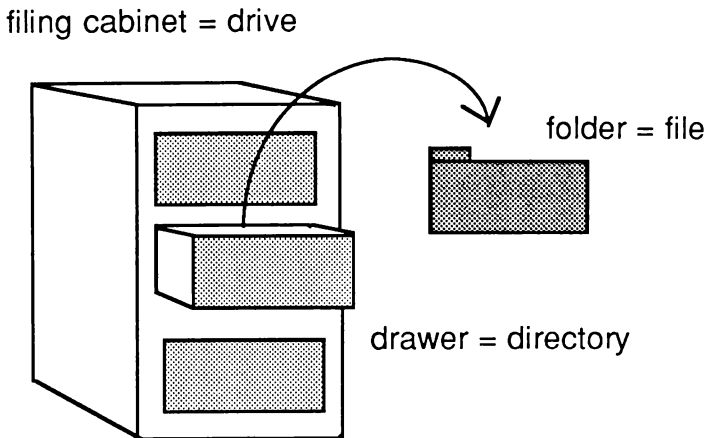
Figure 8.1 Analogy of filing cabinet to files on disk

In order to access the file required, you need to find the right cabinet, then the

correct drawer. Here is the general syntax of the command to access a file in a particular disk.

```
[drive:] [\] [path\] file
```

where drive is the disk letter (followed by a colon), path is a sequence of one or more directory names each separated by a backslash (\) character. Directories are arranged hierarchically, starting with the main or root directory. If LOOKUP is a directory directly below the root directory in the hierarchy, the file HELP may be accessed like this:

```
A: \LOOKUP\HELP
```

Here is a typical arrangement of directories. In order to access the directory A11 in the figure below, you would type:

```
\root directory\directory A\directory A1\directory A11
```
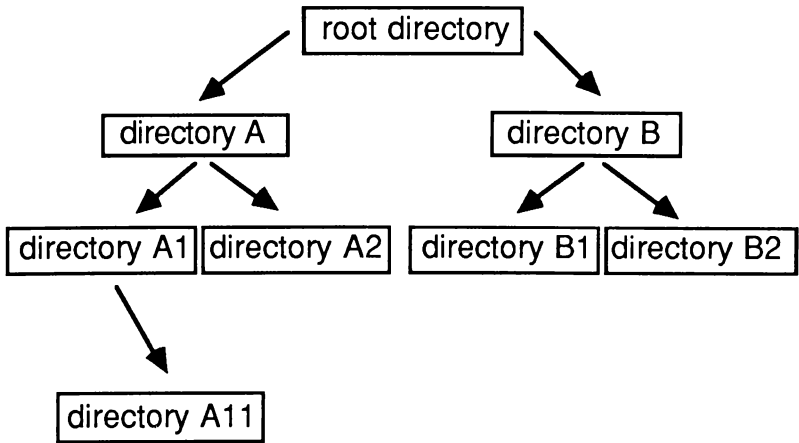

Figure 8.2 Hierarchical paths to directories

Note that the directory names used here are used to show the general scheme, and are not legitimate directory names.

As you might expect, commands are available from within BASIC 2 for changing the current directory, for making a new directory, and for removing a directory.

These commands are:

CD `<newdirectory name>`

to specify a new current directory

MD `<newdirectory name>`

to create a new directory

RD `<directory name>`

to remove or delete a directory.

These commands are all equivalent to MS DOS commands, and take the rest of the line as the parameter. The following commands do the same job as their respective counterparts above, but take *string expressions* as parameters.

CHDIR `<newdirectory name>`

MKDIR `<newdirectory name>`

RMDIR `<directory name>`

You may also use several functions to get information about the directories on the drives. These functions are **CHDIR$** and **FINDDIR$**.

CHDIR$ is used to find the name of the current directory, hence:

`result$ =` **CHDIR$** `[(string)]`

where `(string)` is a drive letter, e.g. `"A"`. If no drive letter is specified, the current directory on the current drive will be returned. **FINDDIR$** is used to find a particular directory on a drive. In fact the parameters used with **FINDDIR$** allow you to search the specified disk for the nth directory whose name matches that specified. This might seem a little bizarre (ordinarily you would not have more than one directory with the same name on a disk). The reason is that you can use *wildcards* when using **FINDDIR$**. A wildcard is a special character that can be used in a name, so if the wildcard is *, the following names will be equivalent:

```
MAINDIR1
M*
MAIN*
MAINDIR*
```

If for example you have the following directories on a disk:

```
MAIN1
MAIN2
MAIN3
MAIN4
MAIN5
```

The command::

```
result = FINDDIR$("MAIN*",3)
```

will return the directory name **MAIN3.** (Assuming that the directories appear in the given order in the parent directory.)

In order to change the current drive, the command

DRIVE string

is used. If you have a single drive it will usually be named drive **A**. If you have two floppy drives, they will be **A** and **B**. If you are lucky enough to have a hard disk it will normally be designated drive **C**.

8.2   File structure

In the rest of this chapter we will be looking at how files are handled in BASIC 2. Before doing this, let us consider some general terms and concepts used in file handling. Computer files are arranged in a number of blocks called **records**. In the simplest case, each record is a single data item: a number or text string. Each data item is held in a measure called a **field**. Again, the simplest type of file has a single field per record. More sophisticated types of files have multiple fields for each record. In the most complex kind of files, each field can individually be accessed by assigning a **key** or **index** to the various fields.
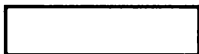
record with	record with multiple fields
single field	(text, numeric or mixed text
(text or numeric)	and numeric)

field1 field2 field3 field 4 field5
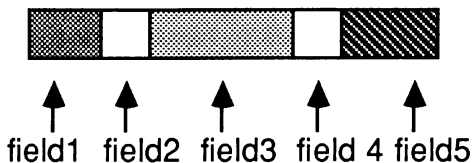
Figure 8.3 Comparison between records with simple and multiple fields

8.3   Sequential files

Sequential files are the most common form of data file. Sequential files are also the easiest form of data file to use, and if you are not going to deal with databases you may find that they serve all your needs. If you have ever used the **DATA** command in a dialect of BASIC then you understand the principle of sequential storage, for example:

```
DIM x(10)
DATA 1,2,3,4,5,6,7,8,9,10
   FOR i = i TO 10
       READ a
       x(i) = a
     NEXT i
END
```

will write the numbers 1 to 10 sequentially into the array x.

This short program segment is fine if you are dealing with a small number of variables, or if the program is not likely to be run many times with different sets of data. If these conditions are true, it may well be better to hold the data in a disk file. In order to demonstrate the simple use of a sequential file, let us look at how the data in the above example could be written to a disk file, and then retrieved by a second short program.

The first task is to create a sequential file. The important command is **OPEN**,

which takes the following general form.

```
OPEN #stream [qualifier] mode filename
```

In this case we will use stream #3. There are 16 streams available to BASIC 2, but streams 0 to 2 are reserved for output to windows and the printer. This leaves streams 3-15 for general use. The qualifier is an optional word specifying whether the file is **NEW** or **OLD**. The qualifier is useful if you need to check whether or not the chosen filename preexists.

The **mode** may be **INPUT, OUTPUT**, or **APPEND**. If a filename already exists, and **OUTPUT** is chosen, the existing file is overwritten. To guard against this, use of the qualifier **NEW** before output will ensure that an error is generated if the chosen filename preexists.

In order to write data to a datafile, the command **PRINT** is used, with the stream specified (if no stream is specified, output goes to the **Results-1** window. To read data from a datafile, the command **INPUT** is used, again specifying the stream to be read from.

When you have finished reading to or writing from a file, you should always close it, using the command **CLOSE** (if you wish to close all open files) or **CLOSE** #stream if you want to close only specified streams. Although your simpler programs will probably work fine without taking the precaution of closing files (and BASIC 2 closes all files for you when you exit it), file closure is an excellent habit to acquire. For one thing it ensures that all the data that *should* have been written to a specific file gets to the file, and is not hanging around in some buffer somewhere. It also puts the file into a state from which it can be reopened: useful if you want to **INPUT** from a file to which you had previously **OUTPUT**. Here are two short programs that demonstrate creation and recall of data from a datafile called onetoten.

```
REM program demonstrating writing to sequential
datafiles
REM first create the datafile and write the data to
it
DATA 1,2,3,4,5,6,7,8,9,10
OPEN #3 NEW "onetoten" OUTPUT
  FOR i = 1 TO 10
      READ a
      PRINT #3,a
```

```
   NEXT i
CLOSE #3

REM program demonstrating reading from sequential
datafiles
REM now open the datafile and read the data back from
it
OPEN#3 OLD "onetoten" INPUT
  FOR i = 1 TO 10
     INPUT #3,a
     PRINT a
     REM that prints the data to the screen to show
that it is there!
   NEXT i
CLOSE #3
END
```

This is all you need to know to create sequential files using BASIC 2, apart from a couple of embellishments. The first of these concerns the end of the sequential file. The above simple programs work well because they both work with the same number of data items. In some cases you may not know how many data items you have. A program may need to read up to the total amount of data, and to work out itself when the data is exhausted. If you do not take precautions, data exhaustion will result in an error. You can get over this by placing a special data item at the end of the data set. When you read the data, check each time to see if the data item is this 'end flag'.

For example consider the following data set, held in a file called `datafile`.

`1,7,2,5,6,12,14,23,34,77,12,9999`

The data consists of the first 11 numbers, and the 12th, '9999' is a dummy value acting as the end flag. Here is a general program reading in any number of values from a sequential file, checking each time for the presence of the end flag

```
OPEN #3 OLD INPUT datafile
REPEAT
   INPUT #3, number
   IF number <> 9999 THEN PRINT number
UNTIL number = 9999
END
```

BASIC 2 allows you the function **EOF** to do the same job. **EOF** returns the value 'false' if the end of the file has not been reached, so the above program segment could be rewritten as:

```
OPEN #3 OLD INPUT datafile
WHILE  NOT(EOF(#3))
   INPUT #3, number
   PRINT number
WEND
END
```

and this will be faster in operation.

