

The Beginners Guide to STOS Basic

*A complete programming course
for the Atari ST/STE*

MT Software

The Beginners Guide to STOS Basic

Published by: MT Software
Greensward House
The Broadway
Totland, IOW, PO39 0BX

ISBN: 0 9520730 0 5

AUTHOR: Mark B Thomson

© 1993 MT SOFTWARE. All Rights Reserved

This course and the accompanying software are copyright. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the permission of MT Software.

All trademarks acknowledged

CONTENTS

	- WELCOME	V
CHAPTER 1	- GETTING STARTED	1
CHAPTER 2	- A SPLASH OF COLOUR	33
CHAPTER 3	- NUMERIC VARIABLES	47
CHAPTER 4	- NUMERIC FUNCTIONS	57
CHAPTER 5	- MAKING DECISIONS	83
CHAPTER 6	- LOOPS & PROGRAM CONTROL	89
CHAPTER 7	- GUESS THE NUMBER GAME	111
CHAPTER 8	- MATHS QUIZ GAME	121
CHAPTER 9	- STRING VARIABLES	127
CHAPTER 10	- USER INPUT	141
CHAPTER 11	- GUESS THE WORD GAME	173
CHAPTER 12	- WINDOWS & CHARACTER SETS	185

CHAPTER 13	- GRAPHICS	219
CHAPTER 14	- THE SCREEN	255
CHAPTER 15	- BONK THE GONK GAME	313
CHAPTER 16	- CREATING AN ART PACKAGE	335
CHAPTER 17	- SPRITES & ANIMATION	375
CHAPTER 18	- GUNS, ALIENS AND COLLISIONS	395
CHAPTER 19	- SHOOT THE SPOOK GAME	411
CHAPTER 20	- ALIEN ATTACK GAME	431
CHAPTER 21	- MUSIC & SOUND EFFECTS	461
CHAPTER 22	- FILE MANAGEMENT	487
CHAPTER 23	- CREATING A WORD PROCESSOR	515
CHAPTER 24	- CREATING A DATABASE	535
CHAPTER 25	- MENUS & DIALOGUE BOXES	569
CHAPTER 26	- AND FINALLY	599
	- INDEX	605

WELCOME

WELCOME

Welcome to the Beginners Guide to STOS Basic and to the fascinating world of computer programming.

The Beginners Guide to STOS Basic introduces the reader in a step-by-step fashion to what is one of the most powerful and versatile programming languages available for the Atari ST range of computers - *STOS BASIC*.

The course takes the newcomer from basic principles through to the development of complete programs. A practical approach is adopted throughout with the emphasis being on using the computer rather than wading through reams of theory. Programming should be fun, and with this in mind, the course tries to cover as many interests as possible. It does not matter whether you are retired or still at school there is something for everybody. You will learn how to produce shoot-em-up games, art programs, junior educational programs, GCSE mathematics programs, databases, word processors, musical programs and much much more.

The reader should by the end of the course, if he or she has studied diligently, be able to program with some degree of proficiency.

THE DISKS

The course comes with two double-sided 3.5" diskettes. If you have a single sided disk drive then return the disks direct to MT Software for replacement. The disks contain all of the example programs, sprite, screen and music data presented within the course. The programs can thus be loaded directly from disk as you work through the course.

Disk 1 contains all the data for chapters 1-15

Disk 2 contains all the data for chapters 16-26

Before using the disks you should make a backup copy and keep the originals in a safe place. If you are unsure of how to copy disks refer to the ST owners manual for full details.

PROGRAMMING STYLE

There are no hard and fast rules regarding the development and coding of programs and the authors style of programming may differ quite considerably from yours or any other programmer. Ask two programmers to create a database and the resultant programs will be completely different in both style and operation. As you work through the course, look at the example programs and think of alternative ways in which they could be written. Maybe you could write the same program using less commands, maybe your version takes more commands but is easier to understand.

Provided the program works it really does not matter how it is written - do not be afraid to experiment.

COPYRIGHT NOTICE

The programs presented throughout this course and the data on the accompanying disks are copyright MT Software. Owners of this course are granted permission to include any of the code within their own work, but the copyright remains with MT Software and such work may not be published without permission.

WELCOME TO THE WORLD OF PROGRAMMING

Chapter 1

Getting Started

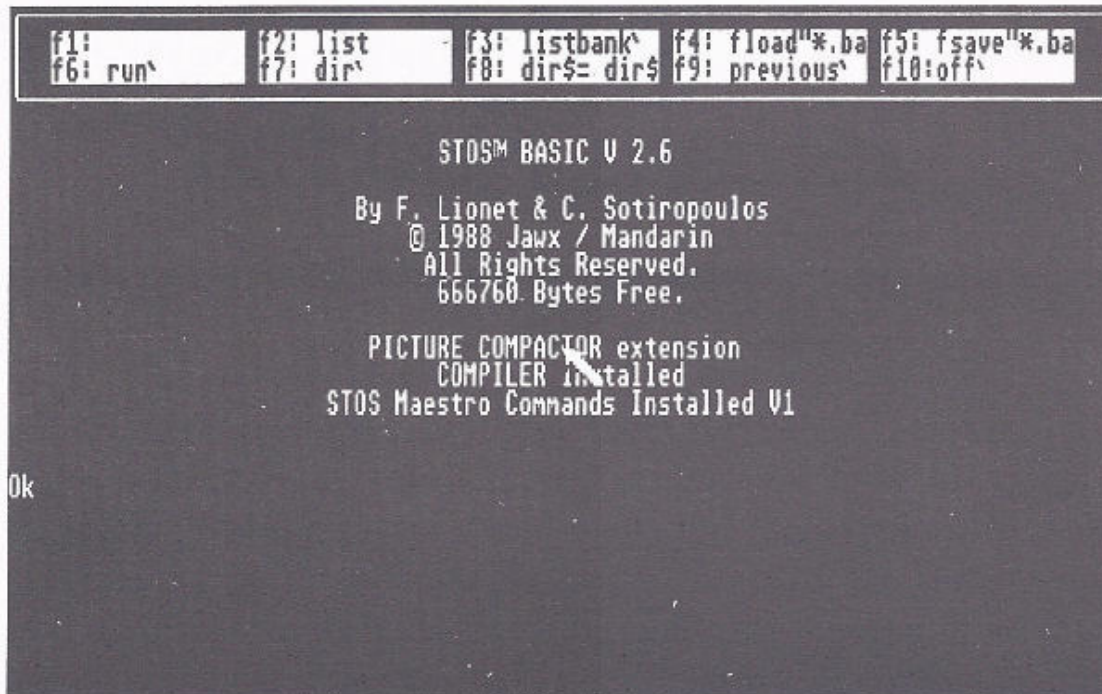
These first few chapters introduce some basic principles and concepts that apply to, and are indeed essential for, the creation of any program. It is important that the topics covered are fully understood before moving on to more advanced sections. It is easy to jump straight on to later chapters but a little time spent studying the basics really will prove rewarding.

GETTING STARTED

STOS consists of a ring bound manual and three disks which are labelled LANGUAGE, ACCESSORIES and GAMES. The games disk contains three demonstration games written in STOS basic. You may already have taken a look at these, and by the end of the course, you will be able to work out exactly how they work. The ACCESSORY disk contains a number of useful extras which we shall look at in due course. Our main concern for now is the LANGUAGE disk as this contains the main STOS Basic interpreter which is required to write programs.

Before continuing you should make a backup copy of all three disks. Use the copies and keep the originals in a safe place.

A disk containing a copy of STOS Basic was given away free with one of the computer magazines and if you obtained your copy from here it should be set up as described on the disk.



To load STOS, place the language disk in to the disk drive and switch on the computer. STOS automatically loads, and after the initial title screen, displays the editor screen as shown above. This is where we enter, edit and generally work on programs.

MOVING AROUND THE EDITOR

Looking along the top of the computers keyboard you will see a row of keys numbered F1-F10. The other keys on the keyboard represent individual letters or characters but the *function keys* can represent complete words. This is handy when programming as it allows us to

enter commands with one key press. The box at the top of the screen is known as the function key window and shows the commands that are currently assigned to the function keys. A particular function may be selected by clicking on the box with the mouse or by pressing the key direct. If you press the shift key or click the right mouse button, the display changes to show the contents of F11-F19. The programmer can change the contents of these keys as we shall see later.

Displayed below the function key box is the version of STOS, the amount of free memory available and a list of any extra accessories that are loaded. Accessories shall be covered later so do not worry about this yet.

Half way down the screen is the word Ok, and underneath this, a small line known as the *text cursor*. As information is typed, the text cursor moves along the line indicating the position of the next character. The text cursor can be moved up, down, left and right using the cursor keys. These are the keys with the arrows on so try pressing them to move the cursor.

The editor works in two modes - insert and replace. Press the INSERT key whilst watching the text cursor and you will see that it changes size. The small cursor indicates *replace* mode with any entered text replacing existing information that may be displayed. The larger cursor indicates *insert* mode and moves any text to the right of the cursor to make way for the new information. We can switch between the two modes by pressing the INSERT key. Let us see how this works.

Suppose we want to type the word *print*, but by accident, we type *prent*.

Enter the following:

prent [and press the RETURN key]

The computer will display a message stating SYNTAX ERROR as it does not understand the entered information. Do not worry about this for now as we are only interested in the operation of the editor at this stage.

The letter "e" needs to be replaced by the letter "i" so move the cursor over the offending letter using the cursor keys. The letter is to be replaced so ensure that the editor is in replace mode (small cursor) and adjust this if necessary using the INSERT key. Now press the letter "i" and it should replace the letter "e". Although the change has been made visually, it has to be entered in to the computers memory. Do this by pressing the RETURN or ENTER key. The ENTER key is found at the bottom of the numeric keyboard and has exactly the same function as the RETURN key.

Suppose now that you have typed *prnt* in error for *print*.

Enter the following:

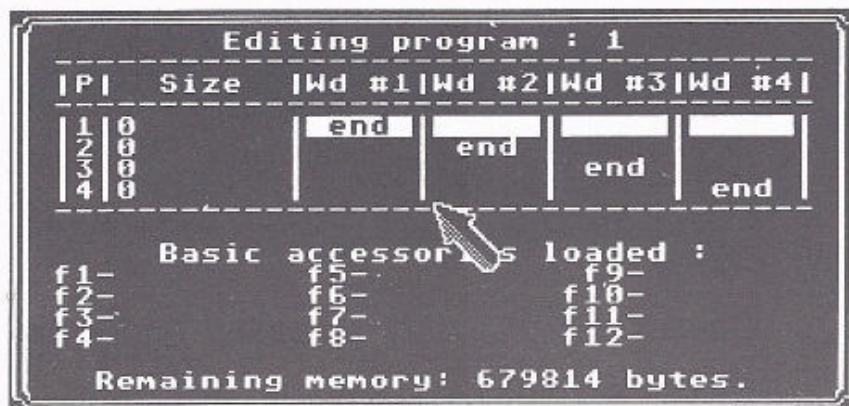
prnt [and press the Return key]

This time the editors insert mode is required to insert the letter "i" between the existing letters "r" and "n". First the text cursor must be positioned over the letter occupying the position where the new letter will go - in this case the cursor must be positioned over the letter 'n'. We have already seen how the cursor keys are used to position the cursor but the mouse can also be used. Point to the letter "n" with the mouse pointer and press the left mouse button. The cursor will jump to the new position and you can now continue editing. Ensure that the editor is in insert mode (large cursor) and press the letter "i". This now has to be confirmed by pressing the return key.

You have just used the insert key and the cursor keys. These keys have specific functions and there are a number of other keys specific to the editor as listed below:

- Clr: Clears the editor display (shift+Clr/Home)
- Delete: Deletes the character currently underneath the text cursor.
- Shift + Delete: Deletes all characters from the current cursor to the end of the line.
- Backspace: Deletes the character to the left of the text cursor.
- Return: Enters the current line (the line that the text cursor is on) in to the computers memory.
- Arrow Keys: Position the text cursor.
- Home: Moves the text cursor to the top left hand corner of the screen.
- Control + C: Terminates the current program and returns control back to the editor. This is useful when testing a program as you do not have to wait for the program to finish, you can simply stop it at any time.
- Help: Displays the program selection window as shown over the page.

Press HELP and the box will appear.



STOS allows four separate programs to be stored in memory simultaneously. The merits of this very useful facility may not be immediately apparent, but as you get involved with more and more complex programs, you will find it a valuable feature. Programs are normally developed in sections, and using the multiple program facility, the programmer can test individual segments of code before adding them to the main program. Throughout this course we shall be concentrating on one program at a time. The top half of the box therefore details the four programs that are currently in memory and these are selected using the cursor (arrow) keys.

STOS also allows ten accessory programs to reside in memory and these are listed in the accessory menu directly below the program menu. Accessories are themselves programs that can be called upon at any time to assist in the creation of our programs. There are some accessories supplied with STOS which we shall look at later, but do not worry too much about this yet. Finally, the amount of available memory is shown at the bottom of the box.

Pressing HELP a second time removes the box from the screen so go ahead and **press HELP**.

That just about covers the editor. This course is all about programming, and as the editor is required to write a program, you will soon become accustomed to its operation.

OUR FIRST PROGRAM

At last you cry, the first program!

Enter the following:

```
10 print "I am your friendly ST"  
20 print "How are you ?"
```

After each line you must press either the Return or Enter keys to enter the information in to the computers memory.

If you make any mistakes when typing in the program lines, go back and correct the mistakes as previously shown or simply re-type the whole line in again and it will replace the old, faulty line.

It may only be two lines but it is a valid computer program. Notice the numbers at the beginning of each line. These are known as *line numbers* and each line of a STOS basic program must start with a line number. These are used by the computer to establish the order in which the commands should be carried out or executed. Line 10 is carried out first, followed by line 20, line 30 and so on. The computer always starts execution at the lowest line number and works through in numerical order. As you type additional lines they are automatically placed in order.

Notice that line numbers 10 and 20 have been used as apposed to 1,2, etc. It is good practice to use multiples of ten rather than consecutive line numbers as this leaves space for any additional lines that you may wish to add later.

Following the line number is the program command. Line 10 contains the *print* command and this instructs the computer to print or display the text or information that follows. In this case a message is to be displayed on the screen. The text is placed within speech marks (these

can be found above the number 2 on the keyboard) but these will not be displayed when the message is printed. Line 20 prints a second line of text.

The program is now stored in memory but the computer does nothing further until told. To actually carry out, or execute the program, we use the *run* command. The *run* command instructs the computer to fetch the program from memory and carry out the commands that it contains.

Enter the following:

run [and press the RETURN or ENTER key]

Notice this time that no line number is required. When the line number is omitted, the command is carried out immediately and does not form part of the program. The computer carries out your command and prints the information on the screen as shown below:

I am your friendly ST
How are you ?

Notice that the speech marks are not printed but only the message within them.

CLEARING THE SCREEN

The display would look neater if it was cleared before displaying the message and another command can be added to the program to carry out this function. The screen must be cleared before the messages are printed, and thus if the new line is given the number 5, it will be placed before the other two lines.

Enter the following:

```
5 cls           [ remember to press the RETURN key ]
```

You do not have to type the whole program again but only line 5. Program lines are always placed in numerical order and so the new line 5 becomes the first line of the program. The *cls* command stands for clear screen and tells the computer to clear anything that is currently displayed on the screen. The program now consists of three lines which can be viewed using the *list* command. The *list* command lists the entire program with any new lines automatically shown in the correct position.

Enter the following:

```
list
5 cls
10 print "I am your friendly ST"
20 print "How are you ?"
```

The program is immediately recovered from memory and displayed on the screen.

Run the program and you will see that the screen is cleared before displaying the message.

REMOVING THE FUNCTION KEY WINDOW

When the program is run, the screen is cleared and the message printed. Although the screen has been cleared, the function key window is still displayed at the top of the screen. STOS automatically assumes that we want this displayed but it can be quite easily switched off by adding another line to the program.