Sequential files are useful in many different areas, especially mathematical, statistical and technical fields where lists of numbers are to be processed, and the data does not have to be searched in any way (the advantages of other types of file for searching will be discussed in the next section). Here are two programs that build on your knowledge of file handling and of graphics programming gleaned from the last chapter. The first program (called DOTTO) allows you to draw a shape made of up to 100 lines by pointing at the next point position and clicking the mouse button. When you have finished your masterpiece, you can save it to a sequential file by pressing the S key. You can then load and run the second program (PICASSO). This will prompt you for the name of a sequential file and will redraw any file created using DOTTO.

```
REM DOTTO point connection program
'allows you to connect points on the screen into a
'shape and then to save the file to disc
'the sequential file can be recalled using the
'PICASSO progam
'first set up arrays
DIM xpts(100),ypts(100)
pt_counter = 0
CLS
WINDOW  OPEN
WINDOW  TITLE  "dotto"
WINDOW  FULL

PRINT "start clicking, press the mouse button and a
key to save the file"
```

126

```
FOR i = 1 TO 100
pt_counter = pt_counter+1
'
'now read the x and y points - note the formula to
'convert XMOUSE and YMOUSE screen coordinates to
values
'on the virtual screen
LABEL wait
GOSUB rep
    x_coord = (XMOUSE-XPLACE)+XPIXEL
    y_coord = (YMOUSE-YPLACE)*YPIXEL
'check to see if two identical points follow each
other
IF x_coord =xpts(pt_counter-1) AND y_coord =
ypts(pt_counter-1) THEN GOTO wait
xpts(pt_counter)=x_coord
ypts(pt_counter)=y_coord
IF pt_counter = 1 GOTO flag
LINE
xpts(pt_counter-1);ypts(pt_counter-1),xpts(pt_counter)
;ypts(pt_counter)
MOVE 100;4500
PRINT "points = ",pt_counter,"lines = ",pt_counter-1
LABEL flag
  GOSUB rep
NEXT i
LABEL savefile
'this routine saves the coordinates as a sequential
file
    CLS
    MOVE 500;4500
    IF INKEY$="s" THEN GOTO savefile
    INPUT"filename to save as";filename$
    OPEN #5 OUTPUT filename$
    PRINT #5 pt_counter
FOR i = 1 TO pt_counter
    PRINT #5,xpts(i),ypts(i)
NEXT i
CLOSE 5
END
LABEL rep
```

```
IF INKEY$<>"" THEN GOTO savefile
temp = BUTTON
IF temp=-1 GOTO rep
RETURN

REM PICASSO point drawing program
'allows you to draw points on the screen
'created using the DOTTO program
CLS
WINDOW OPEN
WINDOW FULL
CLOSE #5
'first open the saved file
INPUT"filename for input";filename$
OPEN #5 INPUT filename$
CLS
INPUT #5 pt_counter
DIM xpts(pt_counter),ypts(pt_counter)
'now read in the x and y coordinates
   FOR i = 1 TO pt_counter
   INPUT #5 xpts(i),ypts(i)
    NEXT i
'now draw the shape
FOR i = 1 TO pt_counter-1
LINE xpts(i);ypts(i),xpts(i+1);ypts(i+1)
NEXT i
CLOSE 5
WHILE BUTTON=-1:WEND

END
```

8.4 Random files

The problem with sequential files is that they are rather like fruit pastilles in a tube. To get to the flavour you want, you have to eat the others first! Random files are like chocolates in a box: you can hog your favourites without eating those you don't like.

Information in random files can be read in any order and can be changed directly. These two advantages make them more flexible than sequential files, but at the expense of a more complex exercise in computer programming. Random files

are also more expensive on disk space than are sequential files.

The first difference between sequential and random files is the way in which data is stored in them. As you will never directly access a sequential data file 'in the middle', the length of each data item is of no real consequence: any data item that can fit into a variable will go into a sequential data file. With a random file on the other hand, you may wish to access the 500th data item. It would be very inefficient if the computer had to painstakingly work its way down 499 data items, checking how long they were, before it arrived at number 500. All the data items in a random file therefore have the same fixed length, and this length is called a **record**. A record can consist of more than one data item (and of varying types of data item in such cases).

Let us simplify the explanation of random files in BASIC 2 by dealing with a clear example. This example is a telephone account list. The entries will be of the form:

name   telephone   number   amount

As with sequential files, the important steps are opening and closing the file, together with reading to and writing from the file. Recall that the **OPEN** command for the sequential file was:

OPEN #stream [qualifier] mode filename

The equivalent random file is opened using the general form:

OPEN #stream [qualifier] **RANDOM** filename [**LENGTH** length]

There are several differences to sequential files. Firstly, you do not specify whether you want the file for input or output. You can do both to an open random file. Apart from the **RANDOM** specification for the file, the other main difference is that the length of each record may be amended. If the **LENGTH** specifier is not set, it defaults to 128 characters per record in BASIC 2.

For each random file, a record structure has to be defined. This is done using the command **RECORD**. **RECORD** defines the names to be used for the fields within each record, and has the general form:

RECORD record name; field1, field2,....

129

where each field can have both a **range** and be allocated a **storage class**. The range, if present, denotes that the field is a one dimensional array, and the space occupied by each storage class is as follows:

Field	Space occupied (characters)
`field name`	8
`field-name$` **FIXED** **n**	**n**
BYTE,UBYTE	**1**
WORD,UWORD	**2**
INTEGER	**4**

(The uses of these various storage classes were discussed in Chapter 4, Section 4.4.)

Here is a typical record structure where each record is to hold a name (up to 20 characters), a telephone number (8 characters), and a total charge (8 characters).

```
RECORD phonebill; name$ FIXED 20, number, amount
```

The record would look like this:
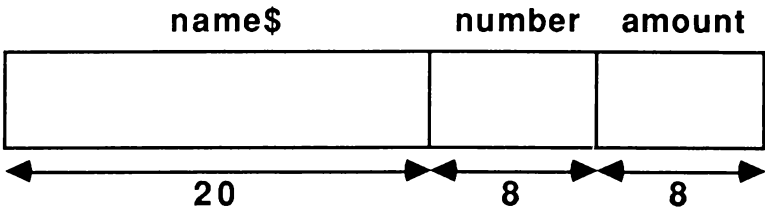
Figure 8.4 Record structure

How can the records be used? The first step is to be able to enter information into the record. This is done by assigning a value to the field within the record. The variable and the record are specified using the general form:

```
recordname.field
```

so for the name part of our **phonebill** record, we could assign:

```
a$.phonebill.name$ = "ANNE RANSOM"
```

130

which will fill 13 characters of the allowed 20 in the field **name$**. The rest of the characters remain as nulls. If we then `PRINT` the record, the string is returned, so:

`PRINT` a$.phonebill.name$

will print the assigned name.

Before looking in more detail at our specific example of a random file, there are several other commands and functions that have to be understood. These are concerned with accessing particular records. For instance, you may want to change the price of a stock item in an inventory list. If you know the record number you can do this without accessing the other records.

At any time, an imaginary 'pointer' sets the record number that may be accessed. To find where this pointer is sitting at any time, you may use the function **LOC,** which returns an integer number:

pointer position = **LOC**(#stream)

where #stream is the stream assigned to the random file. To move to a specific record position, the commands:

POSITION #stream **NEXT**
or

POSITION #stream **AT** record number

are used.

A second function, **POSITION$**, produces a character string specifying the current position.

pointer position string = **POSITION$** (#stream)

You can use **POSITION$** to specify the record number to move the pointer to as an alternative to using **LOC**. It might seem pedantic to have two functions returning the same result, and **POSITION$** is in fact more widely used with **keyed files** discussed in the final part of this chapter.

POSITION #stream **AT** position string

131

The only remaining random file commands to learn are those for reading and writing to random files. These are analogous to the **PRINT** and **INPUT** commands used with sequential files, and are called **PUT** and **GET**.

```
PUT #stream, string expression [position]

GET #stream, string variable [position]
```

The string expression is the data string to be written to the file.

We can now put all these commands together to make a working random file. For simplicity, we'll consider the phone account file for which we have already seen the **RECORD** template. You may well find it easier to step through the following program than to understand the description of the various commands separately.

There are four main processes to handling a random file. First the file has to be opened (if it is a new file, the **OPEN** command also 'declares' the file). Next, the **RECORD** structures have to be defined. The records can then be manipulated: added, deleted or moved around. Finally (and most important) the file must be **CLOSE**d when processing is complete.

In the program PHONEFILE below, you can do these various operations. The program creates a list of names and the amounts in their accounts, together with the phone number. Given the record number, an individual record can then be accessed *without stepping through the other records.*

PHONEFILE allows you to create, list and individually review entries. It is a skeleton program that could be easily adapted to hold other information. You could modify it to hold data on the members of a club, for example: given the 'membership number', all the information for a particular member could be instantly recalled.

```
REM random file demonstration program PHONEFILE
   WINDOW OPEN
   WINDOW FULL
   LABEL top
   CLOSE #5
   RECORD phonebill;name$ FIXED 40,number INTEGER,
amount
   filename$="accounts"
```

```
   OPEN #5 RANDOM filename$ LENGTH 53
   alert_value = ALERT 1 TEXT "click on required
option" BUTTON "new file","list file","select item"
   IF alert_value = 2 THEN GOTO listall
   IF alert_value = 3 THEN GOTO select_item
'data input section for new file
   PRINT "PHONE ACCOUNTS PROGRAM"
   PRINT "Please enter data carefully as prompted"
   PRINT "Press the mouse button to proceed"
   WHILE BUTTON=-1:WEND
CLS
'initialize the temporary holding string for data
temp$=STRING$(53,0)
   INPUT "How many customers will you be
entering";cust
   no=0
   REPEAT
      INPUT "Type customer name";temp$.phonebill.name$
      no=no+1
      INPUT "Type customer phone number";
temp$.phonebill.number
      INPUT "Type the amount owing"; temp$.phonebill
.amount
'now write the information to the random file
   PUT #5,temp$
   POSITION #5 NEXT
   UNTIL no=cust
GOTO end_of_program
'listing section
LABEL listall
   CLS
   rec_number = 0
   REPEAT
      rec_number = rec_number+1
      GET #5, rec$ AT rec_number
      PRINT rec$.phonebill.name$
      PRINT rec$.phonebill.number
      PRINT rec$.phonebill.amount
   UNTIL EOF (#5)
   WHILE BUTTON=-1:WEND
   GOTO end_of_program
```

```
'item selection section
LABEL select_item
CLS
    REPEAT
        INPUT "Type entry no or 0 to quit";ent_no
        IF ent_no = 0 THEN GOTO end_of_program
        GET #5, rec$ AT ent_no
        PRINT rec$.phonebill.name$
        PRINT rec$.phonebill.number
        PRINT rec$.phonebill.amount
    UNTIL ent_no = 0
    WHILE BUTTON=-1:WEND

LABEL end_of_program
CLS
alert_value = ALERT 1 TEXT "OK hot shot? What now!"
BUTTON "start again","close file and end"
IF alert_value = 1 THEN GOTO top
CLOSE #5
END
```

8.5 Keyed files

The notion of keyed files is not common among BASIC dialects, and BASIC 2 and Mallard BASIC are the only common ones that feature keyed files. A keyed file is a random file where each record is not specified by its record *number* but by a number of **keys** defined for the record.

Imagine a personnel file for employee number 123. 123 is the employee's *record number*, and the only way you could access his data (assuming it were stored in a random file) would be by printing out data accessed in the record 123. If the data were instead stored in a keyed file, a number of fields within the records of the file could be accessed by assigning a key to each field.

If we were constructing a random file, the only information that we could directly access would be the employee number. The employee number is an *index* to the data in the same way as the number of an array element is an index to the contents of that element. The advantage of a random file over a sequential file is that you do not have to go through the file entry by entry in order to get to the

information. The advantage of a keyed file over a sequential file is that various fields within each record can be independently accessed. The key or index is the 'map' by which this can be done. The figure on the next page shows a comparison of the three types of file.