Enter the following:

3 key off

and list the program:

```
list
3 key off
5 cls
10 print "I am your friendly ST"
20 print "How are you ?"
```

Notice how the new line has been placed in the correct position again. Run the program (using the *run* command) and you will see the two lines of text, but this time the screen is nice and clear.

```
run
I am your friendly ST
How are you ?
```

STOS allows multiple commands on the same line. In some cases this can be very useful but it can also make a large, complicated program very difficult to follow and it is best to limit each line to one command wherever possible. When the length of the line exceeds the width of the screen it is 'wrapped around' on to the next line and this can reduce the legibility of the program. Although the line may wrap around and visually take up two lines, the computer still considers it as one single line of code. For example, the last program could be written all on one line as shown below:

```
10 key off: cls: print "I am your friendly ST": print "How are  
you ?"
```

ADDING REMARKS TO PROGRAMS

Enter the following:

```
2 rem MY FIRST COMPUTER PROGRAM
15 ?
```

and list the program.

```
list
2 rem MY FIRST COMPUTER PROGRAM
3 key off
5 cls
10 print "I am your friendly ST"
15 print
20 print "How are you"
```

Line 2 contains the *rem* command. This stands for *remark* and allows notes and comments to be included within the program. Any text following the *rem* command is displayed when the program is listed (using the *list* command) but does not perform any function when the program is executed. Lots of programs take days, weeks, or even months to write and it is very easy to loose track of your code. It is surprising how much concentration is required for programming and it is quite easy to completely loose track of what you were doing after only a couple of hours away from the computer. If you get in to the practice of using *rem* commands with the simple examples throughout this course, you will have no trouble when you become a professional programmer.

Line 15 contains a question mark (?) which is a shorthand symbol for *print*. *Print* is one of the most common commands and it can save a lot of time typing a question mark instead of the complete word *print*. As can be seen from the last program listing, STOS automatically changes the question marks to the word *print*. Running the program

produces the following:

```
run
```

```
I am your friendly ST
```

```
How are you ?
```

There is now a blank line between the two lines of text. The *print* command, when placed on its own, causes a blank line to be displayed.

REPLACING AND REMOVING LINES

When typing in programs even the most professional programmers introduce errors. It is very rare indeed for a program to work first time and fault finding is a major element of software development. Typing errors are inevitable but are normally quite easy to locate and correct. A slightly harder problem to solve is when a program performs a different function to that intended by the programmer, and in this situation the actual design of the program may require alteration. STOS is quite helpful in these matters and displays an error message when it encounters a problem. The offending line of code is normally identified by its line number so that the programmer can quickly locate and rectify the problem. Let us place a deliberate fault in the program to see what happens.

Enter the following:

```
12 prin
```

The spelling mistake is intentional so type in the line exactly as shown and run the program.

```
run
I am your friendly ST
```

```
Syntax Error in line 12
12 PRIN
```

STOS reports a *Syntax error* in line 12. A syntax error means that there is something wrong with the syntax of the command. It could be, as in this example, that the actual command is spelt wrong. Alternatively the parameters following the command may be of the wrong format or value. STOS can generate a number of different error codes and these are listed at the back of the STOS Users Guide. The line could be corrected by moving the text cursor back and correcting the word, but we shall replace it instead.

Enter the following:

```
12 print
```

and list the program.

```
list
2 rem MY FIRST COMPUTER PROGRAM
3 key off
5 cls
10 print "I am your friendly ST"
12 print
15 print
20 print "How are you ?"
```

Notice that the new line 12 has replaced the old, faulty line 12. When a duplicate line number is specified, the new line replaces the old line.

Enter the following:

12

and list the program:

```
list  
2 rem MY FIRST COMPUTER PROGRAM  
3 key off  
5 cls  
10 print "I am your friendly ST"  
15 print  
20 print "How are you"
```

This time line 12 has completely disappeared. As per the previous example the new line 12 replaces the existing line 12, but as the line is blank, it is completely removed from the program. Therefore, to erase a line from a program just type the line number and press the return key.

Suppose now that we wanted to add another line after line 2. The problem is that we already have a line 3 and there is no room for the new line. We stated earlier that it is good practice to number lines in increments of 10, but there will still come a time when you meet the situation where all the available lines have been used. To overcome this problem STOS provides a command that allows the program lines to be renumbered.

RENUMBERING THE PROGRAM LINES

Enter the following:

renum

and list the program:

```
list
10 rem MY FIRST COMPUTER PROGRAM
20 key off
30 cls
40 print "I am your friendly ST"
50 print
60 print "How are you"
```

All of the program lines are renumbered starting at 10 and increasing in steps of 10. A new line could now be added between lines 10 and 20.

There are four ways of using the *renum* command. You have just seen the effect of *renum* on its own but the starting line number, the increment and a range of lines to be renumbered can also be specified.

Enter the following:

```
renum 100
```

and list the program:

```
list
100 rem MY FIRST COMPUTER PROGRAM
110 key off
120 cls
130 print "I am your friendly ST"
140 print
150 print "How are you ?"
```

A number placed at the end of the *renum* command specifies the starting line but the lines still increase in units of 10.

Enter the following:

```
renum 1000,100
```

and list the program:

```
list  
1000 rem MY FIRST COMPUTER PROGRAM  
1100 key off  
1200 cls  
1300 print "I am your friendly ST"  
1400 print  
1500 print "How are you ?"
```

This time both the starting line number and the line increment have been specified. The lines start at line 1000 and increment in steps of 100.

The last form of the *renum* command allows us to specify the starting line number, the step size and the range of lines to be renumbered.

Enter the following:

```
renum 1300,10,1300-1500
```

and list the program:

```
list  
1000 rem MY FIRST COMPUTER PROGRAM  
1100 key off  
1200 cls  
1300 print "I am your friendly ST"  
1310 print  
1320 print "How are you ?"
```

Lines 1300-1500 have been renumbered starting at new line number 1300 and increasing in steps of 10 whilst the rest of the line numbers remain unchanged.

To restore the program back to standard line numbering, enter the following:

```
renum  
list  
10 rem MY FIRST COMPUTER PROGRAM  
20 key off  
30 cls  
40 print "I am your friendly ST"  
50 print  
60 print "How are you ?"
```

THE SEMICOLON

Enter the program lines shown below taking particular care with line 80. The symbol at the end of the line is a semicolon and can be found next to the 'L' key on the keyboard. You may like to use a question mark instead of typing print.

```
70 print  
80 print "Welcome to ";  
90 print "STOS Basic"
```

Now list the program to check that the lines have been added.

```
list  
10 rem MY FIRST COMPUTER PROGRAM  
20 key off  
30 cls  
40 print "I am your friendly ST"
```



```
50 print
60 print "How are you ?"
70 print
80 print "Welcome to ";
90 print "STOS Basic"
```

and finally run the program:

```
run
I am your friendly ST

How are you ?

Welcome to STOS Basic
```

Notice how the text in lines 80 and 90 has been printed on the same line. The *print* command always terminates with a line feed and carriage return. This means, that at the end of the text, it moves the cursor down to the beginning of the next line and hence any further *print* commands start printing on this new line. Line 80 has a semicolon on the end and this stops the line feed and carriage return. The cursor thus remains at the end of the printed text and any further *print* commands will continue printing on the same line.

ERASING AND RESTORING PROGRAMS

The last program is no longer required and we must tell STOS that we wish to enter a new program. The *new* command is used for this and instructs STOS to remove the current program from memory so as to make way for a new one.

Enter the following:

```
new  
list
```

Nothing is displayed as the program has effectively been removed from memory. We use the word 'effectively' because, although not listed, the lines are still retained in memory.

Programming is an addictive pastime and it is quite likely that you will soon find yourself working away in to the small hours of the morning. With one eye closed, and the other on the way, there is a chance that you may erase the program by accident before saving it to disk.

The designers of STOS, who are obviously programmers themselves, identified this problem and thus gave STOS an indispensable command that allows programs to be restored. The command is *unnew*.

Enter the following:

```
unnew
```

and list the program:

```
list  
10 rem MY FIRST COMPUTER PROGRAM  
20 key off  
30 cls  
40 print "I am your friendly ST"  
50 print  
60 print "How are you ?"  
70 print  
80 print "Welcome to ";  
90 print "STOS Basic"
```


Hey presto, the program is back. Note, however, that the *unnew* command will only prove successful if used directly after the *new* command. If you start typing in a new program the original program will most definitely be lost for ever.

Enter the following:

```
new  
list
```

The program should have gone again and we can enter a new program.

PRINT ATTRIBUTES

STOS offers a number of attributes that can be applied to enhance the output of the *print* command. There are four main effects, normal, inverted, underlined and shadowed which can be freely mixed. If you have not already done so, remove any existing programs from memory using the *new* command.

We shall now load a program from disk 1 so place this disk in drive A. You will notice that each example program has the filename listed within a *rem* command at the start of the program. Looking at the program on the opposite page you will see that the filename is "a:\start\attrib". Therefore, to load this program enter the following:

```
load "a:\start\attrib"
```

After loading a program you should reset the STOS editor by pressing the UNDO key twice. This resets the system back to the default colours and clears any information that may still be displayed from a previous program. You can now view the program using the *list* command.

```
list
10 rem EFFECTS FOR USE WITH PRINT COMMAND
20 rem PROGRAM= A:\START\ATTRIB
30 :
40 print "This is NORMAL text"
50 :
60 print
70 under on
80 print "This is UNDERLINED text"
90 under off
100 :
110 print
120 shade on
130 print "This is SHADED text"
140 shade off
150 :
160 print
170 inverse on
180 print "This is INVERTED text";
190 inverse off
200 :
210 print
220 print
230 shade on
240 inverse on
250 print "This is INVERTED AND SHADED";
260 inverse off
270 shade off
```

As you work through the course you may either type the programs in direct or load them from disk.

Run the program.

This program is somewhat longer than the previous example but do

not worry as it is quite straight forward. The semicolons (:) are used to separate the program in to a number of blocks or modules and this makes the program a lot easier to read and understand. The colon is similar to the *rem* command in that it has no effect on the operation of the program and is provided only to improve the legibility of the program listing. Let us look at the program then.

Lines 10 and 20 contain remarks (the *rem* command) to indicate the subject of the program and the location of the program on the disks that accompany the course.

Line 40 displays a message in normal text.

Lines 70-90 display text which is underlined. Underlining is switched on and off using the *under on* and *under off* commands and is a nice effect for headings, etc.

Lines 120-140 provide a shadow effect by slightly reducing the brightness of the print. This effect is switched on and off using the *shade on* and *shade off* commands.

Lines 170-190 display a message in inverse video which is handy for highlighting a certain message or area on the screen. This effect is switched on and off using the *inverse on* and *inverse off* commands. Notice that a semicolon has been added to the *print* command in line 180. You have already seen that the *print* command normally terminates with a line feed and carriage return and this can cause a slight problem when using inverse video. Without the semicolon the text cursor moves down to the next line, and because we are using inverse video, the whole line changes colour. You can see the effect of this by removing the semicolon and running the program again.

Lines 230-270 illustrate how these effects can be combined.

SCREEN RESOLUTION

You may be aware that the Atari ST can operate in three screen resolutions, namely low, medium and high. On a normal television or colour monitor you can use low and medium resolutions, but high resolution is reserved for use on high resolution, monochrome monitors only.

The characteristics of each resolution are listed below.

LOW RESOLUTION

Text: 25 lines of 40 characters
Graphics: 320 x 200
Colours: 16

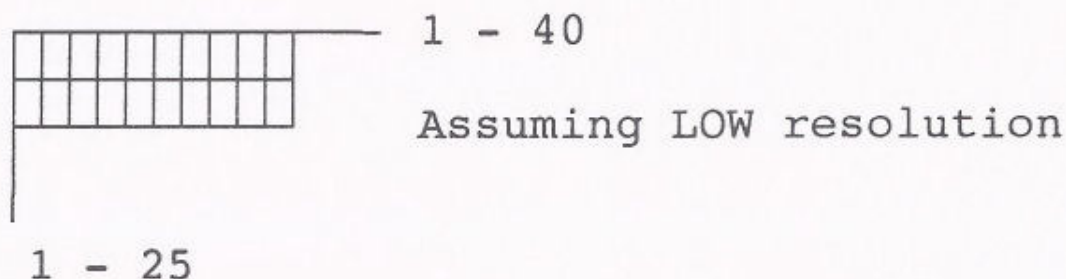
MEDIUM RESOLUTION

Text: 25 lines of 80 characters
Graphics: 640 x 200
Colours: 4

HIGH RESOLUTION

Text: 25 lines of 80 characters
Graphics: 640 x 400
Colours: Black and white only

The layout of the text screen can be directly related to a piece of graph paper with 40 boxes across by 25 boxes down.



All of the examples in this book are designed for low and medium resolution allowing full use to be made of the colour and graphic facilities offered by STOS. Graphics shall be covered in a later chapter but for now we are only concerned with text.

The higher the resolution the lower the number of colours that can be displayed. When choosing the screen resolution there are a number of factors that require consideration. The ultimate situation would be to have all the colours of low resolution combined with the higher resolution of medium mode, but this is not possible and thus a compromise between colours and resolution has to be met. A shoot-em-up game would probably benefit from the increased colour palette of low resolution whilst a word processor would not need too many colours and would benefit more from the larger text capacity of medium resolution.

The screen resolution is set at the start of a program using the *mode* command although it can also be changed at any time during a program. The format of the command is shown below:

mode N

where N is a value in the range 0-2 which indicates the resolution as listed below:

- 0 = Low resolution
- 1 = Medium resolution
- 2 = High resolution

Let us look at a short program to illustrate the difference between low and medium resolution.

Clear the previous program (using the *new* command) and load the following:

```
10 rem DIFFERENT SCREEN RESOLUTIONS
20 rem PROGRAM = A:\START\RES
30 :
40 key off
50 mode 0
60 print "Low Resolution"
70 :
80 wait key
90 :
100 mode 1
110 ?"Medium Resolution"
```

Line 40 switches off the function key window, line 50 sets the screen resolution low and line 60 displays a message. Line 80 contains the *wait key* command which halts the program and waits for the user to press any key on the keyboard. When a key is pressed the program continues. Line 100 changes the screen resolution to medium and line 110 prints another message.

Look carefully at the size of the printed messages. You may need to run the program a few times to see the effect of the resolution change. In low resolution the computer displays 40 characters across the screen, but in medium resolution the computer displays 80 characters across the screen and hence each character must be smaller.

CENTRED TEXT

In this section we shall see how to position text at a specified position on the screen. We know that low resolution mode offers 25 lines of 40 characters and medium resolution offers 25 lines of 80 characters and we may wish to display information say half way across the screen. The *print* command always starts printing at the left hand edge of the screen but suppose we want to display a heading centred across the screen. The following line of code would perform this task:


```
10 print "      WELCOME TO STOS BASIC"
```

Placing a series of spaces before the message would push the title across the screen but we would have to calculate the number of spaces required or rely on trial and error to position it exactly central. Luckily we do not need to muck around like this as STOS offers the *centre* command which is designed specifically for the job.

```
10 centre "WELCOME TO STOS BASIC"
```

The *centre* command displays the text at an exact central point across the screen.

Clear the last program. Enter and run the following:

```
10 key off : mode 0
20 under on
30 centre "WELCOME TO STOS BASIC"
40 under off
```

```
run
```

```
WELCOME TO STOS BASIC
```

This displays an underlined heading in the centre of the screen as shown.

Notice that the *key off* and *mode* commands have been placed together on the same line. The majority of programs do not require the function key window and thus we shall switch it off each time that we set the screen resolution. The *key off* command is placed before the *mode* command to ensure that the window is removed before the screen is updated by the *mode* command. If the commands are reversed you will find that the function key box briefly appears before being switched off, and this makes programs look rather unprofessional as well as worrying the user who may think that

something has gone wrong.

Now add a line 50 as shown below:

```
50 print "Hello"
```

and list the program to check that the line has been added:

```
list
10 key off : mode 0
20 under on
30 centre "WELCOME TO STOS BASIC"
40 under off
50 print "Hello"
```

Run the program and it will produce the following:

```
run
      WELCOME TO STOS BASICHello
```

The "Hello" message has been printed directly after the title. This is because, unlike the *print* command, the *centre* command does not produce a line feed and carriage return and the text cursor therefore remains on the same line at the end of the message. This problem is easily solved by placing a *print* command at line 45 which will perform a line feed / carriage return moving the cursor to the next line.

CHARACTER POSITION

The *centre* command is ideal for placing text at the centre of a line but we may want to place information at some other position along the line. There are two similar commands available for this - *space\$* and *tab*.

Clear the previous program and enter the following:

```
10 key off : mode 0
20 print space$(10);"10 Spaces Along"
```

Run the program.

```
run
      10 Spaces Along
```

The program prints ten spaces followed by the message. The number in brackets at the end of the *space\$* command indicates the number of spaces to be printed. Notice that the *space\$* command is separated from the main text by a semicolon. We have already seen that a semicolon inhibits the line feed/carriage return and hence the message is printed directly after the blank spaces.

The same result can be obtained with the program shown below:

```
10 key off : mode 0
20 print tab(10);"10 Spaces Along"
```

This time the *tab* command has been used instead of *space\$*. This produces exactly the same result although there is a fundamental difference between the two commands. *Space\$* actually prints ten spaces whilst *tab* moves the text cursor directly without printing the spaces.