If you understand how to use random files, you should not have much difficulty in grasping the rudiments of the use of keyed files. Keyed files are processed in four stages. (1) Opening the file, (2) defining record structures, (3) processing the records and (4) closing the file. These stages are essentially the same as for a random file. For each keyed file, between 1 and 20 indexes may be created. The indexes are not created in the same way as the keyed files, but are automatically generated by the **KEYSPEC** command.

sequential file   random file   keyed file
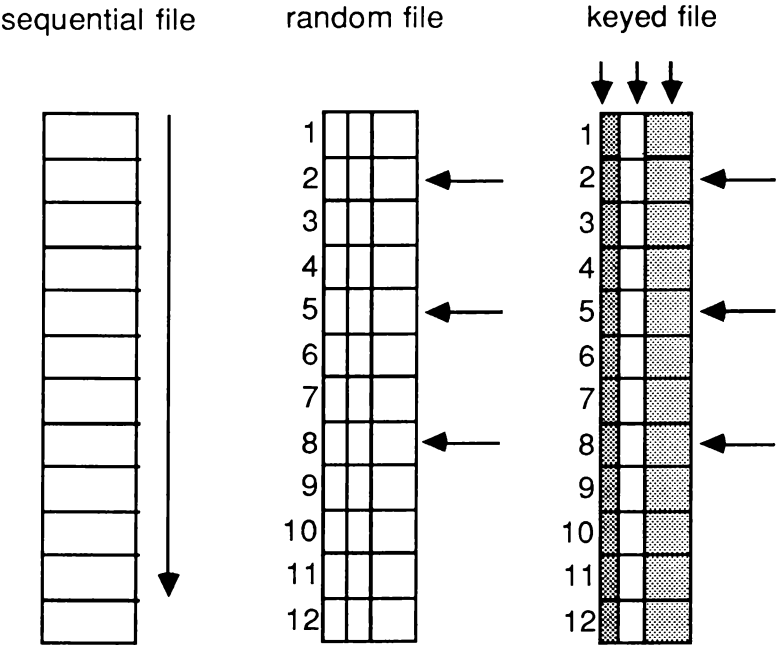
Figure 8.5 Comparative access of sequential, random and keyed files. Sequential files may only be accessed by reading all the entries from the top of the file. Sequential files also contain only one field per entry. Random files are arranged in records and have multiple fields, and any record can be accessed directly by number. Keyed files are also arranged in records, and individual fields within each record can be accessed by defining a key (or index) for each field.

KEYSPEC #stream **INDEX** indextype [key-type] [**UNIQUE switch**]

The index type can be any of the storage classes considered for random files, namely **BYTE**, **UBYTE**, **WORD**, **UWORD**, **INTEGER**, or **FIXED**. The **key-type** is only defined if the keys are not to be variable length strings. **switch** is a logical expression setting whether or not duplicate keys are allowed. Duplicate keys are allowed if the value is **OFF**. They are not allowed if the switch is set to **ON**.

Each index must only be defined once. Let us look at a typical keyed file system: an employee record file. The file contains the following data for each employee.

Employee number
Name
Age
Date of Birth
Time in Service
Department

We might wish to access the records in the file by using any of these fields as keys. In this case, an index for each of the keys must be set up.

Before the set of keys can been defined, the keyed file itself must be created. This is done by using the **OPEN** command, as for the other file types. In this case the syntax is slightly modified.

OPEN #stream [qualifier] **RANDOM** filename **INDEX** indexname [**LENGTH** recordlength]

Stream, qualifier and filename are as for the other kinds of file. The index name is the name for the index file, and the length may be varied from the standard 128 default.

```
REM create employee record
'
REM * constants and variables *
index_file$ = "emp.idx"
data_file$ = "emp.dat"
```

```
employee_index = 1
name_index = 2
age_index = 3
time_index = 4
dept_index = 5
'
SCREEN TEXT FLEXIBLE
WINDOW CURSOR ON
CLS
PRINT "employee record creation program"
PRINT
'
REM * set up file *
CLOSE #5: 'this is the chosen stream
IF FIND$(data_file$)<>"" THEN KILL data_file$
IF FIND$(index_file$)<>"" THEN KILL index_file$
OPEN #5 NEW RANDOM data_file$ INDEX index_file$
LENGTH 50
'
KEYSPEC #5 INDEX employee_index UWORD UNIQUE OFF
KEYSPEC #5 INDEX age_index UBYTE UNIQUE OFF
KEYSPEC #5 INDEX time_index UBYTE UNIQUE OFF
'
' define record structures *
RECORD emp_file;emp_no UWORD, name$ FIXED 40,age
UBYTE,tis UBYTE,dept$ FIXED 4
r$ = STRING$ (50,0):'initialise string'
'


'process records
LABEL loop
PRINT "type employee no. followed by return"
INPUT "or 0 to exit";r$.emp_file.emp_no
IF r$.emp_file.emp_no = 0 THEN GOTO exit
'
'now we have a bona-fide emp no, so get other data
INPUT "employee name";r$.emp_file.name$
INPUT "employee age";r$.emp_file.age
INPUT "time in service";r$.emp_file.tis
INPUT "department (4 letter
abbreviation)";r$.emp_file.dept$
```

```
'now add the record
ADDREC #5, r$ KEY emp_no INDEX employee_index
ADDKEY #5 KEY age INDEX age_index
ADDKEY #5 KEY tis INDEX time_index

GOTO loop
'now finish
LABEL exit

CLOSE #file
END


REM list employee record
'
REM * constants and variables *
index_file$ = "emp.idx"
data_file$ = "emp.dat"
employee_index = 1
name_index = 2
dept_index = 5
'
SCREEN TEXT FLEXIBLE
WINDOW CURSOR ON
CLS
PRINT "employee record listing program"
PRINT
'
REM * set up file *
CLOSE #5: 'this is the chosen stream
'
OPEN #5 OLD RANDOM data_file$ INDEX index_file$
LENGTH 50
'
' define record structures
RECORD emp_file;emp_no UWORD, name$ FIXED 40,age
UBYTE,tis UBYTE,dept$ FIXED 4
POSITION #5 INDEX 3
'3 is the index number for age
```

```
'process records
LABEL loop
WHILE NOT EOF (#5)
GET #5,r$
PRINT "age of employee "r$.emp_file.age
PRINT "name            "r$.emp_file.name$
PRINT "time in service "r$.emp_file.tis
PRINT "department      "r$.emp_file.dept$
PRINT "-------------"
PRINT "press 1 for next record, 2 to abort"
LABEL getkey
  a$=INKEY$
  IF a$="" THEN GOTO getkey
IF a$ = "2" THEN GOTO exit
POSITION #5 NEXT
GOTO loop
WEND
'now finish
LABEL exit
PRINT
PRINT "file ended ..."
CLOSE #file
END
```

Chapter Nine

Advanced Topics

9.1 Advanced BASIC 2

By this time you will have had some experience in using the BASIC 2 user
interface, and should be routinely using the mouse and windows when writing
your programs. You may already have worked through some quite complex
material, particularly the sections on computer graphics in Chapter 6 and the
latter parts of Chapter 7 dealing with random and keyed files. There are a number
of odds and ends that you should know about before considering yourself to be a
fully-fledged BASIC 2 programmer, and we will use this final chapter to
introduce some of these remaining topics.

9.2 Error handling

You will by now have generated numerous errors during your BASIC 2
programming. No matter how seasoned or experienced a programmer you are,
errors are a natural part of the program development process. Errors can be
generated by the programmer, the computer user (if not the programmer) or the
computer itself. The Golden Rule of programming is to assume that all errors
are your own in the first instance. Computer hardware errors are quite rare, and if
you are the programmer as well as the program user, there is normally only one
guilty party!

Errors can occur at three main stages in program development.

1. As the program is being developed (programmer errors).

2. As the program is being tested (programmer errors).

3. As the program is being run in its final form (user errors).

Errors in the first category are likely to involve silly things like the wrong
typing of keywords (**PRONT** instead of **PRINT**), wrong syntax (not closing

`FOR .. NEXT` loops for example) and so on. Errors in the second category often manifest themselves in the form of logic errors: the program does not do what the programmer intended, and therefore requires further modification. Errors of the final group involve miskeying in response to prompts, insertion of a wrong disk, or insertion of no disk at all at a crucial moment. Mistakes of this kind, although technically errors, are often not the fault of the poor computer user but are usually a result of the programmer not making the program 'foolproof'.

Programmer errors (1)

The first class of errors show up soon enough when you try to run all or part of your fledgling program. As you will have found out, the general form of the error message you receive is:

Syntax error

And this information is often sufficient for you to be able to rectify the error. Each error has a numeric error code associated with it, and the code of the most recent error is returned by the function **ERR.** There are several other useful error information facilities. The message associated with the most recent error is returned by the function **ERROR$,** and a particular error can be simulated by the command **ERROR.**

ERROR 4

PRINT" The most recent error was";**ERR**

PRINT" The associated message is";**ERROR$**

If you expect errors to arise (or if you are just plain careful), you can *trap* for them by using an error handling routine. A statement of the form:

ON ERROR GOTO label

forces the program to jump to a particular label when an error is found. The error handling routine must end with a **RESUME** command. There are three different possibilities.

RESUME

This command restarts execution at the *beginning* of the statement that produced the error.

RESUME `label`

This command restarts execution at the given label.

RESUME NEXT

This command restarts execution at the beginning of the statement *following* the statement that produced the error.

To turn off error trapping, the statement:

ON ERROR GOTO `0`

should be used.

A complete list of error codes and the associated error messages will be found in Appendix 4.

A major source of runtime errors is the attempt to open a file that does not exist. Although you can use an **OPEN** statement to create a file for output, the statements

```
INPUT filename$
OPEN #5 INPUT filename$
```

will result in an error if filename$ is not a valid filename, or if the file does not exist. In order to trap for such errors, you may use the **OPEN** command as an *assigned command*, for example

```
ok_test = OPEN #5 INPUT filename$
```

If the variable `ok_test` is set to zero, then the command succeeded. Other values of `ok_test` will be error numbers.

Programmer errors (2)

There is a major problem with the use of error traps, and this is that they are *total*. Finding errors and correcting them is a healthy activity for the computer programmer, and you should not be tempted into using error handling routines to paste over errors without understanding the full implications of what you are doing. However, you might find intelligent use of such a routine of value when debugging a program. You could for example provide a routine that listed the values of all the important program variables on to the screen (or sent them to a printer) whenever an error occurred.

Like all other BASIC statements, the ON ERROR GOTO command will only be active from the point that line of the program is reached during program execution. If you want such a command to work for the whole program, remember to put it in the first few lines.

It is likely that you will spend most of your programming life debugging programs. Once you have worked out the errors that cause the program to halt execution, you will have to ensure that the program is doing exactly what you want it to do. To a large extent this is when you are on your own. Unlike more sophisticated language systems that include debugging facilities (for example to print out states of program variables during execution), BASIC 2 offers only relatively crude help for the programmer. This is one of the few areas in which BASIC 2 is currently lacking, as it does not possess a **TRACE** command to enable you to step through the program line by line.

You can, however, physically stop the program during its execution, and can then directly 'question' the computer about the state of variables. The facilities that you can use are the **Ctrl-C** key to suspend program execution, together with the **? (PRINT)** command to print out variable values in the **Dialogue** window. **?** is used rather than **PRINT** as it directs the output to the **Dialogue** window so that output from the program to the normal output windows is not upset.

The command **CONT** typed into the **Dialogue** window will restart the program again. This command can also be used to restart a program stopped with **STOP** command within the program. **STOP** has essentially the same effect as the **Ctrl-C** keys, except that you can of course control the exact point at which the program is halted. Again, you can use the **?** command to print out variable values to the **Dialogue** window.

User errors

By the time the program is in its final state and is in routine use, it is to be hoped that all the errors will be out of the program. It will still be possible in most cases to 'trip up' the program during execution: by pressing a wrong key, or by failing to carry out an instruction correctly. If the programmer is the program *user*, these events will be of little consequence, because the programmer will know the structure of his or her program pretty well. In many cases, the program will have been written for someone else, and this is often where the fun begins. Consider the following example. A program to calculate Fahrenheit temperatures from Centigrade and *vice versa* has been written. The user dutifully loads in program TEMPCHANGE from disk, and selects **RUN** from the **Program** menu. Horror of horrors, the following appears in the **Results-1** window!

?

What is the user to do? Type in a Centigrade value? A Fahrenheit value? No - logic dictates that the program must first be told which direction the conversion is to be carried out in. Does the program expect an **F** or a **C**? Help!

The programmer must therefore consider what the user can be expected to know and what must be explained. Much more helpful would be for the program to display the following:

```
TEMPCHANGE PROGRAM

TYPE 'F' TO CHANGE FROM FAHRENHEIT>CENTIGRADE

TYPE 'C' TO CHANGE FROM CENTIGRADE>FAHRENHEIT

?
```

So far so good, but now the poor user in her excitement types a **D** instead of an **F**. Will the program know what to do? If only an **F** or a **C** are expected, then the program may either return a strange result or an error may be generated. The programmer must therefore avoid this happening by only allowing the required letters to be typed. A possible programming solution is:

```
LABEL readin
  scale$ = INKEY$
  IF scale$ = "F" THEN GOTO ok
  IF scale$ = "C" THEN GOTO ok
  GOTO readin
LABEL ok
```

The program will remain looping around this code until one of the two letters **F** or **C** is pressed.

The important point to realise is that the programmer must always be sympathetic to the needs of the user. Every possible error that can be generated by pressing wrong keys and entering wrong information must be worked out in advance by the person writing the program. Many programs are fault trapped in only the most rudimentary of ways. Make sure that you acquire good habits and your programs will be successful!

9.3   More on streams: the Results-2 window

As this is a 'loose-ends' chapter, we will now return to a promise made back in Chapter 2, and this concerns the functioning of the **Results-2** window. We have not bothered before with this window, as it seemed sensible to concentrate on the generation of the full range of graphic and text effects to the main **Results-1** window.

The **Results-2** window is written to using stream #2, so if you move the windows around the screen to show the **Results-2** window and then type:

PRINT #2 " Here we go ..."

into the **Dialogue** window, the output will go to the **Results-2** window.

The main use of the **Results-2** window is for text output. This window does not allow use of graphics output, unless a suitable **SCREEN** command is given. You could for example set up a program to draw graphics into the **Results-1** window and text into the **Results-2** window. Here is a simple illustration of this technique, based on the window manipulating commands that you met in Chapter 6.

```
REM example of use of text and graphics windows
'using streams #1 and #2
CLS
'first set up the windows
SCREEN #2 TEXT 620,50
WINDOW #1 CLOSE
WINDOW #2 CLOSE
WINDOW #1 SIZE 620,115
WINDOW #1 PLACE 0,200
WINDOW #1 OPEN
WINDOW #2 SIZE 620,5
WINDOW #2 PLACE 0,0
WINDOW #2 OPEN
'now create the output
PRINT #2 "press the mouse button to draw a shape"
WHILE BUTTON=-1:WEND
    CIRCLE 2000;2000,500 COLOUR 7 FILL
    PRINT #2 "now see what happens when you press
again!"
WHILE BUTTON=-1:WEND
    CIRCLE 1500;2500,200 COLOUR 3 FILL
    CIRCLE 2500;2500,200 COLOUR 3 FILL
    PRINT #2 "try pressing yet again!"
WHILE BUTTON=-1:WEND
    ELLIPSE 2000;1700,300,.2 COLOUR 5 FILL
    CIRCLE 1700;2300,50 COLOUR 6 FILL
    CIRCLE 2300;2300,50 COLOUR 6 FILL
    SHAPE 1800;2200,2200;2200,2000;1800 COLOUR 1
FILL
    PRINT #2 "press now to end the program"
WHILE BUTTON=-1:WEND

END
```

9.4 Output to other devices

For most purposes, you will be sending output either to the screen or to a
printer. We have concentrated so far on screen output, but text output to a
printer is quite straightforward, providing that your printer is connected correctly
to the computer. (Methods of connection are described in your *Amstrad PC1512*

147

User Instructions or appropriate User Manual.) The BASIC 2 commands:

LPRINT

and:

TAB (n)

can be used. The printer is specified as stream #0, so if you set the current stream to 0, the output will go to the system printer.

To define other streams for printer output, you should use the command:

OPEN #stream PRINT [device number]

To use a graphics device (eg a plotter, tablet or camera) you must make sure the appropriate GEM driver is available, and that you have sufficient memory on your computer!

OPEN #stream DEVICE [device number]

GEM reserves device numbers as follows:

Device number	Device
1-10	screens
11-20	plotters
21-30	printers
31-40	metafiles
41-50	camera
51-60	tablet

Screen dumps

If you have a suitable printer attached to your computer you can obtain a dump of the image on the screen at any time within the GEM environment. To do this you use the **shift** and **Prt Sc** keys. First, however, you must prepare the printer in the following way.

Boot up your computer using MS DOS, and when the **A>** prompt appears, type:

after pressing <return>, type:

MODE LPT1:,8

This selects the appropriate line spacing on the printer. You are now ready to load in GEM, and you can do this on the Amstrad PC1512 by typing GEM. (This is probably true for most other GEM implementations as well.) When the Desktop has appeared you can dump the screen (before or after entering BASIC 2) by pressing the **shift** and **Prt Sc** keys together. If you need to abort the screen dump, press the **Break** key: this will immediately halt printing.

If your screen dump appears with spacing between the lines, you may have to adjust your printer. The screen dumps in this book were obtained using an EPSON MX80 graphics printer, with the line spacing software selected at 7/72". If you need to select the line spacing on your printer you will probably be able to do it from BASIC 2 by using the **LPRINT** command: see your printer manual for details.

An alternative, more long winded, and less satisfactory way of dumping the screen is to use the GEM desk accessory **Snapshot** to save a portion of the screen to disk. The advantage of this is that the resultant file could be loaded by a graphics application, e.g. GEMPAINT. The diasadvantage is that you will need a memory expansion to run **Snapshot** within BASIC 2, as BASIC 2 and the Desktop use 256K of memory each. A 512K Amstrad PC1512 has no space left for the **Snapshot** files to be created, and will return a rude message if you try to save more than a tiny part of the screen. You can make enough space for **Snapshot** to work on a PC1512 by stripping out the files containing data for the screen fonts, together with the desk accessories other than **Snapshot**. You will find details of the screen font files in the appendices of the *PC1512 User Guide*.

You can transfer a picture from BASIC 2 to GEMPAINT in the following way. First, make sure that the image you want to transfer is on the screen. Next select **Snapshot** from the BASIC 2 menu. A **Snapshot** box should then appear on the screen as in Figure 9.1. If you click on the question mark in this box, details of how to use **Snapshot** are given. Click on the camera icon, and a file requester box appears (Figure 9.2), in which you select a file name to store the image in. The file name chosen must end in the suffix .IMG.
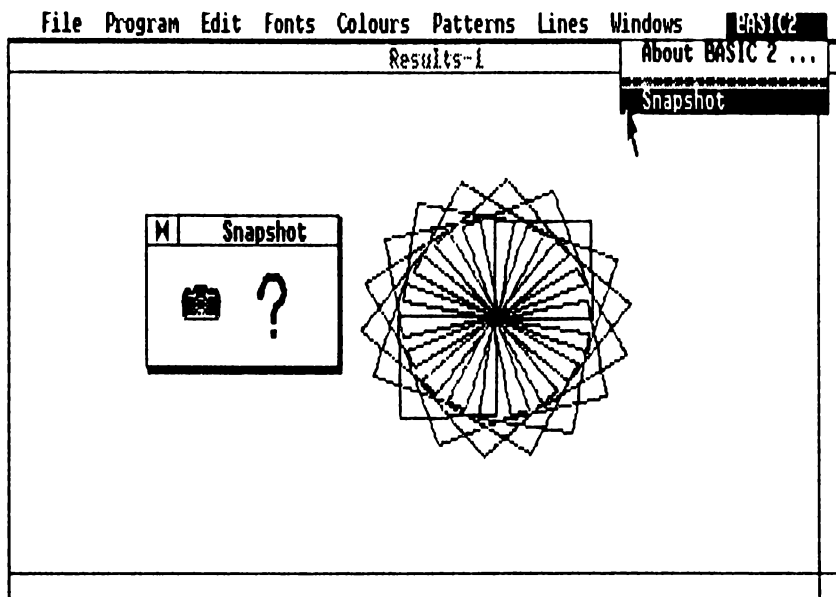
Figure 9.1 Use of the **Snapshot** desk accessory

When you have selected a file name, the file requester box and the **Snapshot** box will disappear, and the arrow-shaped mouse pointer will be replaced by a cross. Position the cross at the upper right hand corner of an imaginary box to contain your image, and press the mouse button down. Keeping the button depressed, move the arrow towards the bottom left hand corner of the screen. You will see a box defining the size of the selected area (Figure 9.3). When the box is large enough, take your finger off the mouse button, and the image file will be created. There are two possible problems at this stage. The first is the dreaded 'not enough memory' alert box, shown in Figure 9.4. If this appears, you must either select a smaller screen area, or strip out more GEM system files.