PAGE POSITIONING

Space\$ and *tab* both allow text to be formatted on a single line but suppose we want to print something say seven lines down from the top of the screen. For this STOS offers a much more powerful command which gives total control over the whole screen. Consider, for

example, that we wish to print "HELLO" seven spaces in from the left hand side and six lines down from the top of the screen. One way of doing this is as follows:

```
10 key off : mode 0
20 print
30 print
40 print
50 print
60 print
70 print tab(7);"HELLO"
```

Five *print* commands are used to move the text cursor down to the sixth line and a *tab* command is used to position the text seven spaces from the left hand edge of the screen.

The same effect can be achieved using the *locate* command which allows the text cursor to be moved directly to a specific position on the screen.

Clear the previous program and enter the following:

```
10 key off: mode 0
20 locate 7,6
30 print "HELLO"
```

Run the program and the message will be displayed.

The format of the *locate* command is shown below:

```
locate X,Y
```

where X indicates the character position across the line and can range from 0 to 79 in medium resolution and 0 to 39 in low resolution. Y indicates the line number and can range from 0 to 24 in all

resolutions.

ASCII CODES

We have seen various examples throughout this chapter that print information to the screen. For example, to print the letter "A" we would use the following command:

```
10 print "A"
```

The computer does not understand human languages and represents information as numbers or codes. All of the printable character are represented by a code known as the ASCII code. ASCII stands for American Standard Code for Information Interchange and the ASCII code for the letter "A" is 65. The letter could be printed by specifying its ASCII code as shown below.

Enter the following:

```
print chr$(65)  
A
```

The *chr\$* command allows a character to be specified by its ASCII code. A second command *asc* is used to identify the ASCII code of a character.

Enter the following:

```
print asc("A")  
65
```

We do not normally need to worry about ASCII codes as characters and numbers can be displayed directly using the *print* command, but ASCII codes also represent characters that are not accessible from the

keyboard. STOS offers various other characters such as foreign letters and graphical symbols that can only be displayed using the ASCII codes.

For example, enter the following:

```
mode 0  
print chr$(132)
```

This displays an accented letter "a".

```
print chr$(171)
```

This prints a 'half' symbol.

```
print chr$(189)
```

This prints a copyright symbol.

In chapter 6 we shall write a small program that will display all of the characters on the screen.

Well that just about brings us to the end of chapter 1. All of the examples in this first chapter have displayed information in black and white and it is now time that we thought about adding some colour.

Chapter 2

A Splash of Colour

In the last chapter we saw how to display information, how to apply different effects and how to position information exactly where we want it.

The Atari ST is renowned for its colourful graphics and STOS offers more features than any other language for utilising these to the full. Graphics are covered in later chapters but colour can be used to great effect with the *print* command to produce colourful and eye catching displays.

There are two commands for controlling colour with text. These are *pen* for specifying the colour of the text and *paper* for specifying the colour of the background - quite appropriate names! The *pen* and *paper* commands are followed by a number which indicates the number of the colour. This number is known as the *colour index* and can range from 0 to 15. In low resolution mode all sixteen colours can be displayed simultaneously, but in medium resolution only four colours can be displayed and thus colour index numbers are limited to the range 0-3.

Enter or load the following:

```
10 rem LOW RESOLUTION COLOURS
20 rem PROGRAM = A:\COLOUR\COLOUR
30 :
40 rem SET SCREEN RESOLUTION LOW
50 key off : mode 0
60 :
70 rem SET PEN AND PAPER COLOURS
80 paper 4
90 pen 14
100 :
110 rem DISPLAY TEXT
120 print "Purple text on a yellow background"
130 print
140 :
150 inverse on
160 print "Yellow text on a purple background"
170 inverse off
180 print
190 :
200 paper 0
210 pen 1
220 print "Boring old black and white again"
```

Run the program.

Lines 10 and 20 contain remarks indicating the nature of the program and the location of the program on the disks that accompany the course.

Line 50 switches off the function key window and sets the screen resolution to low.

Line 80 sets the paper colour to colour index number 4 which is set

by default to yellow. Line 90 sets the pen colour to colour index number 14 which is purple.

Line 120 prints a message which is displayed using the new pen and paper colours. The message is thus displayed as purple text on a yellow background.

Lines 150-170 display another message, this time in inverse video. The *inverse on* command inverts or swaps the pen and paper colours and hence the message is displayed as yellow text on a purple background.

Line 200 changes the paper colour to colour index 0 which is black and line 210 changes the pen colour to colour index 1 which is white. Line 220 then displays a message in boring old black and white again.

DEFINING INDIVIDUAL COLOURS

As mentioned previously, there are sixteen (0-15) colour index numbers which are used with the *pen* and *paper* commands. STOS initially assigns a range of colours to these but they can be changed within a program. The Atari ST offers a colour palette of 512 colours and we can assign any of these to any of the colour index numbers. The Atari STE computer has an extended colour palette of 4096 colours, but at the time of writing this course, STOS is unable to utilise these extra colours.

The default colours for low resolution are shown in Table 1 over the page but these can be changed using the *colour* command. The format of the *colour* command is shown below:

colour INDEX, \$RGB

INDEX is a value in the range 0-15 and indicates the colour index.

Following the index number is the new colour information which is expressed as three numbers in the range 0-7 and signifies the relevant strengths of the three primary colours Red(R), Green(G) and Blue(B). By mixing these red, green and blue elements a very large range of colours and shades can be produced. Table 2 shows some examples.

TABLE 1: The Default Colour Settings (low resolution)

Index	Colour	Index	Colour
0	Black	1	White
2	Flashing	3	Black
4	Yellow	5	Brown
6	Light Brown	7	Green
8	Light Grey	9	Dark Grey
10	Red	11	Light Green
12	Light Yellow	13	Mauve
14	Pink	15	Light Blue

Table 2: Example Colour Mixtures

Colour	Colour Elements	\$RGB
Black	R=0 G=0 B=0	\$000
White	R=7 G=7 B=7	\$777
Red	R=7 G=0 B=0	\$700
Green	R=0 G=7 B=0	\$070
Blue	R=0 G=0 B=7	\$007
Yellow	R=7 G=7 B=0	\$770

Therefore, to set colour index 5 to the colour yellow we would use the following command:

10 colour 5,\$770

This sets colour element RED and colour element GREEN to the maximum strength seven.

It is important to note that colour changes take immediate effect and that any information currently displayed on the screen will change accordingly.

DEFINING THE COMPLETE COLOUR PALETTE

Programmers normally define the colours at the start of a program although they can also be changed at any time during the program. To define all sixteen colours would require sixteen separate *colour* commands so STOS offers another command, *palette*, which allows all of the colours to be defined using the one command. Note, however, that you do not have to define all of the colours. If you only want to use say five colours then that is all you need to specify. If the colours are specified at the start of the program using a *palette* command they can easily be changed at a later stage in the program using another *palette* command or the *colour* command.

The *palette* command is followed by a list of the colour definitions for each colour index as shown below:

```
10 rem SET COLOURS IN LOW RESOLUTION
20 mode 0
30 palette $000, $777, $070, $700
```

In this example the screen resolution is set to low and four of the possible sixteen colours are defined as shown below:

0 = black (\$000)	1 = white (\$777)
2 = green (\$070)	3 = red (\$700)

Clear the previous program and load the following:

```
10 rem ILLUSTRATE PALETTE + COLOUR
20 rem PROGRAM = A:\COLOUR\PALETTE
30 :
40 rem SET SCREEN RESOLUTION LOW
50 key off : mode 0
60 :
70 rem DEFINE COLOUR PALETTE
80 palette $0,$777,$700,$700,$70,$7
90 :
100 rem PRINT MESSAGES
110 pen 1
120 print "I like WHITE"
130 pen 3
140 print "I like RED"
150 pen 4
160 print "I like GREEN"
170 pen 5
180 print "I like BLUE"
190 :
200 rem WAIT FOR USER INPUT
210 wait key
220 :
230 rem CHANGE THE COLOURS
240 colour 3,$777
250 colour 4,$777
260 colour 5,$777
```

Run the program and you will see four coloured messages on the screen. Press any key and the messages will all change colour to white.

The program demonstrates the *palette* command and also highlights the fact that existing information changes colour when the colour

index number is re-defined. The program displays four lines of text each in a different colour, and when the user presses a key, the displayed text changes to white.

Line 50 switches off the function key window and sets low screen resolution.

Line 80 defines the colour palette. Index numbers 0-5 have been defined as shown below:

0 = black (\$000)	1 = white (\$777)	2 = red (\$700)
3 = red (\$700)	4 = green (\$070)	5 = blue (\$007)

Lines 110-180 display four lines of text each in a different colour.

Line 210 waits for the user to press a key.

Lines 240-260 change colour index numbers 3-5 to white (\$777). As previously mentioned, colour changes take immediate effect and hence the currently displayed information changes colour to white.

FLASHING COLOURS

We have just seen how to assign individual colours to the colour index numbers but STOS takes this one step further by allowing multiple colours to be assigned to a single colour index.

Enter or load the following:

```
10 rem FLASHING COLOURS
20 rem PROGRAM = A:\COLOUR\FLASH1
30 :
40 rem SET SCREEN RES LOW
50 key off: mode 0
```



```
60 :  
70 rem DEFINE THE FLASHING SEQUENCE  
80 flash 2,"(000,25)(777,25)"  
90 :  
100 rem DISPLAY MESSAGE  
110 pen 2  
120 print "I like flashing!"
```

Run the program and you will see that it displays a flashing black and white message. You can stop the flashing by pressing the UNDO key twice. The main area of interest is the *flash* command at line 80 so let's take a detailed look at this.

The format of the *flash* command is shown below:

flash INDEX, "(COL1,TIME1) (COL2,TIME2)...."

INDEX indicates the colour index number to be set flashing.

COL indicates the colour defined as RGB strengths.

TIME indicates the amount of time that the colour should be displayed. This is specified in 50ths of a second and hence a value of fifty (50/50ths) will equal one second; a value of twenty five (25/50ths) will equal half a second, etc.

The *flash* command takes each colour (COL) from the list, assigns it to the colour index (INDEX) for the length of time (TIME). The complete sequence is continuously repeated, so when the end of the list is reached, it starts again.

So how does this apply to line 80 of our program ?

```
80 flash 2,"(000,25)(777,25)"
```

Colour index 2 is set flashing as follows:

(000,25): colour \$000 (black) is displayed for 25/50ths (half) of a second

(777,25): colour \$777 (white) is displayed for 25/50ths (half) of a second.

The colour therefore changes between black and white every half a second.

Let us now try another example.

Load the following:

```
10 rem FLASHING COLOURS
20 rem PROGRAM = A:\COLOUR\FLASH2
30 :
40 rem SET SCREEN RES LOW
50 key off: mode 0
60 :
70 rem DEFINE THE FLASHING SEQUENCE
80 flash 2, "(777,50) (700,10) (777,50) (070,10)
(777,50) (007,10)"
90 :
100 rem DISPLAY MESSAGE
110 pen 2
120 print "Press any key"
```

Run the program and you will see colour index 2 flash through a sequence of colours as defined below:

white (777) for 50/50ths of a second
red (700) for 10/50ths of a second
white (777) for 50/50ths of a second

green (070) for 10/50ths of a second
white (777) for 50/50ths of a second
blue (007) for 10/50ths of a second

As you can see, some quite complicated sequences can be defined and the best part of all is that the whole operation is carried out under interrupt. This means, that once the sequence is defined, our program can carry on doing other things whilst STOS maintains the flashing in the background.

Just before we leave, load the following:

```
10 rem FLASHING BACKGROUND
20 rem PROGRAM = A:\COLOUR\FLASH3
30 :
40 rem SET SCREEN RESOLUTION LOW
50 key off: mode 0
60 :
70 centre "A FLASHING BACKGROUND"
80 :
90 rem DEFINE FLASHING COLOURS
100 flash 0, "(111,5) (222,5) (333,5) (444,5) (555,5)
(666,5) (777,5)"
```

This time colour index 0 (zero) has been defined as a flashing sequence of colours. The screen background always defaults to colour index 0, and if this is redefined, so the background colour will change. Therefore the program makes the background flash. This flashing can be terminated with the *flash off* command.

Enter the following:

flash off

and normality should be restored. To restore the editor back to

standard black and white, press the UNDO key twice.

So to recap - the *colour* command assigns a single colour to a colour index number, the *flash* command assigns a sequence of colours to a colour index number and the *palette* command defines multiple colour index numbers within the one command.

SHIFTING THE COLOUR PALETTE

The last colour effect we shall look at is *shift*. The *shift* command rotates the colours in the colour palette through each of the sixteen colour indexes and can produce some very nice effects. The description of the *shift* command in the STOS Users Guide is rather misleading as the command only rotates the sixteen colour index numbers and does not rotate the entire five hundred and twelve colour palette as implied. Let us look at an example.

Enter or load the following:

```
10 rem THE SHIFT COMMAND
20 rem PROGRAM = A:\COLOUR\SHIFT
30 :
40 rem SET SCREEN RES LOW
50 key off : mode 0
60 :
70 rem DEFINE COLOUR PALETTE
80 palette .....,$700,$70,$7,$770,$777
90 :
100 rem DISPLAY MESSAGE
110 pen 11
120 print "H  H EEEEE L    L    OOOOOO"
130 pen 12
140 print "H  H E    L    L    O  O"
150 pen 13
```



```
160 print "HHHHHHH EEE L L O O"
170 pen 14
180 print "H H E L L O O"
190 pen 15
200 print "H H EEEEE LLLLL LLLLL OOOOOO"
210 :
220 rem SHIFT COLOURS STARTING AT INDEX 11
230 shift 15,11
```

Run the program and it will display the word hello with shifting colours. Press the UNDO key twice to stop the effect.

Line 50 switches the function key window off and sets the screen mode to low resolution.

Line 80 defines the colour palette as shown below:

Colour index 11= red (\$700) Colour index 12= green (\$070)
Colour index 13= blue (\$007) Colour index 14= yellow (\$77)
Colour index 15= white (\$777)

Lines 110-200 display the text using a different pen colour for each line.

Line 230 switches on colour shifting using the *shift* command. The format of the *shift* command is shown below:

shift TIME, START INDEX

TIME indicates the delay between each change of colour and is specified in 50ths of a second.

INDEX specifies the starting colour, and if omitted, the whole 16 colours are shifted. Let us see how this applies to our program. Line 230 is shown on the next page.

230 shift 15,11

This means that each colour will be shifted every 15/50ths of a second and that only colours with an index number of 11 or above will be effected.

The effect of the shifting process and the contents of each colour index are illustrated in the table below:

	11	12	13	14	15
START	red	green	blue	yellow	white
SHIFT 1	white	red	green	blue	yellow
SHIFT 2	yellow	white	red	green	blue
SHIFT 3	blue	yellow	white	red	green
SHIFT 4	green	blue	yellow	white	red

Well that brings us to the end of the colour commands.

Chapter 3

Numeric Variables

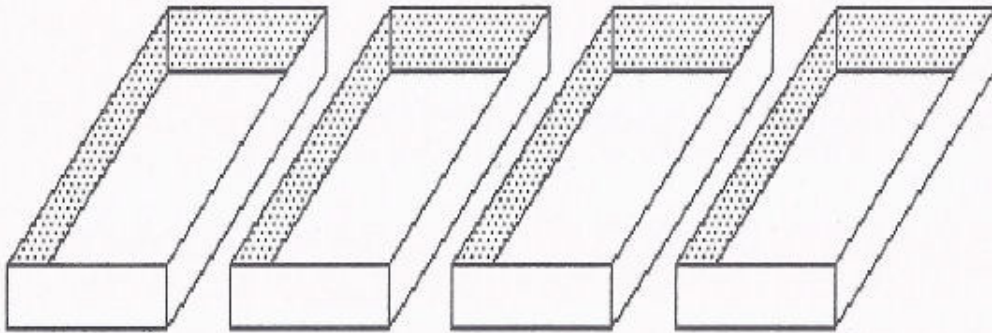
You now know how to display information on the screen complete with colour and effects, and before moving on to more interesting topics, there is one important area that must be covered. One of the largest chapters in many books is devoted to the subject of variables and arrays, and it is a subject which is often over complicated. It is true that variables are a very important aspect of programming and it is essential that you understand the concepts behind them before moving on to further sections, but the best way to learn is to experiment with example programs. The theory in this section has therefore been kept to a minimum. In fact this section is one of the smallest in the course so that you may move quickly on to the next chapter and see the theory put in to practice.

VARIABLES

Computer programs are stored in the computers memory and can be recalled as and when required. Whenever we issue a *list* or *run* command the computer recalls the program information from memory and either displays the information on the screen or carries out the

relevant commands. This chapter looks at the ways in which we can use the computers memory to store and recall our own information.