Results-1

ITEM SELECTOR

Directory: A:\BASIC2*.IMG_____

| M | *.IMG |

Selection: SHAPE .IMG

◆ EXAMPLES.___
◆ PROGRAMS.___
ROB1   .IMG

OK

Cancel
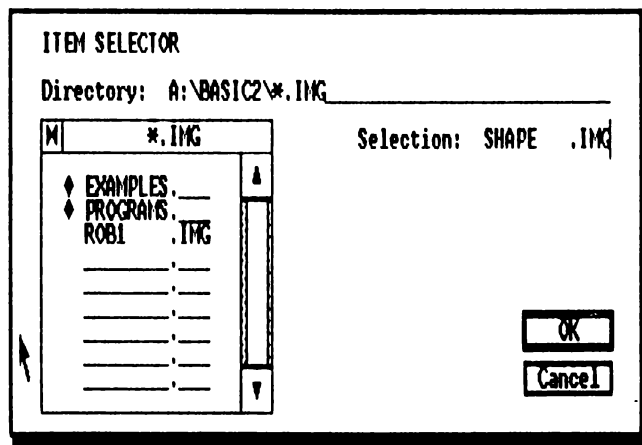
Figure 9.2 The requester box asking for a .IMG file name to save the **Snapshot** file

Results-1
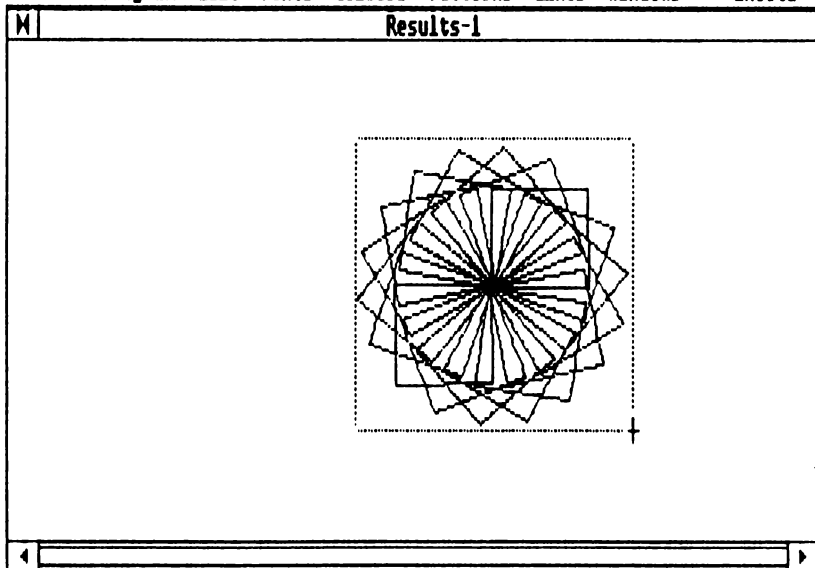
Figure 9.3 The box defining a selected **Snapshot** region
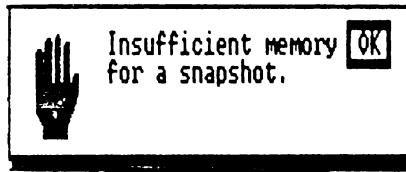
Figure 9.4 A hazard of using BASIC 2 and GEM on a 512K machine

If your disk is too full to accept the created file, you will get the next message. In this case, merely insert a disk with more space on it and try again.

Figure 9.5 No disk space to save the .IMG file

If all has gone well, you will have a .IMG file on disk. This can be directly loaded by GEMPAINT, and the result will be as shown in Figure 9.6. You can then use your artistic creativity on the image. This kind of picture transfer is useful when you have created mathematically precise figures in a BASIC 2 program. Such graphics can be time consuming to draw directly in GEMPAINT. Alternatively, you may wish to transfer business graphs or histograms to embellish with GEMPAINT.

9.5 Program protection

You may be writing programs for other people to use, perhaps students in a class. In such cases, the last thing you want is for the user to meddle with your program: deleting a few lines or adding a line or two of gibberish is a sure way to crash a program. You can stop the user breaking into the program using the **Ctrl-C** keys by adding the following command at the start of the program.

```
OPTION  RUN
```

If you want to restore the ability to break into the program, the command:

`OPTION  STOP`

may be used.
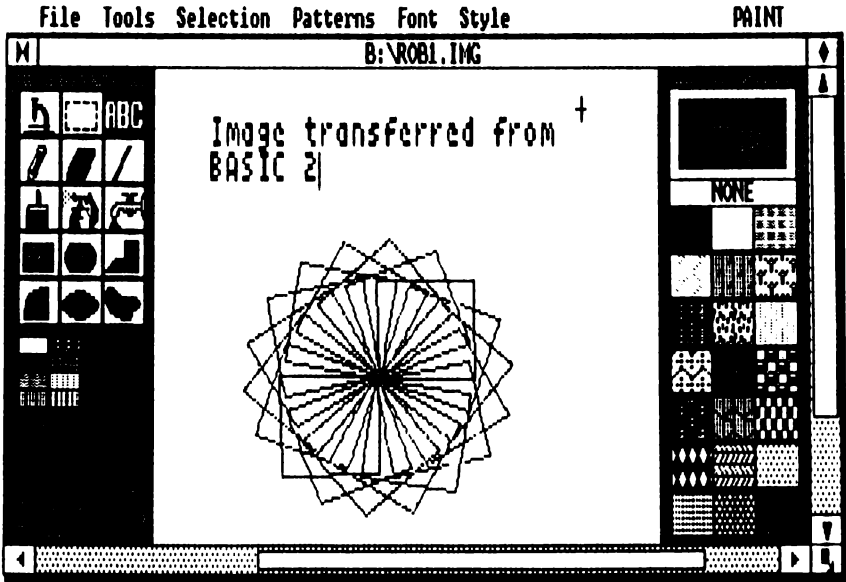
Figure 9.6 The transferred image safe within GEMPAINT

Appendix One

BASIC 2 COMMANDS

BASIC 2 keywords are written here in capital letters.
Variable data is written within angle brackets '< >'
Optional data is written within square brackets '[ ]'
Data items which may be repeated are followed by ......

This appendix is not intended as a full description of the BASIC 2 keywords, but rather as a handy reference guide. For further information you may wish to consult either the *BASIC 2 User Guide* or the forthcoming *Locomotive BASIC 2 Reference Manual*. Note that the commands here are grouped by *function* rather than in pure alphabetical order. This may help you to find the command you want rather more quickly than is possible using the *User Guide* itself. Note that an alphabetical index of BASIC 2 keywords may be found at the end of the book.

A1.1 System commands

These commands are often used outside the program itself, and are normally concerned with the activity of the computer and its action on the program, rather than being part of the program itself. Commands for manipulating disk directories are also included.

Command	Action	Syntax
CD		
	Change directory	CD directory name
CHDIR		
	Change directory	CHDIR directory name
CLEAR		
	Clears all variables, closes all file streams	CLEAR

CLEAR RESET

As CLEAR but also resets   CLEAR RESET
all streams and windows to
the initial state

DEL

Delete files   DEL filename

DIR

Gives a listing of the   DIR [filename]
current directory in
the Dialogue window

EDIT

Enter edit mode   EDIT

ERASE

Delete files   ERASE filename

FILES

Lists the directory on a given   FILES [#stream,] [filename]
stream (default if no stream
specified)

KILL

Erase a given file, including   KILL filename
the filename

MD

Create a directory   MD directory name

MKDIR

Create a directory   MKDIR directory name

NAME

Name a disk file   NAME filename AS filename

NEW

Clears any program currently   NEW
in memory

QUIT

Quit BASIC 2 and return to System	QUIT

RD

Delete the given directory	RD directory name

REN

Rename a disk file	REN oldname , newname

RESET

Reset file system enabling disks to be changed	RESET

RMDIR

Delete the given directory	RMDIR directory name

RUN

Execute program currently in memory	RUN

SYSTEM

Quit BASIC 2 and return to System	SYSTEM

ZONE

Set the print zone size on the Dialogue stream	ZONE integer expression

A1.2 Option commands

These commands allow you to set various general parameters in your programs, for example the units in which angles are measured, or preventing a program from being stopped.

Command Action   Syntax

OPTION CURRENCY$

Set the currency symbol $ for use with the command PRINT USING	OPTION CURRENCY$ string expression

157

OPTION DATE

Set the form of a date string OPTION DATE form
[,separator expression]

OPTION DECIMAL

Set the decimal point and OPTION DECIMAL string
thousands separator for use expression, string expression
in PRINT USING

OPTION DEGREES

Set angle values to degrees OPTION DEGREES

OPTION INKEY$

Set translations for special OPTION INKEY$ string
keys for INKEY$ expression

OPTION RADIANS

Set angle values to radians OPTION RADIANS

OPTION RUN

Prevent a program from OPTION RUN
being halted

OPTION STOP

Reverses action of OPTION OPTION STOP
RUN

OPTION TRAP

Turn on or off trapping of OPTION TRAP logical
undefined values expression

A1.3 Assignment statements

These are used to assign numeric or character values to specific sections or locations in the computer's memory.

Command	Action	Syntax
DATA	Specifies a list of values to be used by the READ statement	DATA <item> [,<item>] ...
LET	Assigns a numeric value or a sequence of characters to a variable	LET <variable> = <expression>
LSET	Set one string to another, left justified	LSET <stringvariable1> = <stringvariable2>
PUT	Send data to a random access or keyed access file	PUT #stream, string expression [LOCK lock] [position]
READ	Take data items from the current position in the data statement and assign each value in turn to the listed variables	READ <variable> [,<variable>]
RESTORE	Reposition the READ pointer to the first item in a specified data statement or if no line is specified to the first item in the first data statement	RESTORE linenumber
RSET	Set one string to another, right justified	RSET <stringvariable1> = <stringvariable2>

SWAP

Swaps the contents of two   SWAP variable1,variable2
variables

A1.4 Control statements

These control the order in which instructions are carried out, either conditionally
or absolutely.

Command Action   Syntax

CONT

Continue after Ctrl-C or   CONT
STOP

END

End of program   END

ERROR

Causes error action to be   ERROR <integer expression>
taken

FOR

Control loop. Marks the   FOR <variable> = <initial
beginning of a section of   value> TO <terminating value>
code that is executed a set   [STEP <increment>]
number of times

GOSUB

Calls a subroutine   GOSUB <line number>

GOTO

Jumps to a given line   GOTO <line number>

IF..THEN..ELSE

If a condition is true, then perform the following statements or jump to specified line

IF logical expression THEN statement [ELSE statement]
or
IF logical expression GOTO line number [ELSE statement]

NEXT

Marks end of FOR control loop

NEXT [<variable>]

ON

Control is passed to another part of the program if a condition is satisfied, for example ...

ON ERROR GO TO

Set an error trap

ON ERROR GO TO linenumber

ON .. GOSUB

Control is passsed to the named subroutine when the condition is true

ON integer expression GOSUB linenumber

ON .. GOTO

Control is passsed to the named linenumber when the condition is true

ON integer expression GOTO linenumber

REPEAT .. UNTIL

Repeat a sequence of commands until the condition given on the until line is true

REPEAT .. UNTIL logical expression

RESUME

Resume normal execution after processing an error

RESUME
or RESUME linenumber
or RESUME NEXT

RETURN

    Return to calling GOSUB   RETURN
    statement

STOP

    Stop execution of the   STOP
    program

WHILE .. WEND

    Control loop in which   WHILE logical expression ...
    execution continues while   WEND
    a condition is true

A1.5 Input/output statements

Command	Action	Syntax
CLOSE	Close files in given streams	CLOSE #stream[,#stream]...
DRIVE	Set the default disk drive	DRIVE <number>
GET	Get a record from a random or keyed file	GET #stream, string variable [position] [LOCK lock]
INPUT	Input data items	INPUT [#stream]
LINE INPUT	Input line	LINE INPUT [#stream]
LPRINT	Print to the line printer	LPRINT
OPEN	Open a file for serial input	OPEN #stream INPUT filename [LOCK lock]

	Open a file for serial output	OPEN #stream [NEW/OLD] APPEND filename
OUTPUT		OPEN #stream [NEW/OLD] filename
	Open a file for random or keyed access	OPEN #stream [NEW/OLD] RANDOM filename [INDEX filename] [LENGTH record length] [LOCK lock]
	Open a window, printer or graphics device	OPEN #stream WINDOW integer expression
		OPEN #stream PRINT integer expression
		OPEN #stream DEVICE integer expression
PRINT		
	Print to a stream (default is **Results-1** window)	PRINT [#stream,] [print specifier] where print specifier can be: AT (column, line) COLOUR (colour) COLOR (colour) EFFECTS (effects bits) TAB (integer expression) FONT (font) POINTS (point size) ANGLE (angle in degrees) MODE (write mode) MARGIN (position) ADJUST (point size) ZONE (zone width)

A1.6 Graphics statements

TURTLE GRAPHICS COMMANDS

BASIC 2 is unique amongst BASIC dialects in that it includes LOGO-like
turtle graphics statements for generating graphics.

Command	Action	Syntax
FD **FORWARD** [#stream,]	Move turtle forward, line drawn unless MOVE is specified	[MOVE] FORWARD [WIDTH line width] [STYLE line style] [COLOUR colour] [MODE write mode] [START end style][END end style]
HEADING	Returns the current heading (angle) of the turtle	HEADING [#STREAM]
LT **LEFT**	Turn turtle left by a given angle	LEFT [#stream,] angle
MOVE FORWARD	Move turtle forward without drawing a line	MOVE FORWARD
POINT	Set turtle pointing in a specific direction	POINT [#stream,] angle
RT **RIGHT**	Turn turtle right by a given angle	RIGHT [#stream,] angle

TOWARD

Returns the bearing of point   TOWARD ([#stream,]x;y)
(x,y) from the turtle

WIMP GRAPHICS COMMANDS

The commands in this section are specific to the GEM-based window and
mouse facilities offered within BASIC 2.

Command Action   Syntax

ALERT

Display an Alert Box on   ALERT icon number TEXT
screen   string expression [,string
                                      expression [,string
                                      expression [,string expression
                                      [,string expression]]]] BUTTON
                                      button spec [,button spec
                                      [,button spec]]

BUTTON

Returns the state of a given   BUTTON [(button)]
mouse button

CLOSE WINDOW

Close a given window   CLOSE WINDOW number

OPEN WINDOW

Open a given window   OPEN WINDOW number

WINDOW CLOSE

Make a window invisible   WINDOW [#stream] CLOSE

WINDOW CURSOR

Make the cursor in a given   WINDOW [#stream] CURSOR
window visible or invisible   logical expression

WINDOW FULL

Expand a window to its   WINDOW [#stream] FULL
full size or reduce it in size   logical expression

165

WINDOW INFORMATION

Display the given text in the window's information bar

WINDOW [#stream] INFORMATION string expression

WINDOW MOUSE

Change the form of the mouse pointer in the given window

WINDOW [#STREAM] MOUSE integer expression

WINDOW OPEN

Make a window visible

WINDOW [#stream] OPEN

WINDOW PLACE

Move a window to the point (xpixel;ypixel)

WINDOW [#stream] PLACE l xpixel;ypixel

WINDOW SCROLL

Scroll a window over its virtual screen to point (x;y)

WINDOW [#stream] SCROLL x;y

WINDOW SIZE

Make a window the given size

WINDOW [#stream] SIZE width,height

WINDOW TITLE

Set the title displayed for the given window

WINDOW [#stream] TITLE string expression

166

GENERAL GRAPHICS COMMANDS

This group of commands comprises the main commands for drawing text and graphics on the screen.

Command Action   Syntax

BOX

Draw a box   BOX [#stream,]x;y,width,height
                                            [ROUNDED] [FILL [ONLY]
                                            [WITH fill style]]
                                            [WIDTH line width]
                                            [STYLE line style]
                                            [COLOUR colour]
                                            [MODE write mode]

CIRCLE

Draw a circle with   CIRCLE [#stream] x;y,radius
centre at (x;y)   [PART start angle,end angle]
                                            [START end style] [END end
                                            style]
                                            [FILL [ONLY] [WITH
                                            fill style]]
                                            [WIDTH line width]
                                            [STYLE line style]
                                            [COLOUR colour]
                                            [MODE write mode]

CLS

Clears a virtual screen and resets the cursor position. Will also reset text and graphics options to the default if RESET is specified.

CLS [#stream] [RESET]

ELLIPSE

Draw an ellipse with centre (x;y)

ELLIPSE
[#stream,]x;y,radius,aspect
[PART start angle,end angle]
[START end style]
 [END end style]
[FILL [ONLY]
 [WITH fill style]]
[WIDTH line width]
[STYLE line style]
[COLOUR colour]
[MODE write mode]

ELLIPTICAL PIE

Draw a segment of an ellipse with centre (x;y)

ELLIPTICAL PIE
[#stream,]x;y,radius,aspect,start angle, end angle
[FILL [ONLY] [WITH fill style]]
[WIDTH line style]]
[STYLE line style]
[COLOUR colour]
[MODE write mode]

FLOOD

Paint a bounded area containing the point (x;y)

FLOOD [#stream]x;y [,limit]
[COLOUR colour] [MODE write mode] [FILL WITH fill style]

GRAPHICS

Change the current graphics style

GRAPHICS [#STREAM,]
[CURSOR integer expression]

[COLOUR colour]
[MARKER SIZE size]
[MARKER marker style]
[[LINE] WIDTH line width]
[[LINE] STYLE line style]
[FILL[STYLE][WITH] fill style]
[[LINE] START end style]
[[LINE] END end style]
[MODE write mode]

GRAPHICS UPDATE

Update printer or plotter streams

GRAPHICS [#stream,] UPDATE [NEW]

LINE

Draw one or more straight lines through a series of points

LINE [#stream,] x;y [,x;y] ... [START end style] [END end style] [WIDTH line width] [STYLE line style] [COLOUR colour] [MODE write mode]

LOCATE

Move the cursor to a given line and column of the virtual screen

LOCATE [#stream,][column][;line]

MOVE

Move the cursor to point (x;y) in a graphics screen

MOVE [#stream,] x;y

PIE

Draw a circle segment centered at (x;y)

PIE [#stream] x;y,radius,start angle,end [FILL [ONLY] [WITH fill style]] [WIDTH line width] [STYLE line style] [COLOUR colour] [MODE write mode]

PLOT

Plot one or more points

PLOT [#stream,] x;y [,x;y] ...
[MARKER marker type]
[STYLE line style]
[COLOUR colour]
[MODE write mode]

SCREEN

Define a virtual screen

..as a graphics screen
SCREEN [#stream]
GRAPHICS [width
[FIXED], height [FIXED]]
[MINIMUM width,height]
[MAXIMUM width,height]
[UNIT width,height]
[INFORMATION logical
expression]
..as a text screen
SCREEN [#stream] TEXT
[FLEXIBLE][width [FIXED],
height [FIXED]]
[MINIMUM width,height]
[MAXIMUM width,height]
[UNIT width,height]
[INFORMATION logical
expression]

SET

Set the default text
properties

SET [#stream] [WRAP logical
expression] [COLOUR colour]
[EFFECTS effects bits]
[FONT font]
[POINTS point size]
[ANGLE angle in degrees]
[MODE write mode]
[MARGIN position]
[ZONE zone width]

SHAPE

Draw a polygon with
at least three points

SHAPE [#stream] x;y [,x;y] ...
[FILL [ONLY][WITH fill style]
[WIDTH line width]
[STYLE line style]
[COLOUR colour]
[MODE write mode]

TEXT

Clear part/all of screen

TEXT [#screen] CLEAR

Delete a line or character
from a text screen

TEXT [#screen] DELETE
[LINE}

Insert a line into a text screen

TEXT [#screen] INSERT LINE

Move the cursor up or down
a given number of lines

TEXT [#screen] FEED integer
expression

USER ORIGIN

Set the graphics screen
origin to (x;y)

USER [#stream,] ORIGIN x;y

USER SPACE

Set the graphics screen user
coordinate space

USER [#stream,] SPACE size

A1.7 File handling commands

This category includes commands for manipulating files and records.

Command Action   Syntax

ADDKEY

Add a new key for an existing
record in a keyed file

ADDKEY #stream KEY
expression [INDEX index no.]
[LOCK lock]

ADDREC

Add a new record and key to a keyed file

ADDREC #stream, string expression KEY expression [INDEX index no.] [LOCK lock]

CONSOLIDATE

Mark keyed file as consistent

CONSOLIDATE (#stream)

DELKEY

Delete a key for a record in a keyed file

DELKEY #stream [,position][LOCK lock]

DISPLAY

Display contents of a file

DISPLAY [#stream] string expression

KEYSPEC

Define an index for a keyed file

KEYSPEC #stream [,] INDEX numeric expression [key type] [UNIQUE logical expression]

LOCK

Change a record lock

LOCK #stream [POSITION] [LOCK lock] *or* LOCK #stream [POSITION] [,lock]

POSITION

Set current position in a random or keyed file

POSITION #stream, position [LOCK lock]

REPOSITION

Change the current position to another position for the current record

REPOSITION #stream, position [LOCK lock]

A1.8 Miscellaneous commands

Command	Action	Syntax
DEF	Define an expression for a function	DEF FN name [(parameters)] = expression
DIM	Declare the dimensions of an array	DIM (constant[,constant ...]) **or** DIM (constant1 TO constant2[,constant...])
LABEL	Define a statement label	LABEL name
RECORD	Set a record structure	RECORD name; field1 [,field2...]
REM	Remark line. The whole of the line after the REM statement is ignored. A single quote character may also be used (')	REM message
STREAM	Set the number of the default stream	STREAM #stream
TYPE	List file to the **Dialogue** window	TYPE filename

Appendix Two

BASIC 2 FUNCTIONS

BASIC 2 keywords are written here in capital letters.
Variable data is written within angle brackets '< >'
Optional data is written within square brackets '[ ]'
Data items which may be repeated are followed by **......**

Functions are rules relating sets of values to each other. The value in the first set is known as the argument, and the value in the second set is the result.

A2.1 Numerical and mathematical functions

This group includes functions of use in science, technology and mathematics.