Consider the computers memory as a series of boxes each of which can hold one piece of information.



This information may consist of numbers or text. So how does STOS use these boxes ?

Suppose that we want to maintain the age of a boy called Peter and that Peter is 10 years old. We need to assign the value 10 to a variable using the *let* command.

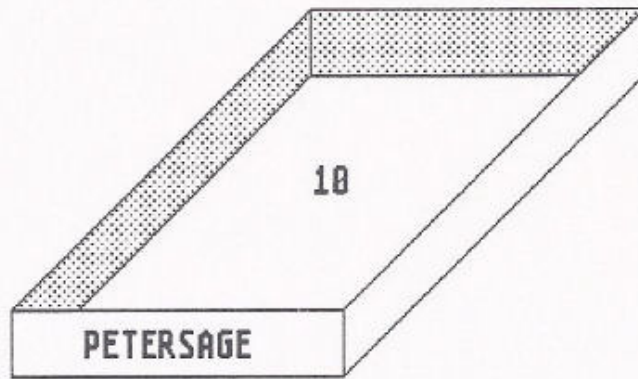
Enter the following:

let petersage = 10

The computer responds to your command by performing the following operations:

- (1) it selects one of the unused memory boxes.
- (2) it attaches the name PETERSAGE to the box.
- (3) it stores the number 10 inside the box.

The effect of this is shown in the diagram on the next page.



These electronic memory boxes cannot be seen with your eyes and the only way to check that the information has been stored is to ask the computer to recall it.

You already know that the *print* command prints information on the screen, so enter the following:

```
print petersage  
10
```

The computer displays the contents of the box named PETERSAGE. The name PETERSAGE is known as a *variable*. It is said to be 'variable' because the contents of the box can be altered or can 'vary'. For example, assume that Peter has a birthday and is now 11 years old.

Enter the following:

```
petersage = 11  
print petersage  
11
```

The box named PETERSAGE already exists, currently holding the value 10. When a new piece of information is assigned to the same variable name it overwrites or replaces the existing information. In this case the number 10 is replaced with the number 11. Notice also that the *let* command has been omitted this time. The *let* command is optional in STOS basic and hence we shall no longer bother with it.

Enter the following:

```
new
print petersage
0
```

You will notice now that the value 0 (zero) is displayed. We know that the *new* command erases the current program from memory but it also tells the computer to forget all the names that have been assigned to the memory boxes and the contents of those boxes. We can achieve the same effect from within a program using the *clear* command. This clears all of the variables whilst maintaining the program in memory. The following program illustrates this.

Enter the following:

```
10 age = 20
20 print age
30 clear
40 print age
```

run the program and it will produce the following:

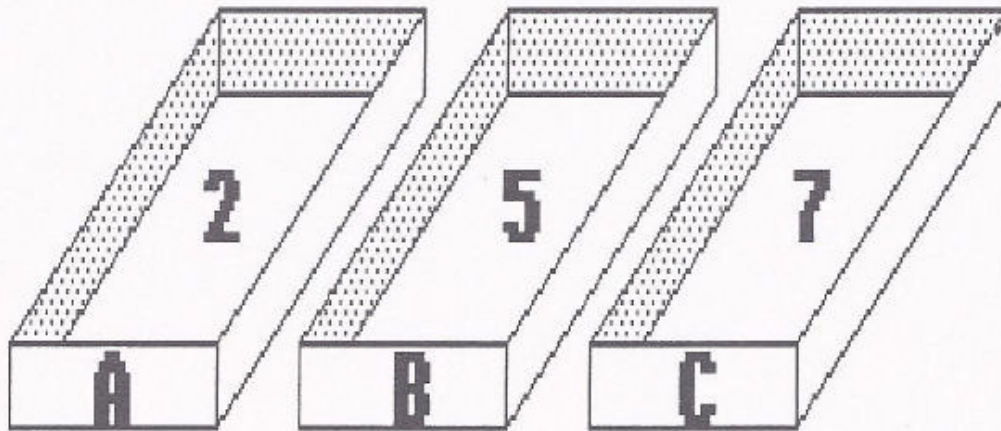
```
run
20
0
```

This shows that the *clear* command has cleared the variable.

We shall now write a simple program using variables. Clear the previous program and enter the following:

```
10 a = 2
20 b = 5
30 c = a + b
```

```
40 print c
```



Run the program.

```
run  
7
```

The program does nothing more than print the value 7. Line 10 assigns the value 2 to variable A, line 20 assigns the value 5 to variable B and line 30 adds variables A and B together and assigns the result to variable C. Variable C is then printed at line 40. The program uses 3 variables as shown in the diagram above. The exact operations that the computer carries out are listed below.

- (1) Assign the value 2 to a memory box and label it A.
- (2) Assign the value 5 to a memory box and label it B.
- (3) Fetch the contents of memory boxes A and B and add them together.
- (4) Assign this new value to a new memory box and label it C
- (5) Fetch the contents of the memory box labelled C and print it.

TYPES OF VARIABLE

The variables A, B and C, as used in the previous example are known

as *numeric variables* as they represent numerical information. Numerical variables can be divided in to two categories - *integer* and *real number*. Integers are whole numbers with no decimal element (e.g. 1, 7, 24, 1028) whilst real numbers are full decimal numbers (e.g. 1.24, 3.68). STOS assumes all numeric variables to be integer unless told otherwise. It does this because integers can be processed a lot faster than real numbers. To make the variable a real number we simply add a "#" symbol to the end of the variable name. For example:

```
length# = 124.356
```

```
ans# = 24.21
```

If you try and assign a real number to an integer variable you will lose the decimal element. For example, enter the following:

```
s = 12.45  
print s  
12
```

The variable S is assigned the number 12 or the integer part of the number only. It is, however, ok to store integers in real number variables.

When designing a program, decide in advance whether you will require integer or real number variables. If you only need integers then stick to numeric integer variables as this speeds up program execution and requires slightly less memory. If you require real numbers, or are unsure whether part of a program may result in the generation of a real number, then use real number variables. You can freely mix both types of variable within a program so it may be that you only need one real number variable whilst the rest are ok as integers.

Variables can also represent text information. This type of variable is known as a *string variable* and is represented by a '\$' (shift-4) at the end of the variable name. For example:

```
10 name$ = "Mr J Smith"
```

We shall look at string variables in another chapter so do not worry about these yet.

Just to recap, the types of variables offered by STOS are as follows:

AGE	- numeric integer variable.
T#	- real number with digits after the decimal point.
NAME\$	- string variable for storing text information.

VARIABLE NAMES

It makes sense to assign variable names appropriate to the information that they are going to represent. For example, in the previous examples the variable name PETERSAGE was used to represent the age of a boy named Peter. The programmer is fairly free to allocate variable names although they must conform to the following:

- (1) The variable name must start with a letter although numbers may be included. For example D9 is acceptable but 9D is not.
- (2) The variable must only consist of one word. If you wish to use more than one word then link them with an underscore symbol. For example telephone_no, last_name\$, etc. The underscore symbol can be found next to the 0 (zero) key along the top row of keys. The underscore symbol is the one at the top of the key so remember to press the shift key as well.
- (3) The length of the variable name must not exceed 31 characters.

(4) The variable name must not include any of the words AND, AS, ELSE, GOSUB, GOTO, MOD, STEP, THEN, TO, DATE\$, TIME\$, PI, OR or XOR. This is due to the fact that STOS may have trouble differentiating between the variable names and the program commands. This list is not as restrictive as it may first appear and should allow you to choose variable names that are associated with the actual data that they are going to represent. For example, you may use the variable name SURNAME\$ to represent a persons surname or the variable name AGE to represent a persons age.

The following variable names are valid...

a\$ x\$ surname\$ description\$ zx98 stck_number

The following variable names are illegal...

last name\$	- this consists of two words.
sort	- this contains the word "OR" which is not allowed.
9d	- variable names must start with a letter.
stock_number	- this contains the word "TO" which is not allowed.

VARIABLE ARRAYS

STOS, like many versions of basic, allows a number of memory boxes or variables to be grouped together under one name. This group is known as an *array* and any type of variable can form an array. In addition to providing more variables, arrays offer a lot more flexibility as shall become clear as you work through later sections of this course. Consider the following program:

```
10 dim A(3)
20 A(1) = 2
30 A(2) = 5
40 A(3) = A(1) + A(2)
```

```
50 print A(3)
```

Line 10 introduces the *dim* command. *Dim* stands for *dimension* and it dimensions or sets the size of the array, or the number of elements. The number in brackets, in this case 3, indicates the number of elements that the array will have. This creates 4 memory boxes named A(0), A(1), A(2) and A(3). All of the boxes have the value 0 (zero) in the case of numeric variables, or a null string (") in the case of string variables, initially assigned to them. Notice that, although the array is only dimensioned to 3, we actually get 4 variables. This gives a memory box arrangement as shown below:

A(0)	0
A(1)	2
A(2)	5
A(3)	7

Note that if we now try and use variable A(4), a 'Subscript out of range' error message is generated as the array only has four elements (0-3).

Arrays, as previously stated, can also consist of string variables as shown below:

```
10 dim Z$(4)
20 Z$(0) = "What "
30 Z$(2) = "is "
40 Z$(3) = "your name"
50 Z$(4) = Z$(0) + Z$(2) + Z$(3)
```

Line 10 dimensions an array named Z\$ to have 5 elements. Each array element is then defined. The information would be stored as

follows:

Z\$(0)	What
Z\$(1)	
Z\$(2)	is
Z\$(3)	your name
Z\$(4)	What is your name

Notice that we do not have to use all of the array elements. In the above example we have left out variable Z\$(1).

Arrays can also be two and three dimensional but this can become complicated and is rarely required. Just to illustrate the concept consider the following:

10 dim X(5,5)

X(0,0)	X(0,1)	X(0,2)	X(0,3)	X(0,4)	X(0,5)
X(1,0)	X(1,1)	X(1,2)	X(1,3)	X(1,4)	X(1,5)
X(2,0)	X(2,1)	X(2,2)	X(2,3)	X(2,4)	X(2,5)
X(3,0)	X(3,1)	X(3,2)	X(3,3)	X(3,4)	X(3,5)
X(4,0)	X(4,1)	X(4,2)	X(4,3)	X(4,4)	X(4,5)
X(5,0)	X(5,1)	X(5,2)	X(5,3)	X(5,4)	X(5,5)

You can imagine how a three dimensional array would look!

Chapter 4

Numeric Functions

In this chapter we shall continue on the theme of variables and shall look at the various facilities offered by STOS for controlling and using numeric variables. Your Atari ST computer really is very clever. In addition to basic mathematics it offers a full range of trigonometry functions allowing angles, lengths of sides of triangles, areas of circles, etc. to be calculated.

The Beginners Guide to STOS Basic has been designed to cover as wide a range of interests as possible, and whilst many of these mathematic facilities will be of no interest to the general programmer, there are probably a lot of readers who are still at school or involved in higher education who will benefit immensely from the information contained within the following few pages.

The next few sections therefore introduce a number of small programs that will prove invaluable to anyone studying mathematics, especially if working towards the GCSE. Topics include the calculation of areas and volumes, Pythagoras' theorem and the various trigonometry rules, and by the end of the chapter you should be able to write a program to cover any mathematical formula.

BASIC MATHEMATICS

Let us start by looking at the main mathematical facilities which can be applied to both integer and real number variables. The four rules of mathematics are shown below along with the symbols that the computer uses to represent them.

<u>RULE</u>	<u>SYMBOL</u>
Addition	+
Subtraction	-
Multiplication	* [found above the 8 key]
Division	/ [found next to the right shift key]

Notice that addition and subtraction are indicated by the standard symbols but multiplication and division use two new symbols.

The following segments of code illustrate the type of operations that can be performed.

```
10 N1 = 4
20 N2 = 2
30 ADD = N1 + N2
40 print N1;" + ";N2;" = ";ADD
```

This program produces the following:

$4 + 2 = 6$

Lines 10 and 20 assign the values 2 and 4 to the variables N1 and N2.

Line 30 adds the value of variable N1 to the value of variable N2 and assigns the result to variable ADD.

All of the mathematical rules can be used in this way and they may also be included as part of the *print* command as the following

example illustrates:

```
10 X = 10
20 Y = 5
30 print X;" x ";Y;" = ";X*Y
```

This produces:

10 x 5 = 50

Real number variables can be used as shown below:

```
10 A# = 1
20 B# = 5
30 ANS# = B#-A#
40 print ANS#
```

Real and integer values can be mixed:

```
10 A# = 1.25
20 B = 5
30 C = B / A#
40 print "The answer is: ";C
```

This produces:

The answer is 4

Now look very carefully at the next example:

```
10 N1 = 5
20 N1 = 2
30 N2 = 3
40 T = N1 + N2
50 print "The answer is ";T
```


This program produces the following:

The answer is 5

Notice that the variable N1 has been assigned two values. Remember that a variable will always represent the last allocated value, and hence in this case, N1 will represent the value 2. Line 10 therefore serves no other purpose than to simply re-illustrate this fact.

We shall now write some practical programs using numerical variables.

CALCULATING AREAS

In this section we shall produce a number of programs for calculating areas of shapes.

Area of a rectangle

Consider a program that calculates the area of a rectangle. The area of a rectangle is equal to the length multiplied by the width as shown below:

$$\begin{array}{ccc} \boxed{\text{AREA} = \text{LENGTH} * \text{WIDTH}} & \text{WIDTH} & \\ & \text{LENGTH} & \end{array}$$

A program to carry out this operation is shown on the next page.

```
10 LENGTH = 10
20 WIDTH = 5
30 AREA = LENGTH * WIDTH
40 print "The Area of the Rectangle is ";AREA
```

Lines 10 and 20 assign values to the variables LENGTH and WIDTH, line 30 performs the calculation and assigns the result to variable AREA and line 40 displays the result.

If you now wish to find the area of a rectangle whose length is 25, you would have to change the value assigned to the appropriate variable. In the same manner, if the width changes, the variable WIDTH would also have to be updated. We know that if we use a duplicate line number the line will be replaced with the new information. To calculate the area of a rectangle with length 25 we could just replace line 10 by typing:

```
10 LENGTH = 25
```

The ability to change old lines for new is a powerful facility for the programmer, but most programs are written for use by non programmers who would not know how to make such amendments. We therefore require an alternative method that allows the user to assign any value to the variable.

Load the following:

```
10 rem CALCULATE THE AREA OF A RECTANGLE
20 rem PROGRAM: A:\NUMERIC\RECTANG
30 :
40 rem INPUT DATA
50 print "Enter the length"
60 input L#
70 print "Enter the width"
80 input W#
```



```
90 :  
100 rem PERFORM CALCULATION  
110 AREA# = L# * W#  
120 print "The area of the rectangle is ";AREA#
```

Run the program and you will be asked to enter the length. Type in a number and press the Return key. You will then be asked to enter the width so enter another value and press the Return key. The computer will then display the area of the rectangle.

The program introduces the *input* command. When the computer reaches the *input* command it displays a question mark and waits for the user to enter some information. The user enters the information and presses the Return key to indicate that they have finished. When entering information the user may use the editing facilities (backspace, delete, etc) to amend any mistakes before pressing the Return key. Once the Return key is pressed, the computer assigns the entered information to the variable. The program can now be used over and over again with different values for the LENGTH# and WIDTH#. Notice that real number variables have been used so that the user may use full decimal values.

When the program is run it produces the following:

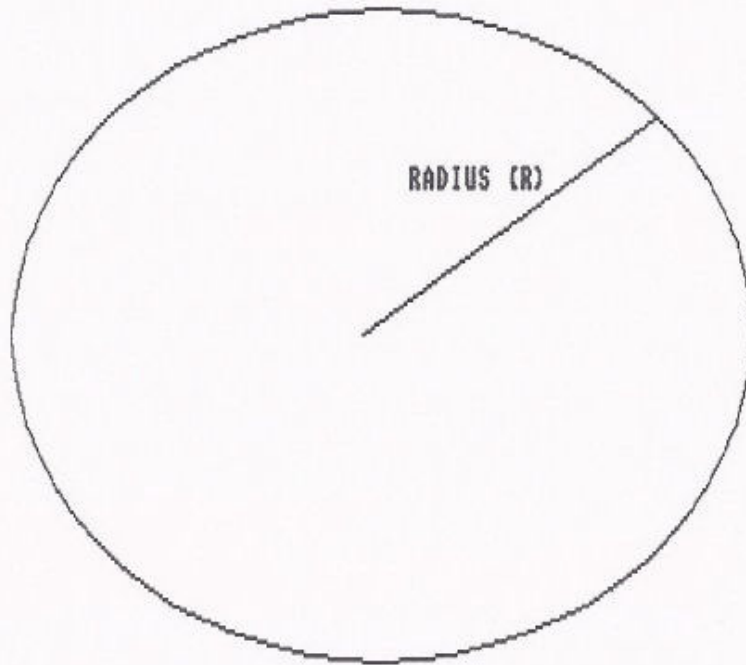
```
run  
Enter the length  
? 25  
Enter the width  
? 12  
The area of the rectangle is 300
```

Area of a circle

The area of a circle is calculated by the following formula:

$$\text{AREA} = \pi R^2$$

The area is equal to PI times the radius squared. So let us write a program to perform this calculation.