Function	Result	Syntax
ABS		
	Gives the absolute value of a given number	ABS <numeric expression>
ACOS		
	Arc cosine. Returns the angle from a given cosine	ACOS <numeric expression>
ASIN		
	Arc sine. Returns the angle from a given sine	ASIN <numeric expression>
ATAN **ATN**	Arc tangent. Returns the angle of which the tangent is a given value	ATAN <numeric expression>

CEILING

Returns the given value rounded **up** to an integer

CEILING <numeric expression>

CINT

Converts a value to the nearest 32 bit signed integer

CINT <numeric expression>

COS

Returns the cosine of a given angle

COS <numeric expression>

DEG

Returns the given angle converted from radians to degrees

DEG <numeric expression>

EXP

Return the exponential of a numeric value

EXP <numeric expression>

FIX

Return the given number as an integer (rounded towards 0)

FIX <numeric expression>

FRAC

Returns the fractional part of a given number

FRAC <numeric expression>

HEX$

Converts a number into a string of hexadecimal digits

HEX$ <integer expression [,field size]>

INT

Returns the given value rounded **down** to an integer

INT <numeric expression>

LOG

Converts a value to its natural logarithm

LOG <numeric expression>

LOG10

Converts a value to the log base 10

LOG10 <numeric expression>

LOWER

Gives the lower bound of an array dimension

LOWER (array name () <integer expression>)

MAX

Determines a maximum value

MAX (numeric expression [,numeric expression ...])

MIN

Determines a minimum value

MIN (numeric expression [,numeric expression ...])

RAD

Returns an angle in radians

RAD (numeric expression)

RND

Random number generator

RND [(integer expression)]

ROUND

Rounds a value to a given number of decimal places

ROUND (numeric expression [,places])

SGN

Returns the sign of a given value

SGN (numeric expression)

SIN

Gives sine of an angle

SIN (numeric expression)

SQR

Gives the square root of a positive number

SQR (numeric expression)

TAN

Returns the tangent of a given number

TAN (numeric expression)

UPPER

	Gives the upper bound of an array dimension	UPPER (array name () <integer expression>)

A2.2 String functions

These functions are used for manipulating characters and text

Function	Result	Syntax
ASC		
	Returns the ASCII value of a given character	ASC (string expression)
BIN$		
	Converts a number into a string of binary digits	BIN$ (integer expression [,field width])
CHR$		
	Returns the character whose ASCII code is specified	CHR$ (integer expression)
DATE$		
	Returns the date as a string in the current form	DATE$
DEC$		
	Returns the formatted string representation of the given value	DEC$ (numeric expression, format template)
INSTR		
	Returns the position of a substring in a given string	INSTR ([integer expression,] searched string, searched-for string)

LEFT$

Return the left hand part
of a given string

LEFT$ (string
expression,integer expression)

LEN

Returns the length of the
given string

LEN (string expression)

LOWER$

Gives the string converted
to lower case

LOWER$ (string expression)

MID$

Return part of a string

MID$ (string expression, start
position [,substring length])

RIGHT$

Returns the right hand part
of the given string

RIGHT$ (string expression,
integer expression)

STR$

Converts a number into
a string of digits

STR$ (numeric expression)

STRING$

String of a specific character

STRING$ (integer expression,
character expression)

UPPER$

Converts a string to
upper case

UPPER$ (string expression)

VAL

Return a numeric value
equivalent to the given
string

VAL (string expression)

`WHOLE$`

Return the whole of a
fixed length string, not
including any nulls
WHOLE$ (string expression)

A2.3 System functions

This group of functions allow the user to look at the state of different aspects of
the computer system. Graphics functions are not included here, and are listed in
Section A2.4 below.

Function	Result	Syntax
`CHDIR$`		
	Gives the current directory for a given drive	CHDIR$ [(drive string)]
`CURRENCY$`		
	Return current currency string	CURRENCY$
`EOF`		
	End of file test	EOF (#stream)
`ERR`		
	Error number	ERR
`ERROR$`		
	Return the error message for the given error number	ERROR$ (error number)
`FIND$`		
	Search for a given file and return its exact filename	FIND$ (filename expression)

FONT$

Return the name of the font
represented by a given
font number

FONT$ ([#stream,],font)

FRE

Returns the amount of free
memory on the computer

FRE

OSERR

Gives an indication of an
operating system dependent
error

OSERR

TIME

Return the time (in 1/100ths
of seconds since 12.00am)

TIME

VERSION

Determines which version
of BASIC 2 is in use

VERSION (integer expresion)

A2.4 Graphics functions

The graphics functions enable the user to look at various graphics parameters:
the position of the cursor, the windows and so on.

Function   Result   Syntax

ATAN2

Returns the bearing of the
point (x;y) from the origin

ATAN2 (x,y)

EXTENT

Gives the distance in user
coordinates moved by an
equivalent PRINT command

EXTENT ([#stream] [print
function] print string)

POINTSIZE

Gives the nearest smaller
point size of the given font
to that requested

POINTSIZE ([#stream,] font,
pointsize)

POS

Returns the current character
position of the cursor

POS (#stream)

TEST

Return the colour displayed
at the point (x;y) in a
graphics screen

TEST ([#stream,] x;y)

VPOS

Gives the row number of the
line on which the cursor is
positioned

VPOS [(#stream)]

XACTUAL

Gives the actual width of a
window in pixels

XACTUAL [(#stream)]

XBAR

Gives the width of the RH
window border in pixels

XBAR [(#stream)]

XCELL

Gives the width of a character
cell in user coordinates

XCELL [(#stream)]

XDEVICE

Returns the width of the
device in pixels

XDEVICE [(#stream)]

XMETRES

Returns the width of the
device in metres

XMETRES [(#stream)]

XMOUSE

Returns the X coordinate of the position of the mouse on the screen in pixels

XMOUSE

XPIXEL

Returns the width of a pixel in user coordinates

XPIXEL [(#stream)]

XPLACE

Returns the X coordinate of the position of the window on the screen in pixels

XPLACE [(#stream)]

XPOS

Gives the X coordinate of the cursor position in user coordinates

XPOS [(#stream)]

XSCROLL

Gives the X coordinate of the position of the window on the virtual screen

XSCROLL [(#stream)]

XUSABLE

Gives the usable width of the screen in pixels

XUSABLE

XVIRTUAL

Returns the width of a graphics screen in user coordinates

XVIRTUAL [(#stream)]

XWINDOW

Returns the width of the window in pixels

XWINDOW [(#stream)]

YACTUAL

Gives the actual height of a window in pixels

YACTUAL [(#stream)]

YASPECT

Returns the aspect ratio of the user coordinates, y to x

YASPECT [(#stream)]

YBAR

Gives the height of the bottom window border in pixels

YBAR [(#stream)]

YCELL

Gives the height of a character cell in user coordinates

YCELL [(#stream)]

YDEVICE

Returns the height of the device in pixels

YDEVICE [(#stream)]

YMETRES

Returns the height of the device in metres

YMETRES [(#stream)]

YMOUSE

Returns the Y coordinate of the position of the mouse on the screen in pixels

YMOUSE

YPIXEL

Returns the height of a pixel in user coordinates

YPIXEL [(#stream)]

YPLACE

Returns the Y coordinate of the position of the window on the screen in pixels

YPLACE [(#stream)]

YPOS

Gives the Y coordinate of the cursor position in user coordinates

YPOS [(#stream)]

YSCROLL

Gives the Y coordinate of the position of the window on the virtual screen

YSCROLL [(#stream)]

YUSABLE

Gives the usable height of the screen in pixels

YUSABLE

YVIRTUAL

Returns the height of a graphics screen in user coordinates

YVIRTUAL [(#stream)]

YWINDOW

Returns the height of the window in pixels

YWINDOW [(#stream)]

A2.5 Input/output functions

These functions are used with I/O operations

Function	Result	Syntax

INKEY

Gives the state of a key on the keyboard

INKEY

INKEY$

Returns the equivalent character if a key is typed at the keyboard

INKEY$

INPUT

Input data from the keyboard

INPUT [#stream,] [AT (x;y)] [;] [prompt;] <variable>...[;]

INPUT$

Input a string of fixed length   INPUT$ ([#stream,] integer expression)

A2.6 File handling functions

This group of functions is used in accessing files.

Function Result   Syntax

KEY

Return current key value   KEY (#stream)
in keyed file

KEY$

KEY$ (#stream)

LOC

Return the record number of   LOC (#stream)
the current location in the
given file

LOF

Return the length of the   LOF (#stream)
given file

UNIQUE

Returns a unique value for   UNIQUE (#stream)
a given file

Appendix Three

BASIC 2 CHARACTER CODES

The GEM environment offers you a choice of up to 192 different characters, with codes numbered from 1 - 193, and from 224 - 255. The latter set of symbols includes a number of Greek characters, useful for scientific and technical work. Not all of the characters listed on the following pages are available in all versions of BASIC 2, and the number depends on the version of GEM being used. Note that the characters may not all be available on your printer, even though they are available on the screen. In general, common symbols, upper and lower case characters and numerals will be the same on the printer and the screen.

You can access many of the characters directly from the keyboard. Those that are not available can be printed using BASIC 2 functions. Here is a short program that will print all of the characters available to your computer.

The characters available in the three fonts available on the Amstrad PC1512 (system, Swiss and Dutch) are shown in Figures A3.1 - A3.3, which are screen dumps from the character generating program. As you can see, the system font offers you the Greek characters in preference to symbols 1 - 32.

```
REM program to print available characters
CLS
WINDOW #1 OPEN
WINDOW #1 FULL
WINDOW #1 TITLE "characters available
     INPUT "which font number";n
     INPUT "which font size";sz
CLS
     PRINT "font number = ",n
     SET FONT n
     SET POINTS sz
'set character counter at 0
char = 0
```

```
FOR i = 1 TO 9
    PRINT
    FOR j = 1 TO 30
        char = char + 1
        PRINT CHR$(char) + " ";
        IF char = 255 THEN GOTO jump
    NEXT j
NEXT i
LABEL jump
    WHILE BUTTON = -1:WEND
STOP
END
```
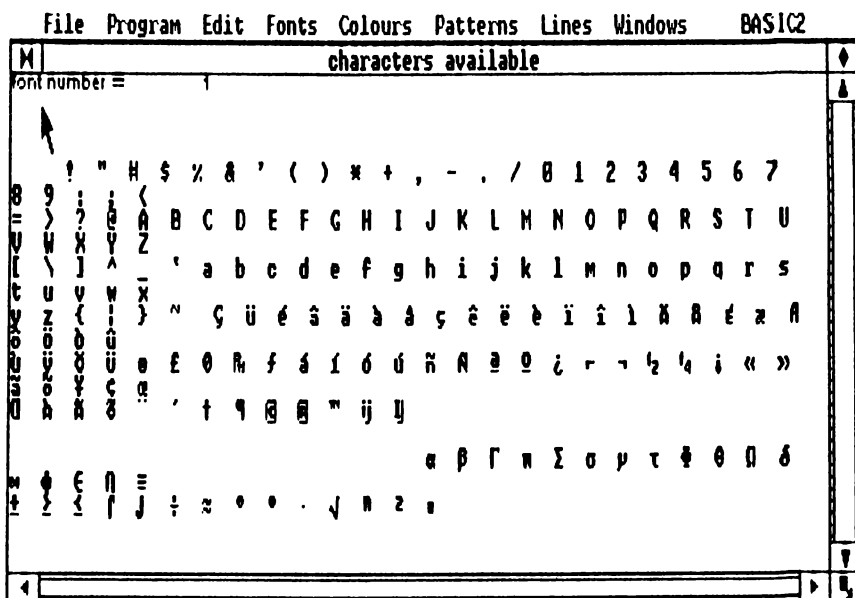