Load the following:

```
10 rem CALCULATE THE AREA OF A CIRCLE
20 rem PROGRAM: A:\NUMERIC\CIRCLE
30 :
40 rem INPUT DATA
50 print "Enter the radius of the circle"
60 input RADIUS#
70 :
80 rem PERFORM CALCULATION
90 AREA# = PI * (RADIUS# * RADIUS#)
100 print "The area of the circle is ";AREA#
```

Run the program.

run

Enter the radius of the circle

? 2

The area of the circle is 12.5664

The program stops at line 60 and waits for the user to enter a value. The user enters a value for the radius and presses the Return key. This value is assigned to variable RADIUS# and line 90 uses this to carry out the calculation. The calculation is performed as follows:

- (1) The radius is squared - (RADIUS# * RADIUS#)
- (2) The result from no (1) is multiplied with the value of PI(π). PI is a mathematical constant representing the value 3.14159 and STOS permanently assigns this value to the variable name PI. PI is thus known as a *reserved variable* as it is reserved for use by STOS and cannot be altered by the programmer.

Notice that the calculation to square the RADIUS# was placed in brackets. Brackets, as per standard mathematical rules, separate formulas to indicate the order in which the calculation should be made. The example below clarifies this.

$$\begin{array}{rcll} (1) & (3 * 4) + & 5 & = 17 \\ & 12 & + & 5 & = 17 \end{array}$$

$$\begin{array}{rcll} (2) & 3 * & (4 + 5) & = 27 \\ & 3 * & 9 & = 27 \end{array}$$

In example (1) the brackets are placed around the first part of the calculation and hence (3*4) is added to 5. In the second example the brackets are moved to the second part of the calculation and we get 3 multiplied by (4+5).

Whilst the calculation for the area of a circle illustrated the use of brackets, it could be simplified and the brackets removed.

Original formula: $AREA\# = PI * (RADIUS\# * RADIUS\#)$
New formula: $AREA\# = PI * RADIUS\#^2$

The ^ symbol is produced by pressing shift 6 and allows any value or variable to be raised to a specific power. This allows values to be squared, cubed, etc. The following examples illustrate this.

Enter the following:

```
print 3^2  
9
```

Enter the following:

```
print 4^3  
64
```

Enter the following:

```
print 2^2 + 5^2  
29
```

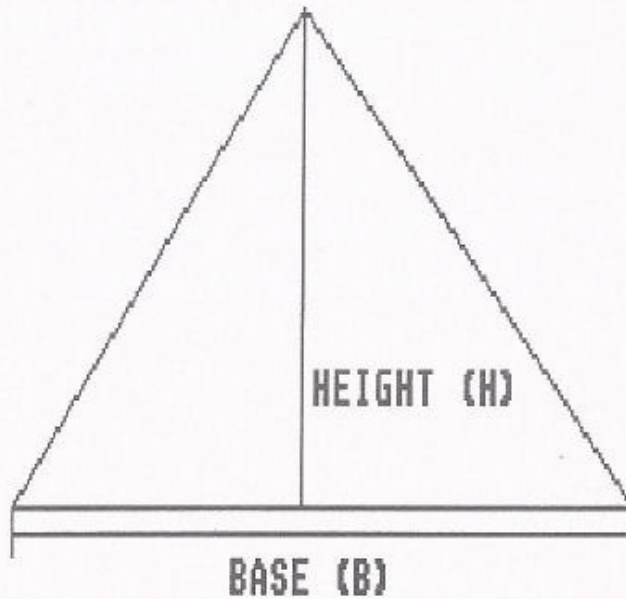
Area of a triangle

To conclude the 'area calculating' programs we shall write a program to find the area of a triangle.

The area of a triangle is calculated by the following formula:

$$0.5 * BASE * HEIGHT$$

The area of a triangle is equal to half the length of the base multiplied by the height as shown over the page.



Load the following:

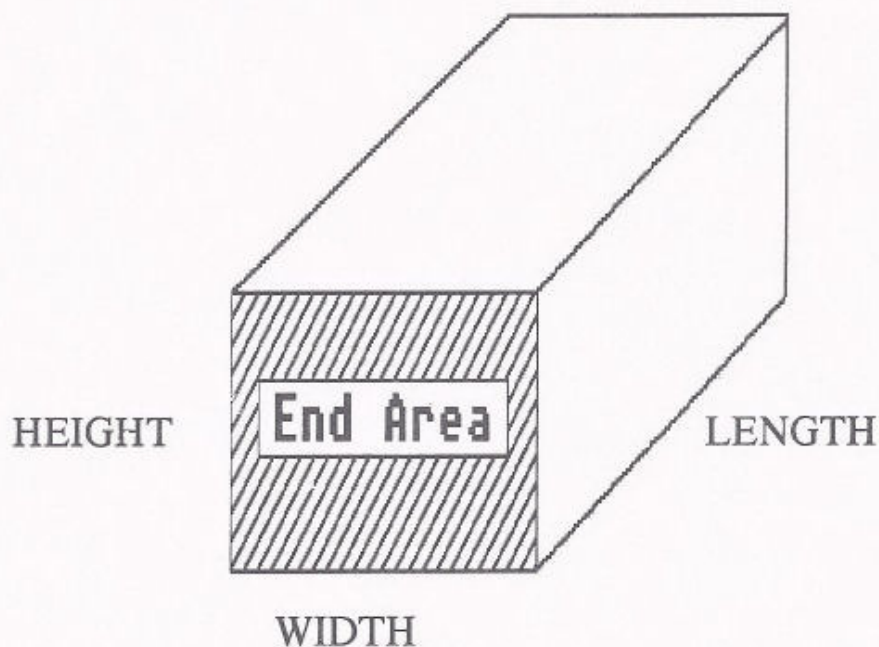
```
10 rem CALCULATE THE AREA OF A TRIANGLE
20 rem PROGRAM = A:\VARIABLE\TRIANGLE
30 :
40 rem INPUT DATA
50 print "Enter the length of the base"
60 input B#
70 print "Enter the height of the triangle"
80 input H#
90 :
100 rem PERFORM CALCULATION
110 AREA# = 0.5 * B# * H#
120 print "The area of the triangle is ";AREA#
```

Look carefully at the calculation in line 110.

CALCULATING VOLUMES

Another area of the GCSE maths syllabus involves the calculation of volumes.

Volume of a cube or cuboid



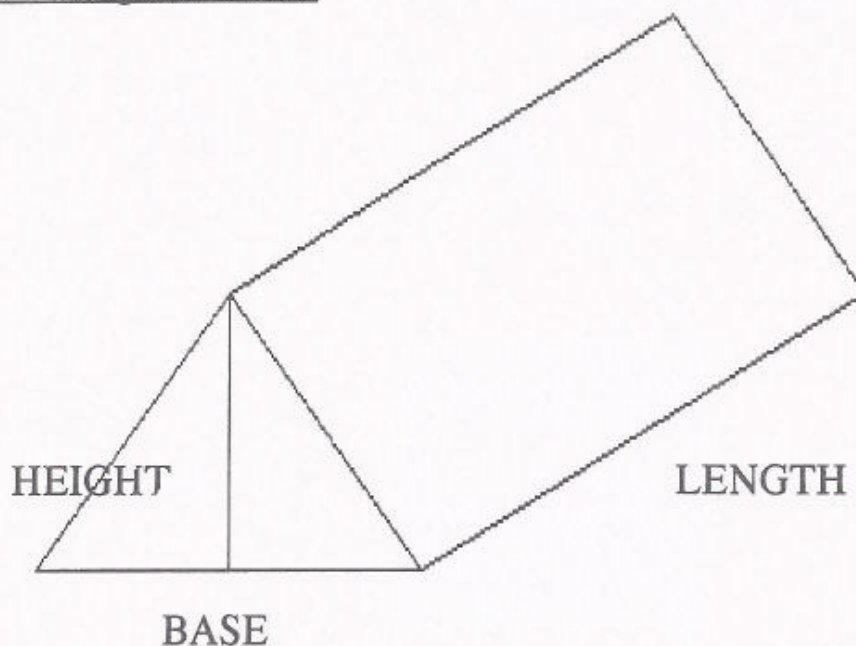
Volume of a cuboid = End Area (area of rectangle) * length
 = Width * Height * Length

```
10 rem CALCULATE VOLUME OF CUBOID
20 rem PROGRAM = A:\NUMERIC\CUBEVOL
30 :
40 rem INPUT DATA
50 print "Enter the width of the cuboid"
60 input W#
70 print "Enter the height of the cuboid"
80 input H#
90 print "Enter the length of the cuboid"
100 input L#
110 :
```



```
120 rem PERFORM CALCULATION
130 V# = W# * H# * L#
140 print "The volume of the cuboid is ";v#
```

Volume of a Triangular Prism

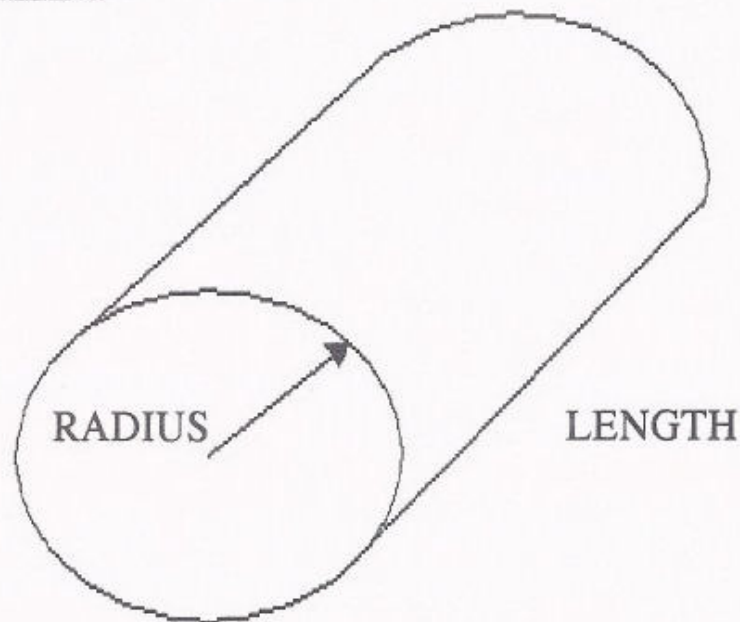


Volume of triangular prism = End Area (area of triangle) *
Length
= $0.5 * \text{Base} * \text{Height} * \text{Length}$

```
10 rem CALCULATE VOL OF TRIANGULAR PRISM
20 rem PROGRAM = A:\NUMERIC\TRIANVOL
30 :
40 rem INPUT DATA
50 print "Enter length of the base"
60 input B#
70 print "Enter height of the triangle"
80 input H#
90 print "Enter length of the prism"
100 input L#
110 :
```

```
120 rem PERFORM CALCULATION
130 V# = 0.5 * B# * H# * L#
140 print "The volume of the triangular prism is ";V#
```

Volume of a Cylinder



Volume of cylinder = End Area (area of circle) * Length
 = $\pi * \text{radius}^2 * \text{Length}$

```
10 rem CALCULATE VOLUME OF CYLINDER
20 rem PROGRAM = A:\NUMERIC\CYLINVOL
30 :
40 rem INPUT DATA
50 print "Enter the radius of the circle"
60 input R#
70 print "Enter the length of the cylinder"
80 input L#
90 :
100 rem PERFORM CALCULATION
110 V# = PI * R#^2 * L#
```



```
120 print "The volume of the cylinder is ";V#
```

PYTHAGORAS' THEOREM

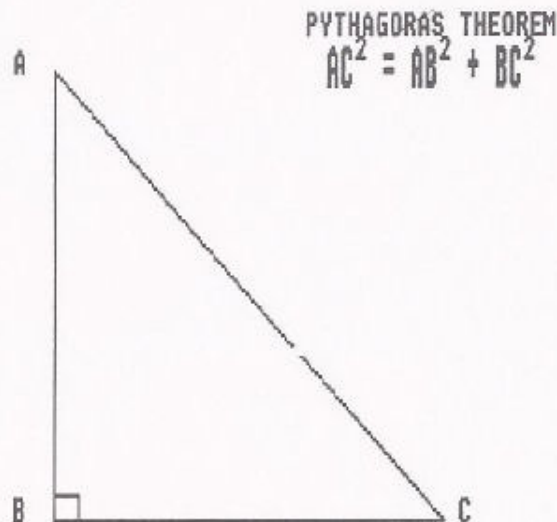
Pythagoras' Theorem states, that for any right-angled triangle, the square of the hypotenuse length is exactly equal to the sum of the squares of the lengths of the other two sides.

Pythagoras' Theorem is used to calculate the lengths of sides of triangles. If the lengths of two sides are known then the length of the third side can be calculated.

From the diagram on the next page, we can construct three programs to calculate the length of each side.

```
10 rem FIND THE LENGTH OF SIDE AC
20 rem PROGRAM = A:\NUMERIC\PYTHAG1
30 :
40 rem INPUT DATA
50 print "Enter length of side AB"
60 input AB#
70 print "Enter length of side BC"
80 input BC#
90 :
100 rem PERFORM CALCULATION
110 AC# = sqr( AB#^2 + BC#^2 )
120 print "Length of side AC = ";AC#
```

```
10 rem FIND THE LENGTH OF SIDE AB
20 rem PROGRAM = A:\NUMERIC\PYTHAG2
30 :
40 rem INPUT DATA
50 print "Enter length of side AC"
60 input AC#
```



$$AC = \sqrt{AB^2 + BC^2}$$

$$AB = \sqrt{AC^2 - BC^2}$$

$$BC = \sqrt{AC^2 - AB^2}$$

```

70 print "Enter length of side BC"
80 input BC#
90 :
100 rem PERFORM CALCULATION
110 AB# = sqr( AC#^2 - BC#^2 )
120 print "Length of side AB = ";AB#

10 rem FIND THE LENGTH OF SIDE BC
20 rem PROGRAM = A:\NUMERIC\PYTHAG3
30 :
40 rem INPUT DATA
50 print "Enter length of side AC"
60 input AC#
70 print "Enter length of side AB"
80 input AB#
90 :
100 rem PERFORM CALCULATION
110 BC# = sqr( AC#^2 - AB#^2 )
120 print "Length of side BC = ";BC#

```


TRIGONOMETRY

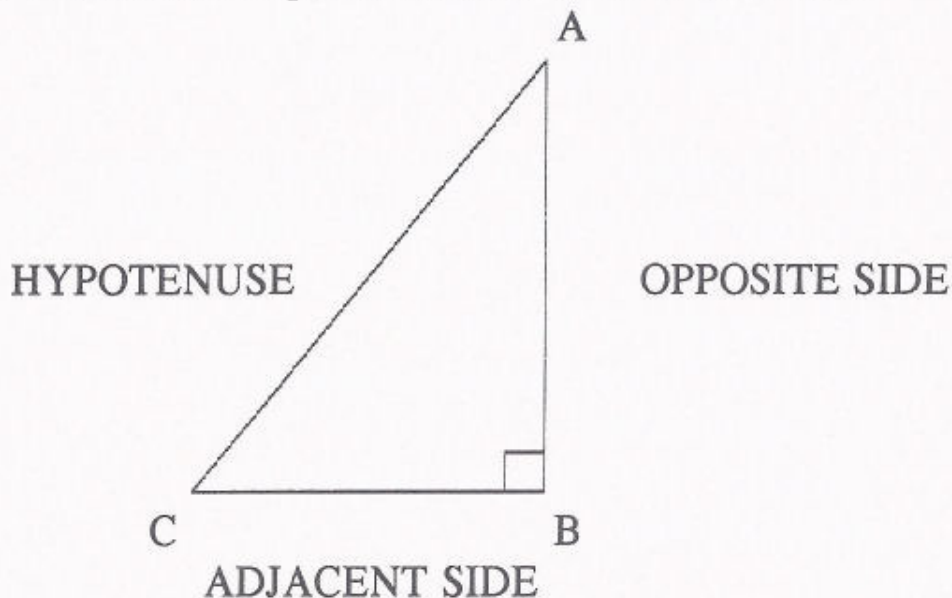
STOS offers a range of trigonometric functions as listed below.

SIN returns the SINE of an angle in radians
 COS returns the COSINE of an angle in radians
 TAN returns the TANGENT of an angle in radians
 ASIN returns the ARCSIN (SIN^{-1})
 ACOS returns the ARCCOS (COS^{-1})
 ATAN returns the ARCTAN (TAN^{-1})
 HSIN returns the hyperbolic sine
 HCOS returns the hyperbolic cosine
 HTAN returns the hyperbolic tangent

We shall limit our calculations to the basic sine, cosine and tangent rules as required for GCSE level.

SINE

The various calculations required for sine are shown below:



$$\text{SIN } C = \frac{\text{OPPOSITE (AB)}}{\text{HYPOTENUSE (AC)}}$$

$$\text{AC} = \frac{\text{AB}}{\text{SIN } C}$$

$$AB = AC * \sin C$$

We shall write separate programs to perform the three calculations.

The first program calculates the angle C given the length of the opposite side (AB) and the length of the hypotenuse (AC). Before writing the program we need to establish the formula. The standard sine rule can be re-arranged to find the formula for angle C as shown below:

$$\sin C = \frac{\text{OPPOSITE (AB)}}{\text{HYPOTENUSE (AC)}}$$

$$\text{therefore } C = \arcsin \frac{\text{OPPOSITE (AB)}}{\text{HYPOTENUSE (AC)}}$$

This can now be applied to a program as shown below:

```
10 rem CALCULATE ANGLE C
20 rem PROGRAM = A:\NUMERIC\SIN1
30 :
40 rem INPUT DATA
50 print "Enter length of opposite side AB"
60 input AB#
70 print "Enter the length of hypotenuse AC"
80 input AC#
90 :
100 rem PERFORM THE CALCULATION
110 C# = asin (AB# / AC#)
120 C# = deg (C#)
130 print "The angle C = "; C#; "degrees"
```

Lines 50-60 request the length of the side AB and assign this to variable AB#. Lines 70-80 request the length of the hypotenuse and

assign this to variable AC#. Notice that real number variables have been used to allow full decimal numbers to be entered. Line 110 performs the calculation and assigns the result to variable C#. The *asin* command returns the answer in radians so line 120 converts the contents of variable C# to degrees using the *deg* function. Line 130 displays the result.

NOTE: on later versions of STOS I have found the *asin* function to be very inaccurate - so much so that it renders the above program unusable. Therefore, if you intend using the above program for serious work, check your answers with a calculator first and compare the results.

The second program will calculate the length of the hypotenuse (AC) given the angle C and the length of the opposite side (AB). The formula can be found as shown below:

$$\text{SIN } C = \frac{\text{OPPOSITE (AB)}}{\text{HYPOTENUSE (AC)}}$$

$$\text{therefore HYPOTENUSE (AC) = } \frac{\text{OPPOSITE (AB)}}{\text{SIN } C}$$

This can now be applied to a program.

```

10 rem CALCULATE THE HYPOTENUSE AC
20 rem PROGRAM = A:\NUMERIC\SIN2
30 :
40 rem INPUT DATA
50 print "Enter length of opposite side AB"
60 input AB#
70 input "Enter the angle C"
80 input C#
90 :
```

```
100 rem PERFORM CALCULATION
110 C# = rad(C#)
120 AC# = AB# / sin(C#)
130 print "The length of hypotenuse AC = "; AC#
```

Lines 50-60 request the length of the opposite side (AB) and assign this to variable AB#. Lines 70-80 request the angle (C) and assign this to variable C#. The *sin* command expects the angle in radians and so the input angle C# must be converted from degrees to radians. This is carried out by the *rad* command in line 110. Line 120 then performs the calculation assigning the result to variable AC#. Line 130 displays the result.

The last of the sine calculations requires the length of the opposite side to be found. The formula is found, as usual, by transposing the standard sine rule as shown below:

$$\frac{\sin C}{\text{HYPOTENUSE (AC)}} = \frac{\text{OPPOSITE (AB)}}{\text{HYPOTENUSE (AC)}}$$

$$\text{therefore OPPOSITE (AB) = HYPOTENUSE (AC) * SIN C}$$

The program can now be written as follows:

```
10 rem CALCULATE THE SIDE AB
20 rem PROGRAM = A:\NUMERIC\SIN3
30 :
40 rem INPUT DATA
50 print "Enter length of hypotenuse AC"
60 input AC#
70 print "Enter angle C"
80 input C#
90 :
100 rem PERFORM CALCULATION
```



```

110 C# = rad(C#)
120 AB# = AC# * sin(C#)
130 print "The length of side AB = "; AB#

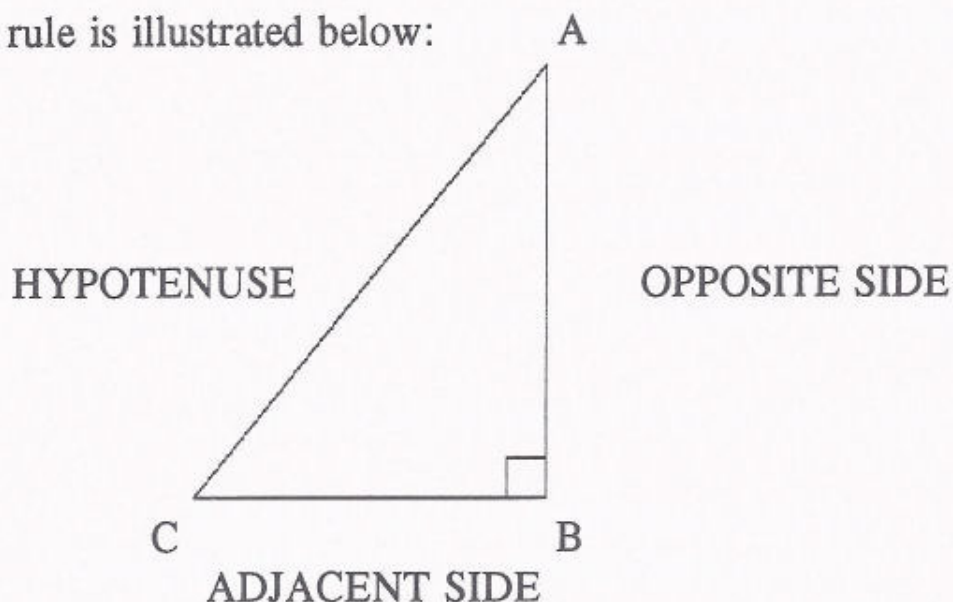
```

Lines 50-60 request the length of the hypotenuse and assign this to variable AC#. Lines 70-80 request the angle C. Remember that the *sin*, *cos* and *tan* commands require the angle in radians and so line 110 converts the entered angle from degrees to radians. Line 120 performs the calculation and assigns the result to variable AB#. Line 130 displays the result.

That concludes the use of sin so we shall now move on to cosine.

COSINE

The cosine rule is illustrated below:



$$\cos C = \frac{\text{ADJACENT (BC)}}{\text{HYPOTENUSE (AC)}} \qquad AC = \frac{BC}{\cos C}$$

$$BC = AC * \cos C$$

As per the last section we shall start by calculating the angle C. The

formula is shown below:

$$\cos C = \frac{\text{ADJACENT (BC)}}{\text{HYPOTENUSE (AC)}}$$

$$\text{therefore } C = \frac{\text{ARCCOS ADJACENT (BC)}}{\text{HYPOTENUSE (AC)}}$$

The program can now be written as shown below:

```

10 rem CALCULATE ANGLE C
20 rem PROGRAM = A:\NUMERIC\COS1
30 :
40 rem INPUT DATA
50 print "Enter length of adjacent side BC"
60 input BC#
70 print "Enter length of hypotenuse AC"
80 input AC#
90 :
100 rem PERFORM CALCULATION
110 C# = acos (BC# / AC#)
120 C# = deg(C#)
130 print "The angle C = "; C#; " degrees"

```

Lines 50-60 request the length of the adjacent side and assign this to variable BC#. Lines 70-80 request the length of the hypotenuse and assign this to variable AC#. Line 110 performs the calculation assigning the result to variable C#. The *acos* command returns the result in radians and hence this must be changed to degrees using the *deg* command as shown in line 120. Line 130 displays the result.

NOTE: on later versions of STOS I have found the *acos* function to be very inaccurate - so much so that it renders the above program unusable. Therefore, if you intend using the above program for

serious work, check your answers with a calculator first and compare the results.

We shall now transpose the cosine rule again to find a formula for the hypotenuse.

$$\cos C = \frac{\text{ADJACENT (BC)}}{\text{HYPOTENUSE (AC)}}$$

$$\text{therefore HYPOTENUSE (AC)} = \frac{\text{ADJACENT (BC)}}{\cos C}$$

Let us write the program.

```

10 rem CALCULATE THE HYPOTENUSE AC
20 rem PROGRAM = A:\NUMERIC\COS2
30 :
40 rem INPUT DATA
50 print "Enter length of adjacent side BC"
60 input BC#
70 print "Enter the angle C"
80 input C#
90 :
100 rem PERFORM CALCULATION
110 C# = rad(C#)
120 AC# = BC# / cos(C#)
130 print "The length of hypotenuse AC = ";AC#

```

Lines 50-60 request the length of the adjacent side and assign this to variable BC#. Lines 70-80 request the angle (C) and assign this to variable C#. Line 110 converts the entered angle to radians and assigns this back to the original variable C#. Line 120 performs the calculation and line 130 displays the result.

The last of the cosine calculations requires the length of the adjacent side to be found. The cosine rule is thus transposed as shown below:

$$\cos C = \frac{\text{ADJACENT (BC)}}{\text{HYPOTENUSE (AC)}}$$

therefore $\text{ADJACENT (BC)} = \text{HYPOTENUSE (AC)} * \cos C$

The program can be written as shown below:

```
10 rem CALCULATE THE ADJACENT SIDE BC
20 rem PROGRAM = A:\NUMERIC\COS3
30 :
40 rem INPUT DATA
50 print "Enter length of hypotenuse AC"
60 input AC#
70 print "Enter the angle C"
80 input C#
90 :
100 rem PERFORM CALCULATION
110 C# = rad(C#)
120 BC# = AC# * cos(C#)
130 print "The length of adjacent side BC = ";BC#
```

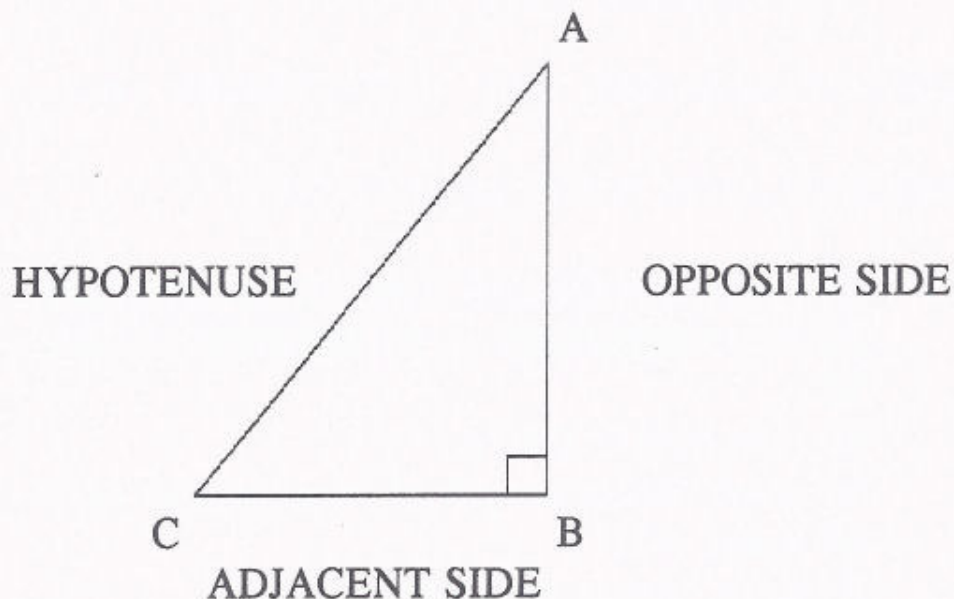
Lines 50-60 request the length of the hypotenuse and assign this to variable AC#. Lines 70-80 request the angle C and assign this to variable C#. Line 110 converts the angle to radians and line 120 performs the calculation. Line 130 displays the result

The final rule is tangent as illustrated over the page. As for the sine and cosine rules, three programs have been produced to perform the various calculations. You should now be able to work out the operation of these for yourself.

$$\text{TAN } C = \frac{\text{OPPOSITE (AB)}}{\text{ADJACENT (BC)}}$$

$$\text{BC} = \frac{\text{AB}}{\text{TAN } C}$$

$$\text{AB} = \text{BC} * \text{TAN } C$$



```

10 rem CALCULATE THE ANGLE C
20 rem PROGRAM = A:\NUMERIC\TAN1
30 :
40 rem INPUT DATA
50 print "Enter length of opposite side AB"
60 input AB#
70 print "Enter length of adjacent side BC"
80 input BC#
90 :
100 rem PERFORM CALCULATION
110 C# = atan(AB# / BC#)
120 C# = deg(C#)
130 print "The angle C = "; C#; " degrees"

```

```
10 rem CALCULATE THE OPPOSITE SIDE AB
20 rem PROGRAM = A:\NUMERIC\TAN2
30 :
40 rem INPUT DATA
50 print "Enter length of adjacent side BC"
60 input BC#
70 print "Enter angle C"
80 input C#
90 :
100 rem PERFORM CALCULATION
110 C# = rad(C#)
120 AB# = BC# * tan(C#)
130 print "The length of the opposite side AB = ";AB#

10 rem CALCULATE THE ADJACENT SIDE BC
20 rem PROGRAM = A:\NUMERIC\TAN3
30 :
40 rem INPUT DATA
50 print "Enter length of opposite side AB"
60 input AB#
70 print "Enter angle C"
80 input C#
90 :
100 rem PERFORM CALCULATION
110 C# = rad(C#)
120 BC# = AB# / tan(C#)
130 print "The length of the adjacent side BC = ";BC#
```

Well that is about it for this chapter. You have seen a number of programs applicable to the GCSE maths syllabus and should have no problems implementing other such formula in to working programs. The calculations performed in the previous programs were quite straightforward, and were in the main, performed using a single line of code. Complicated calculations should be broken down in to a number of smaller calculations.

Chapter 5

Making Decisions

Every day we have to make many decisions, the outcome of which may determine or generate further decisions and actions. For example:

IF you are tired	THEN go to bed
IF it is raining	THEN put up an umbrella
IF you are hungry	THEN eat food

These examples all have the same form:

IF a condition is true **THEN** take action.

This decision making can be implemented on the computer using exactly the same format. STOS provides the *if* and *then* commands which allow decisions to be made and program execution to be controlled. The *if* and *then* commands are always used as a pair and allow us to develop some really powerful programs.

Looking at the first example above, the condition is 'you are tired' and the action is 'go to bed'. If the condition is true then you take the action. So let us see how such decision making can be applied on the

computer.

Consider a program that may be of use to a local football club. The club has a junior team comprising boys under the age of twelve, and when the coach needs new players, he must check their ages to ensure that they are under twelve. He needs a program that will input the boys ages and make a decision whether or not the boy is acceptable.

Enter the following:

```
10 print "Enter the boys age"  
20 input AGE  
30 if AGE < 12 then print "Under 12 - ACCEPTED"
```

Run the program and enter an age of 13:

```
run  
Enter the boys age  
? 13
```

Run the program again and enter an age of 11:

```
run  
Enter the boys age  
? 11  
Under 12 - ACCEPTED
```

As you can see, the boy is only accepted when aged below twelve.

Lines 10-20 request the boys age and assign this to variable AGE. Line 30 decides whether the boy is under the age of twelve. The *if* and *then* commands are used to make decisions as shown below:

IF a condition is true THEN take action

```
IF age < 12          THEN print "Under 12 - Accepted"
```

The ' $<$ ' symbol means is less than, and thus if the contents of variable AGE are less than twelve, the message is printed. If the condition is not met, e.g. the boys age is not under twelve, then the program moves on to the next line without executing the commands that following the *then* command.

ALTERNATIVE ACTIONS

Sometimes we may want to carry out an alternative action if the condition is not true. This is illustrated below:

```
IF you are tired      THEN go to bed      ELSE watch TV
```

If you are tired then go bed. If you are not tired then you can watch the television.

We could modify the last program so that it also displayed a rejection message when the boys age was greater than twelve. Let us replace the line that makes the decision.