Figure A3.1 Character appearance in the system font

characters available

font number =   2

` ! " # § % & ' ( ) ' + , - . / 0 1 2 3 4 5 6 7 8 9 : ; <
= > ? @ A B C D E F G H I J K L M N O P Q R S T U V W
X Y Z
[\ ] ^ _ ` a b c d e f g h i j k l m n o p q r s t u v w x
y z (|) ~ Ç ü é â ä à å ç ê ë è ï î ì Ä Å É æ Æ ô ö ò û
ù ÿ Ö Ü ø £ Ø × ƒ á í ó ú ñ Ñ ª º ¿ ⌐ ¬ ½ ¼ ¡ « » ▪ ░ ▒ ¥ ¢
œ
Œ À Å Õ ¨ ´ † ¶ © ® ™ ij IJ
                β

characters available

font number =   3

` ! " # § ¾ & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 ! ; <
= > ? @ A B C D E F G H I J K L M N O P Q R S T U V
W X Y Z
[\ ] ^ _ ` a b c d e f g h i j k l m n o p q r s t u v w x
y z (|) ~ Ç ü é â ä à å ç ê ë è ï î ì Ä Å É æ Æ ô ö ò û
ù ÿ Ö Ü ø £ Ø × ƒ á í ó ú ñ ñ ª º ¿ ⌐ ¬ ½ ¼ ¡ « » ã õ ¥
¤ œ
Œ À Å Õ ¨ ´ † ¶ © ® ™ ij IJ ¢ ¥
             α β Γ π Σ σ μ τ φ θ Ω δ ∞ φ ∈ ∩ ≡
± ≥ ≤ ⌠ ⌡ ÷ ≈ ° • · √ ⁿ ² ∎

Figure A3.2-A3.3 Character appearance in Fonts 2 (top) and 3 (bottom)

Symbol	Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol	Code
□	0	◄	17	"	34	3	51	D	68
↑	1	⋈	18	#	35	4	52	E	69
↓	2	■	19	$	36	5	53	F	70
→	3	¶	20	%	37	6	54	G	71
←	4	§	21	&	38	7	55	H	72
■	5	↕	22	'	39	8	56	I	73
◣	6	□	23	(	40	9	57	J	74
◆	7	↑	24	)	41	:	58	K	75
✓	8	↓	25	*	42	;	59	L	76
◔	9	→	26	+	43	<	60	M	77
□	10	←	27	,	44	=	61	N	78
♪	11	└	28	-	45	>	62	O	79
▲	12	□	29	.	46	?	63	P	80
▼	13	▲	30	/	47	@	64	Q	81
►	14	▼	31	0	48	A	65	R	82
◄	15	□	32	1	49	B	66	S	83
►	16	!	33	2	50	C	67	T	84

U	85	f	102	w	119	ê	136	Ö	153
V	86	g	103	x	120	ë	137	Ü	154
W	87	h	104	y	121	è	138	ø	155
X	88	i	105	z	122	ï	139	£	156
Y	89	j	106	{	123	î	140	Ø	157
Z	90	k	107	|	124	ĩ	141	P_t	158
[	91	l	108	}	125	Ä	142	ƒ	159
\	92	m	109	~	126	Å	143	á	160
]	93	n	110	Δ	127	E	144	í	161
^	94	o	111	Ç	128	æ	145	ó	162
_	95	p	112	ü	129	Æ	146	ú	163
'	96	q	113	e	130	ô	147	ñ	164
a	97	r	114	â	131	ö	148	Ñ	165
b	98	s	115	ä	132	ò	149	ª	166
c	99	t	116	à	133	û	150	º	167
d	100	u	117	å	134	ù	151	¿	168
e	101	v	118	ç	135	ÿ	152	⌐	169

¬	170	+	187		204		221	∈	238
¹₂	171	q	188		205		222	⌒	239
¹₄	172	©	189		206		223	≡	240
¡	173	®	190		207	α	224	±	241
«	174	™	191		208	β	225	≥	242
»	175	ij	192		209	Γ	226	≤	243
ã	176	IJ	193		210	π	227	⌠	244
õ	177		194		211	Σ	228	⌡	245
¥	178		195		212	σ	229	÷	246
¢	179		196		213	μ	230	≈	247
œ	180		197		214	τ	231	°	248
Œ	181		198		215	Φ	232	·	249
À	182		199		216	θ	233	•	250
Ā	183		200		217	Ω	234	√	251
Ō	184		201		218	δ	235	ⁿ	252
"	185		202		219	∞	236	²	253
'	186		203		220	φ	237	■	254

Appendix Four

BASIC 2 ERROR CODES AND MESSAGES

The following error codes and messages occur in BASIC 2. See Chapter 9 for a description of error handling.

Error code	Message
1	Internal error
2	Command only valid in a program
3	Command not valid in a program
4	Syntax error This is the most common form of error, and will occur if you make a typing mistake in your program, or try to use a command wrongly
5	Invalid numeric constant
6	Memory full No space left in the computer's memory. Your program may be too large. Check the space occupied by arrays and records. Alternatively, there may be too many nested FOR or GOSUB statements in effect.
7	BASIC stack overflow
8	Too many points specified

9	Type mismatch
	Perhaps you tried to assign a string value to a numeric variable, or *vice versa*.

10   Repeated or invalid keyword

11   Too many or too few subscripts
            You have tried to reference an array that was defined with more (or less) than the number of subscripts in the statement at which the error occurred.

12   Line too long

13   Name too long
            A variable name with more than 40 characters has been chosen.

14   Name already in use

15   Not a valid saved program

16   Cannot continue
            Is a program running? Has an error occurred?

17   Array already defined
            After an array was dimensioned, another dimension statement for the same array was found.

18   Record already defined
            A similar error to 17

19   Line does not exist
            You have tried to GOTO or GOSUB to a line that is not in the program.

20   Label does not exist
            Similar to 19, but it is a LABEL statement that is missing.

21   Unexpected RETURN
            A RETURN statement has been encountered without a matching GOSUB statement.

22	Unexpected ELSE
	An ELSE command has been reached outside of an IF THEN statement.

23	Unexpected FI

24	Unexpected CEND

25	Unexpected FEND

26	Unexpected PEND

27	NEXT missing
	A FOR statement exists in your program for which there is no matching NEXT statement.

28	Unexpected NEXT or does not match FOR variable. The opposite to 27. No FOR statement is present to match the NEXT statement.

29	RESUME missing

30	Unexpected RESUME

31	WEND missing
	You have a WHILE statement without a matching WEND.

32	Unexpected WEND
	You have a WEND statement without a matching WHILE.

33	UNTIL missing
	You have a REPEAT statement without a matching UNTIL.

34	Unexpected UNTIL
	You have an UNTIL statement without a matching WHILE.

35	CASE missing

36	Unexpected CASE

| 37 | Cannot RESUME NEXT |
| | A resume statement was encountered without error trapping being in operation. |

| 38 | Unsuitable command for error trap |

| 39 *et seq* | Unknown error |
| | You will not come across this error as you program, but it may occur if you try to use error trapping wrongly in your programming. |

Appendix Five

SCREEN AND KEY MAPS

The Screen (units = pixels)
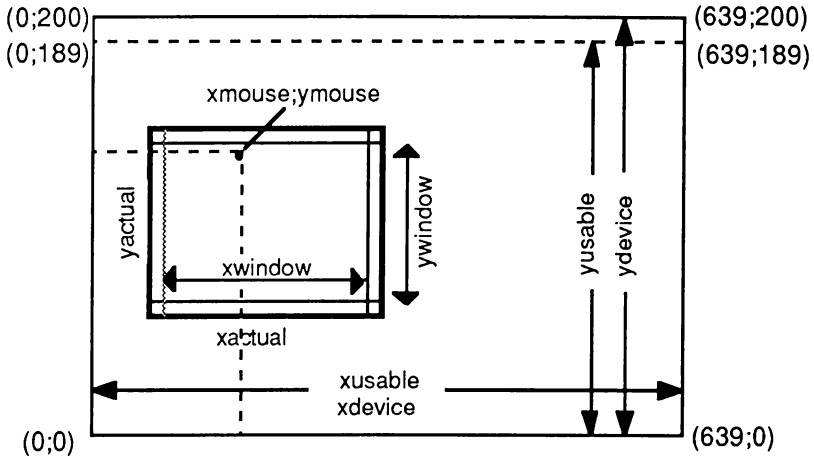
The Virtual Screen (units = user coordinates)

Mark start or end of program section	Clear marked program section
	f1 **f2**
Insert a marked section at the cursor position	Move a marked section from current position to the cursor position
	f3 **f4**
Remove a marked section of the program	**f5** **f6** — Not used
Continue a stopped program at the point it was stopped	**f7** **f8** — Not used
	f9 **f10**
Run the current program	Enter the Editor to edit the program currently in memory

Facilities offered by the function keys in BASIC 2

SUBJECT INDEX

INDEX OF BASIC 2 KEYWORDS

Page references in lighter type indicate examples of keyword usage or in-text definitions. Only a representative selection of references to common keywords (eg **FOR, TO** and so on) are included Page references in bold type indicate entries in Appendices 1 or 2.

YPIXEL	127,**184**
YPLACE	127,**184**
YPOS	108,**184**
YSCROLL	**185**
YUSABLE	**185**
YVIRTUAL	78,79,**185**
YWINDOW	**185**
ZONE	**157**

An Invitation

Sigma Press is still expanding—and not just in computing, for which we are best known. Our marketing is handled by John Wiley and Sons Ltd, the UK subsidiary of a major American publisher. With our speed of publication and Wiley's marketing skills, we can make a great success of your book on both sides of the Atlantic.

Currently, we are looking for new authors to help us to expand into many exciting areas, including:

Laboratory Automation
Communications
Electronics
Professional Computing
New Technology
Personal computing
Artificial Intelligence
General Science
Engineering Applications

If you have a practical turn of mind, combined with a flair for writing, why not put your talents to good use? For further information on how to make a success of your book, write to:

Graham Beech, Editor-in-Chief, Sigma Press,
98a Water Lane, Wilmslow, Cheshire SK9 5BB
or, phone 0625-531035