Enter the following:

```
30 if AGE < 12 then print "Under 12 - ACCEPTED"  
else print "Over 12 - REJECTED"
```

List the program to confirm that the new line 30 has been added.

```
list  
10 print "Enter the boys age"  
20 input AGE  
30 if AGE > 12 then print "Under 12 - ACCEPTED" else  
print "Over 12 - Rejected"
```


Now run the program and enter an age of 13:

```
run
Enter the boys age
? 13
Over 12 - REJECTED
```

Run the program again entering an age of 11:

```
run
Enter the boys AGE
? 11
Under 12 - ACCEPTED
```

The *else* command allows the alternative action to be added. If the boys age is less than twelve then print an acceptance message else (the boys age is not less than twelve) print a rejection message.

The last program used the 'is less than' (<) condition to decide on the message to print. STOS, in common with most dialects of Basic, offers six conditions as listed below:

<	is less than
>	is greater than
=	is equal to
< >	is not equal to
< =	is less than or equal to
> =	is greater than or equal to

Let us write another program that simulates the flipping of a coin. There are two possible outcomes when you flip a coin. The coin can land with the head side facing upwards or the tails side facing

upwards. We can simulate this by choosing one of two values at random.

Enter the following:

```
10 C = rnd(1)
20 if C = 0 then print "HEADS" else print "TAILS"
```

Run the program a number of times and it should display either HEADS or TAILS.

Line 10 introduces the *rnd* command. The format of this is shown below:

```
C = rnd (N)
```

where N specifies the upper limit of the range of numbers that can be selected. The command will thus return a random number in the range 0-N.

Looking back to the program, line 10 picks a random number between 0 and 1 and assigns this to variable C. Line 20 makes a decision by checking the value of variable C. If C is equal to zero then the "HEADS" message is displayed else variable C must be equal to one and hence the "TAILS" message is displayed.

The *if* and *then* commands are very useful for checking that a user has entered the correct information. For example, consider the following segment of code:

```
10 print "Enter a number between 1 and 100"
20 input N
30 :
40 if N < 1 or N > 100 then print "NUMBER OUTSIDE
RANGE"
```


Look carefully at line 40. We are testing this time for two conditions. The *or* function allows multiple conditions to be linked and thus the message "NUMBER OUTSIDE RANGE" will be displayed if the value represented by variable N is less than one OR greater than one hundred.

We can also check multiple conditions using the *and* function as shown below:

```
10 print "Enter a number between 1 and 100"
20 input N
30 :
40 if N >= 1 and N <= 100 then ?"NUMBER
ACCEPTABLE"
```

This time line 40 checks that the value represented by variable N is greater than or equal to one AND smaller than or equal to one hundred.

We can also base decisions on text information. For example:

```
500 print "Would you like to play again (YES or NO)"
510 input PLAY$
520 if PLAY$ = "NO" then end
530
```

If the user enters "NO" the program ends, but if the user enters "YES" the program passes on to line 530.

Most programs require some degree of decision making and you will be seeing a lot more of the *if* and *then* commands throughout the rest of the course.

Chapter 6

Loops & Program Control

In all of the programs we have met so far, the computer executes the commands one after the other in line number order until it reaches the end of the program. This is fine for many programs but consider the following problem.

We need a program that could be used each week to calculate the amount of pay due to employees of a Company. Each employee earns £3 per hour and a program to tackle this function is shown below.

Load the following:

```
10 rem CALCULATE EMPLOYEES PAY
20 rem PROGRAM = A:\LOOPS\PAY1
30 :
40 print "How many hours have been worked"
50 input HOURS
60 PAY = HOURS * 3
70 print "The amount of pay due is £"; PAY
```


Run the program and enter a value for the hours.

```
run
How many hours have been worked
? 40
The amount of pay due is £120
```

The program multiplies the number of hours worked by the hourly rate of pay to generate the total pay due. To calculate the pay for the whole work force would require the program to be run several times, entering the appropriate number of hours that each employee has worked. Running the program time after time would prove rather laborious and it would be far more convenient if we could run the program just once, and during that run, get the computer to execute a series of commands over and over again until we tell it to stop.

REPEAT..UNTIL LOOPS

The repeating of commands is known as *looping* and STOS offers three methods of producing loops. The first of these is the *repeat..until* loop.

```
REPEAT
group of commands
UNTIL told to stop
```

A *repeat..until* loop can be added to the previous program.

Load the following:

```
10 rem CALCULATE EMPLOYEES PAY 2
20 rem PROGRAM = A:\LOOPS\PAY2
30 :
40 repeat
```

```
50 print "How many hours have been worked"
60 input HOURS
70 PAY = HOURS * 3
80 print "The amount of pay due is £"; PAY
90 until HOURS = 0
```

Run the program and enter a couple of different values for the hours.

```
run
How many hours have been worked
? 10
The amount of pay due is £30
How many hours have been worked
? 50
The amount of pay due is £150
How many hours have been worked
? 0
The amount of pay due is £0
```

When a value of zero is entered, the program ends.

The *repeat* command at line 40 instructs the computer to execute over and over again all the commands between line 40 and the *until* command in line 90 and thus lines 50,60,70 and 80 form the group of commands to be repeated. Looking back to the previous definition we need some way of telling the computer to stop the repetition. An employee who works no hours will receive no pay and consequently we can use a value of zero to instruct the computer to stop repeating the commands. The program keeps repeating until the value of variable HOURS equals the value zero. When HOURS=0 the program moves on to the next line after the *until* command. In this example we do not have any further lines and so the program ends.

Let us look at another example. The next program continually adds the entered numbers. When a value of zero is entered, the program

displays the total.

Load the following:

```
10 rem CALCULATOR
20 rem PROGRAM = A:\LOOPS\CALC
30 :
40 repeat
50 print "Enter a number"
60 input N#
70 T# = T# + N#
80 until N# = 0
90 :
100 print "The total is "; T#
```

Run the program and enter a few numbers.

```
run
Enter a number
? 10
Enter a number
? 5
Enter a number
? 15
Enter a number
? 0
The total is 30
```

The group of commands between the *repeat* and *until* commands are repeated. When a value of zero is entered, the program passes on to line 100 where the total is displayed.

WHILE..WEND LOOPS

Another form of loop very similar to *repeat..until* is *while..wend*.

REPEAT

group of commands

UNTIL a condition is true

WHILE a condition is true

group of commands

WEND

The *repeat..until* loop repeats a group of commands UNTIL a certain condition is true.

The *while..wend* loop repeats a group of commands WHILE a condition is true.

In many situations both the *repeat..until* and the *while..wend* loop can be used to perform the same operation. For example, the two programs shown below produce exactly the same result.

- (1) 100 print "Press Y to continue"
 110 repeat
 120 input I\$
 140 until I\$ = "Y"

- (2) 100 print "Press Y to continue"
 110 while I\$ < > "Y"
 120 input I\$
 130 wend

Both programs keep looping until the letter "Y" is entered. So, if both forms of the loop produce the same result, why does STOS offer the two alternatives ? Well, let us look at another example which should

illustrate this.

Load the following:

```
10 rem WHILE..WEND LOOPS
20 rem PROGRAM = A:\LOOPS\WHILE
30 :
40 Print Enter a number (0 or 1)"
50 input N
60 :
60 while N = 1
70 print "Hello"
80 wend
```

Run the program and enter the number 1. This should result in the word "Hello" being continuously printed to the screen.

The entered number is assigned to variable N which is subsequently used to determine the state of the *while..wend* loop. While the value of variable N is equal to one, the loop will continue printing the word "Hello"

Stop the program by pressing Control + C

Run the program again and enter the value zero.

This time nothing is displayed. The condition at line 60 (N=1) is not satisfied and hence the contents of the loop are not executed.

Let us now consider the same operation carried out using a *repeat..until* loop.

Enter or load the following:

```
10 rem REPEAT..UNTIL LOOPS
```

```
20 rem PROGRAM = A:\LOOPS\REPEAT
30 :
40 print "Enter a number (0 or 1)"
50 input N
60 :
70 repeat
80 print "Hello"
90 until N = 0
```

Run the program and enter a value of 1.

The word "Hello" is continuously printed as per the previous program.

Stop the program using Control + C

Run the program again and enter the value zero. This time the word "Hello" is printed once and the program ends. Therefore, when a value of zero is entered, the *while..wend* loop produces a different result to the *repeat..until* loop. The program containing the *while..wend* loop does not print a message but the program containing the *repeat..until* loop does still display the message once.

In most situations both the *repeat..until* and the *while..wend* loop can be used to perform the same task. There is though a major difference between the two that could effect a programs operation. The *repeat..until* loop checks the condition at the end of the loop using the *until* command whilst the *while..wend* loop carries out the check at the start of the loop using the *while* command. This means that the *repeat..until* loop will always be executed once irrespective of the condition, and hence in the last example, the word "Hello" was printed once even though the terminating condition was satisfied.

FOR..NEXT LOOPS

Repeat..Until and *while..wend* loops are used to repeatedly execute a group of commands until or while some condition is satisfied. There are times when we want to execute a group of commands a fixed number of times without depending on a terminating condition, and in this instance, the *for..next* loop is used.

```
FOR variable = start TO finish
program commands
NEXT variable
```

The commands between the *for* and *next* commands are repeated a precise number of times as specified by the start and ending values of the variable.

Load the following:

```
10 rem DISPLAY HELLO
20 rem PROGRAM = A:\LOOPS\HELLO
30 :
40 for A = 1 to 3
50 print "Hello"
60 next A
```

Run the program.

```
run
Hello
Hello
Hello
```

The program prints the word "Hello" three times.

The *for* command in line 10 is followed by a variable name. The

variable name A has been used for this example but any numeric variable can be used. Following the variable name is the range, which in this case is one to three. The computer therefore executes the loop as follows:

(1) The loop variable A is assigned the value 1 and line 20 is then executed. Line 30 contains the *next* command which sends program execution back to the line containing the *for* command (line 10).

(2) The loop variable A is increased by one ($A=2$) and the commands within the loop (in this case line 20) are executed again.

(3) The *next* command at line 30 sends the program back to line 10 where the loop variable is increased again ($A=3$). Line 20 is executed again.

(4) The *next* command at line 30 sends the program back to line 10 where the loop variable increases again ($A=4$). This time the loop variable is greater than the specified end value (3) and hence the loop is terminated and the program moves on to the next line after the *next* command.

Note that the loop variable A represents the value 4 at the end of the loop. This is one more than the ending value (3) specified as part of the *for* command.

We can see how the loop variable is increased by printing it within a *for..next* loop.

Load the following:

```
10 rem DISPLAY LOOP VARIABLE
20 rem PROGRAM = A:\LOOPS\VARIABLE
30 :
40 for Z = 1 to 5
```



```
50 print "The loop variable Z = ";Z
60 next Z
70 :
80 print Z
```

Run the program

```
run
The loop variable Z = 1
The loop variable Z = 2
The loop variable Z = 3
The loop variable Z = 4
The loop variable Z = 5
6
```

Each time the program loops it displays the current value of the loop variable. Finally, the value of variable Z is printed. Notice, as previously stated, that the value of the loop variable Z is one higher than the range specified in the loop.

We could use a *for..next* loop to display the ten times table.

Load the following:

```
10 rem TEN TIMES TABLE
20 rem PROGRAM: A:\LOOPS\TABLES1
30 :
40 for COUNT = 1 to 12
50 ANSWER = COUNT * 10
60 print COUNT; " x 10 = "; ANSWER
70 next COUNT
```

Run the program.

```
run
1 x 10 = 10
2 x 10 = 20
3 x 10 = 30
4 x 10 = 40
5 x 10 = 50
6 x 10 = 60
7 x 10 = 70
8 x 10 = 80
9 x 10 = 90
10 x 10 = 100
11 x 10 = 110
12 x 10 = 120
```

The loop variable is assigned the values 1 to 12 and hence the program repeats 12 times.

The program could be made to display any table by altering lines 50 and 60 or it could be expanded to display any table specified by the user.

Load the following:

```
10 rem DISPLAY TABLES
20 rem PROGRAM = A:\LOOPS\TABLES2
30 :
40 rem INPUT REQUIRED TABLE
50 print "What table would you like ?"
60 input T
70 :
80 rem DISPLAY TABLE
90 for L = 1 to 12
100 ANSWER = L * T
110 print L' " x ";T;" = ";ANSWER
120 next L
```


Run the program and choose the required table.

Notice how the variable T is used within the *for..next* loop to perform the calculation.

The *for..next* loop does not have to start at a value of one. For example, the following three programs all produce the same result.

- (1) 10 for X = 1 to 5
 20 print "GOOD MORNING"
 30 next X

- (2) 10 for X = 30 to 34
 20 print "GOOD MORNING"
 30 next X

- (3) 10 for X = -9 to -5
 20 print "GOOD MORNING"
 30 next X

Variables can be used to represent the range. For example:

```
10 START = 1
20 FINISH = 25
30 for Y = START to FINISH
40 print "STOS"
50 next Y
```

The loop does not have to step in units of one either. For example:

```
10 for Z = 0 to 8 step 2
20 print Z
30 next Z
```

The program produces:

0
2
4
6
8

The *step* command is used to specify the required increment.

Loops can also be placed inside loops producing *nested loops*.

Load the following:

```
10 rem DISPLAY ALL TABLES
20 rem PROGRAM = A:\LOOPS\TABLES3
30 :
40 for X = 1 to 12
50 for Y = 1 to 12
60 print X; "x "; Y; " = "; X * Y
70 next Y
80 print
90 next X
```

Run the program and you will find that it displays all of the multiplication tables in the range 1-12.

USING A LOOP TO DISPLAY ASCII CODES

In chapter 1 we saw how to display information using ascii codes. These allow information to be displayed that is not normally available on the keyboard and which cannot be displayed using a direct *print* command.

The following program uses *for..next* loops to display a list of the ascii codes. The list starts at chr\$(32) as previous codes are used by the system and cannot be displayed by the user.

Load the following:

```
10 rem DISPLAY THE ASCII CODES
20 rem PROGRAM = A:\LOOPS\ASCII
30 :
40 rem SET SCREEN
50 key off : mode 0
60 :
70 for A=32 to 255 step 10
80 :
90 for B=1 to 10
100 C=A+B
110 print C,chr$(C)
120 print
130 next B
140 print "Press any key to continue"
150 wait key
160 :
170 next A
```

Run the program and it will cycle through the codes.

PROGRAM CONTROL

We have just seen how loops can be used to repeat sections of code. Parts of a program can be repeated whilst a condition is true, until a condition is true or for a specific number of times. A program is normally executed in line number order but loops allow us to alter or control the execution of the program. STOS offers further commands for controlling program execution as we shall see now.

The GOTO command

The *goto* command passes control to a specific line without executing any of the lines in between.

Enter the following:

```
10 print "Enter a number"  
20 input N  
30 print "You entered the number ";N  
40 goto 10
```

Run the program and you will be asked to enter a number. Enter a number and press the return key. The program will then ask you to enter another number and another number and another number. The program can be terminated by pressing CONTROL+C.

The *goto* command in line 40 sends the program directly back to line 10 without executing any of the commands in between.

A variable can also be used to represent the line number as illustrated by the following program:

```
100 L=1850  
110 goto L
```

Passing control using IF..THEN

In the previous chapter you saw how the *if* and *then* commands are used to make decisions. As the result of a decision we may wish to pass program execution to a specific line and thus the *goto* command can be incorporated in to the *if..then* test.

```
280 if X=1 then 1000
```


If the value of variable X is equal to one then program execution is passed directly to line 1000.

This is a powerful facility as it allows the programmer to pass control to a specific area of the program based on the result of a decision.

Load the following:

```
10 rem PROGRAM CONTROL USING IF..THEN
20 rem PROGRAM = A:\LOOPS\CONTROL1
30 :
40 rem DISPLAY OPTIONS
50 print "1..Start Game"
60 print "2..Set Level"
70 print "3..Display High Scores"
80 print "Enter option"
90 input OPT
100:
110 rem PASS CONTROL
120 if OPT = 1 then 1000
130 if OPT = 2 then 2000
140 if OPT = 3 then 3000
150 :
160 :
1000 print "Start Game":end
2000 print "Set Level":end
3000 print "Display High Scores":end
```

Run the program a few times and enter each of the options 1-3. In this example the program displays a simple message, but in a real application, lines 1000 on, 2000 on and 3000 on would contain the code to perform the selected operation.

The entered number is assigned to variable OPT and a series of *if..then* commands are used to pass control to the appropriate area of

the program.

A test must be made for each option and the longer the list of options so the longer the list of tests that will be required. STOS therefore allows all the tests to be incorporated in to one line of code.

The ON..GOTO command

The *on..goto* command allows program execution to be passed to a specific line depending on the value of a variable. The last program can be re-written using this method.

Load the following:

```
10 rem PROGRAM CONTROL USING ON..GOTO
20 rem PROGRAM = A:\LOOPS\CONTROL2
30 :
40 rem DISPLAY OPTIONS
50 print "1..Start Game"
60 print "2..Set Level"
70 print "3..Display High Scores"
80 print "Enter option"
90 input OPT
100:
110 rem PASS CONTROL
120 on OPT goto 1000, 2000, 3000
130 :
140 :
1000 print "Start Game": end
2000 print "Set Level": end
3000 print "Display High Scores": end
```

Run the program and you will find that it performs the same function as the previous example.

Notice now that all of the *if..then* tests have been incorporated in to a single line. The format of the *on..goto* command is shown below:

on VARIABLE goto LINE 1, LINE 2, LINE 3.....

VARIABLE is the controlling variable, the value of which determines the line to which execution is passed.

LINE 1,2,3, etc. are a list of line numbers that the program jumps to depending on the value of the controlling variable.

When VARIABLE = 1 the program jumps to LINE 1

When VARIABLE = 2 the program jumps to LINE 2

When VARIABLE = 3 the program jumps to LINE 3

So, looking back to the program:

If the user enters 1, the program jumps to line 1000

If the user enters 2, the program jumps to line 2000

If the user enters 3, the program jumps to line 3000

SUB-ROUTINES

Suppose that during a program we wish to ask the user to "Press any key to continue". The following segment of code would perform this task.

```
10 print "Press any key to continue"  
20 wait key
```

Suppose though that we needed to perform this operation ten times throughout the program. We could enter the above code each time that the operation was required or we could produce a subroutine. A subroutine is a section of code, or a routine, that is entered once and

that can be called upon as many times as required.

Load the following:

```
10 rem SUBROUTINES
20 rem PROGRAM = A:\LOOPS\SUB
30 :
40 print "Hello and how are you today"
50 gosub 1000
60 print "Do you like computers"
70 gosub 1000
80 end

1000 print "Press any key to continue"
1020 wait key
1030 return
```

Run the program and it will produce the following:

```
Hello and how are you today
Press any key to continue      [ press a key ]
Do you like computers
Press any key to continue      [ press a key ]
Ok
```

Notice that the "Press any key to continue" message has been displayed twice even though it has only been entered in the program once.

Lines 1000-1030 of the program form a subroutine.

Line 40 displays a message.

Line 50 *calls* the subroutine. The *gosub* command (goto subroutine) passes program execution directly to line 1000. The program

continues executing the commands until it reaches the *return* command which returns control back to the calling program. The program jumps back to the next line after the original *gosub* command which in this case is line 60.

Line 60 displays another message and line 70 calls the subroutine a second time.

You can see that, although the subroutine has been entered only once, it can be called upon as many times as required during the program.

The ON..GOSUB command

This is very similar to the *on..goto* command except that control is passed to a subroutine rather than just a line number.

Load the following:

```
10 rem PROGRAM CONTROL USING ON..GOSUB
20 rem PROGRAM = A:\LOOPS\CONTROL3
30 :
40 rem DISPLAY OPTIONS
50 print "1..Start Game"
60 print "2..Set Level"
70 print "3..Display High Scores"
80 print "Enter option"
90 input OPT
100:
110 rem PASS CONTROL
120 on OPT gosub 1000, 2000, 3000
130 goto 40
140 :
150 :
1000 print "Start Game": return
```

```
2000 print "Set Level": return  
3000 print "Display High Scores": return
```

Run the program and select various options in the range 1-3. Notice how the program jumps back and re-displays the menu each time. Press CONTROL + C to terminate the program.

In this example each of the options are programmed as subroutines. When the user selects an option, program control jumps to the start of the appropriate subroutine, and once the option has been serviced, execution is passed back to the calling program using the *return* command. In this example the program jumps back to line 130 which subsequently displays the menu so that the user may make another selection.

Subroutines allow programs to be constructed in modular fashion. They allow sections of code to be repeated and make the task of programming easier. We shall be using the loop and program control techniques quite a lot during the rest of the course so you will have adequate chance to explore these further.

We shall now progress on to chapter 7 where we shall use the various techniques discussed so far to produce our first, complete game.

Chapter 7

Guess the Number Game

Let us start this chapter with a very bold statement.

WRITING A PROGRAM IS EASY

Whilst you may not currently agree with this statement you will find that most programmers will say the same. The hard part of programming is the definition, design and planning rather than the final coding which should be quite straightforward if the program has been well planned.

Writing a program involves a lot more than simply typing commands in to the computer. Commercial companies employ staff who analysis the requirements and develop program plans which are then used by the programmers to design and enter the code. If a program is designed properly, the programmer should end up with a plan that makes coding the program a very simple task indeed - the better the planning the easier the program to code.

The temptation to sit straight down at the computer with a new idea is often difficult to resist. Whilst it is ok to experiment with a few ideas, it is lethal to attempt a major programming project as this will invariably lead to tears and frustration.

The planning and design process allows the programmer to work out the exact aims and functions of the program, overcoming potential operational and programming problems. Only when the definition is complete can we start to consider the actual program operation. So let us see what is involved in designing a program.

There are three main design stages as listed below:

- (1) DEFINITION: define the exact operation of the program.
- (2) DESIGN: decide how the program should be constructed and list the individual program modules.
- (3) CONSTRUCTION AND CODING: construct the program on paper, enter it in to the computer and locate and rectify any errors.

(1) DEFINING PROGRAM OPERATION

Before we start thinking about designing a program we need to define its operation. What will the game do, what is the objective of the player/s, how many lives will the player have, what screen resolution will be required, how many space ships will be displayed on the screen at any one time, what size will the space ships be, what colours should be used, etc., etc.

As you start to define these various options and operations you will start to encounter problems that may not have been apparent at the outset. One area does not work with another, the game play will be too hard, etc. It is easier to think about and rectify such problems at

the development stage than later when trying to enter the program code. A good approach in these early design stages is to ask yourself 'What if?' - what if the user presses the wrong keys, what if the user does not have a joystick, what if the user removes the disk when the program is running, etc.

(2) DESIGN

Once the exact requirements and operation have been established you can start to think about the design of the program. Having completed the program definition you will know what you want the program to do and can start thinking about the ways in which this can be achieved. Rather than considering the program as a whole, consider it as a series of small modules that interface together to perform the overall task. Each module will perform a specific function. For example, one module may set the screen resolution and colour palette whilst another module may make a space ship explode. This modular approach allows the designer to consider one area at a time and also simplifies the process of coding and debugging (finding faults) at a later stage.

(3) CONSTRUCTION AND CODING

At the end of the design stage you should have a list of separate operations or modules which can now be converted to program code. Before moving on to the computer, the program should be constructed using pen and paper. Some programmers will write the Basic code out directly whilst others prefer to write the operations out in standard English first (often referred to as pseudo code). There are no hard and fast rules as to which method you should use but I have found a combination of the two to be the most productive. If you are unsure how to code a particular area then write the series of operations out using plain English first.

Remember that the whole aim of development is to provide a thorough understanding of what the program must do and to make the task of coding as easy as possible for YOU. For all but the smallest of programs it is advisable to write the program out first. Apart from clarifying the exact operation and series of events it also makes fault finding easier. It is very rare indeed for a program to work first time as there may be bugs (errors) introduced through misspelt words and commands, or there may be operational bugs due to a section of code producing different results to those expected. It does not matter how much time and effort goes in to planning, you must expect some problems during coding. When an error occurs it can be quite difficult tracing through the program listing displayed on the screen. Looking back through your design notes will allow the problem to be located quicker and hence is another plus point for extensive planning and design.

To illustrate the planning and design concepts we shall now apply these to a practical program. We shall develop a simple number guessing game where the computer chooses a random number and the user has to guess the number.

NUMBER GUESSING GAME

(1) DEFINITION

Step one is to define the exact requirements of the program.

The computer will choose a random number between one and one hundred and the user must try and guess the number. Each time the user makes a guess, the computer will check this against the random number and will indicate whether the guess is too low or too high. This process will continue until the correct number is guessed. The computer will then display the number of goes that the user took to guess the number and the program will end.

That defines the required operation quite clearly.

(2) DESIGN

The definition can now be broken down in to a number of operational modules.

- (1) Set screen resolution and display instructions.
- (2) Choose a random number between 1 and 100.
- (3) Ask user for guess.
- (4) Check the entered guess against the number and report back.
- (5) Number correctly guessed so display the number of goes and end.

The program has been separated in to five main areas and we can now consider the construction and coding of the program.

(3) CONSTRUCTION AND CODING

For this example the program has been written out in pseudo code. Notice how plain English has been used to construct the program in such a way that can be easily translated to Basic code.

- (1)
Set screen resolution
Display title and instructions
- (2)
Choose random number between 1 and 100
- (3)
REPEAT
 Increment the total number of guesses
 Ask the user to enter their guess

(4)

IF the guess is lower than the number THEN display "too low"

IF the guess is higher than the number THEN display "too high"

UNTIL the guess equals the number

(5)

Display the number of guesses

END

Remember that this production of pseudo code is not essential and we could have written the program out direct.

At last we come to the interesting bit. The code can now be produced by relating directly to the construction information. Let us code the program then.

Load the following:

```
10 rem GUESS THE NUMBER GAME
20 rem PROGRAM = A:\NUMGAME\NUMGAME
30 :
```

(1)

```
40 rem DISPLAY INSTRUCTIONS
50 key off: mode 0
60 under on
70 centre "GUESS THE NUMBER"
80 under off
90 print
100 print
110 print "I will think of a number between"
120 print "1 and 100 and you must guess the"
130 print "the number."
140 :
```

```
(2) 150 rem CHOOSE THE RANDOM NUMBER
    160 NUMBER = rnd(100)
    170 :

(3) 180 repeat
    190 :
    200 rem ASK USER TO ENTER GUESS
    210 inc NUMBER_OF_GUESSES
    220 print
    230 print "Enter your guess"
    240 input GUESS
    250 :

(4) 260 rem COMPARE GUESS WITH COMPUTERS
    NUMBER
    270 if GUESS < NUMBER then print "TOO LOW"
    280 if GUESS > NUMBER then print "TOO HIGH"
    290 :
    300 until GUESS = NUMBER
    310 :

(5) 320 print
    330 print "WELL DONE"
    340 print "You took "; NUMBER_OF_GUESSES; "
guesses"
```

Run the program and play the game a few times.

As you can see, the program complies exactly with the construction notes.

As previously stated, it is rare for a program to work first time. You have been spoilt on this occasion as the program was supplied for you and thus contains no bugs. Irrespective of your typing skills we all make mistakes when typing in programs. Luckily STOS normally

points us to such errors by issuing a Syntax Error or similar. Operational errors on the other hand are a little harder to trace. It may be that a certain routine does not operate as originally intended and hence we have to refer back to the development notes to discover the cause. Whilst it is impossible to eliminate problems completely, a little time and effort spent defining, planning and constructing the program really will minimise any later headaches.

The program uses a loop and decision making so we had better take a closer look at its operation.

Lines 10 and 20 contain remarks indicating the nature of the program and its location on the disk.

Lines 40-130 display the instructions. Lines 60-80 display the title which is centred using the *centre* command and underlined using the *under on* and *under off* commands.

Line 160 selects a random number in the range 0-100 and assigns this to the variable NUMBER.

Lines 180-300 form a *repeat..until* loop which keeps repeating until the entered guess is correct. Line 210 increments the variable NUMBER_OF_GUESSES which maintains the number of guesses that the user has made. Lines 230-240 request the users guess and assign this to variable GUESS. Line 270 checks to see if the users guess is lower than the secret number and line 280 checks to see if the users guess is higher than the secret number. Line 300 continues the loop until such time that the correct number is guessed.

Lines 320-340 display the final number of guesses.

MAKING THE GAME HARDER

The game could be made harder by changing the range of the random number in line 160. You could, for example, make the program choose a random number in the range 0-1000. Line 160 would thus be changed to the following:

```
160 NUMBER = rnd(1000)
```

Chapter 8

Maths Quiz Game

In this chapter we shall develop and code another program that brings together the loop and decision making facilities.

Can you remember the three development phases ?

- (1) DEFINITION: define the exact operation of the program.
- (2) DESIGN: decide how the program should be constructed.
- (3) CONSTRUCTION AND CODING: construct the code, enter the program and debug.

(1) DEFINITION

The program will display a series of mathematical questions (addition or subtraction) and the user must enter the answer. The user will enter the number of questions required and at the end of the program the computer will display how many of these the user answered correctly. The numbers for the questions will be chosen at random and shall be in the range zero to twenty.

Now we know exactly what we wish to achieve we can move on to the design where we decide on the individual modules.

(2) DESIGN

- (1) Set screen resolution and display title.
- (2) Ask user which type of question they require.
- (3) Ask user how many questions they require and initiate a loop.
- (4) Choose the random numbers and display the question.
- (5) Input and check the answer.
- (6) Display the final score.

The program has been separated in to six sections and can now be constructed.

(3) CONSTRUCTION AND CODING

The program has first been listed in plain English although we shall no longer worry about this stage in the rest of the programs throughout this course.

- (1) Set screen resolution
 Display title
- (2) Ask user which type of question they require (TYPE)
- (3) Ask user how many questions they require (QUESTIONS)
 FOR loop equals one to total number of questions
- (4) Choose two random numbers (N1 and N2)
 Display the question

- (5) Ask user for answer
 IF the answer is correct THEN display "CORRECT" and
increment score
 IF the answer is incorrect THEN display "WRONG"
NEXT loop
- (6) Display final score
 END

Finally the program can be coded.

Load the following:

```
10 rem MATHS QUIZ
20 rem PROGRAM = A:\MATHQUIZ\MATHS
30 :
(1) 40 rem DISPLAY TITLE
50 key off : mode 0
60 under on
70 centre "MATHS QUIZ"
80 under off
90 :
(2) 100 rem WHAT TYPE OF QUESTION
110 print
120 print
130 print "(1) ADDITION"
140 print "(2) SUBTRACTION"
150 print
160 print "Enter required option (1-2)"
170 input TYPE
180 if TYPE < 1 or TYPE > 2 then 160
190 :
(3) 200 rem HOW MANY QUESTIONS
210 print
220 print "How many questions would you like"
```



```
230 input QUESTIONS
240 :
250 rem CREATE LOOP
260 for COUNT = 1 to QUESTIONS
270 :
(4) 280 rem CHOOSE RANDOM NUMBERS
290 N1 = rnd(20)
300 N2 = rnd(20)
310 if N2 > N1 then swap N1,N2
320 :
330 rem DISPLAY THE QUESTION
340 print
350 if TYPE = 1 then print N1;" + ";N2;" = "; :
A = N1 + N2
360 if TYPE = 2 then print N1;" - ";N2;" = "; : A = N1 -
N2
370 input ANSWER
380 :
(5) 390 rem CHECK THE ANSWER
400 if ANSWER = A then print "CORRECT" : inc SC
410 if ANSWER < > A then print "WRONG"
420 :
430 next COUNT
440 :
(6) 450 rem DISPLAY FINAL SCORE
460 print
470 print
480 print "Your scored ";SC;" out of ";QUESTIONS
```

Lines 40-80 display the title.

Lines 100-180 request the user to enter the type of question required. Addition or subtraction can be selected by entering the appropriate number which is then assigned to variable TYPE. Line 180 checks that a valid number has been entered, and if not, sends the program

back to ask the user again.

Lines 200-230 request the user to enter the required number of questions and assign this to variable QUESTIONS.

Line 260 initiates a *for..next* loop starting at one and ending at the total number of questions represented by variable QUESTIONS. Remember that a *for..next* loop repeats a section of code a specific number of times and thus the code between lines 260 and 430 will be repeated.

Lines 290-310 choose two random numbers in the range 0-20 and assign these to the variables N1 and N2. If the value of N2 is greater than that of N1, the contents of the two variables are swapped by line 310. This ensures that the subtraction questions will be of the correct format.

Lines 350 and 360 display the question and assign the correct answer to variable A. Notice the semicolon at the end of the question. If you can remember back you will recall that this terminates the line feed and carriage return normally associated with the *print* command, and when the user enters their answer it will be displayed on the same line after the equals sign.

Line 370 waits for the user to enter an answer and assigns this to variable ANSWER.

Line 400 checks to see if the entered answer is correct. If it is correct an appropriate message is displayed and the score (variable SC) is increased by one.

Line 410 checks to see if the entered answer is incorrect and if so displays an appropriate message. No adjustment of the score is necessary.

Line 430 checks the state of the loop variable COUNT to see if all the questions have been asked. If there are still more questions to come, program execution is passed back to the line containing the *for* command (line 260). If all the questions have been displayed, the loop terminates and program execution moves on to line 440.

Lines 450-480 display the final score.

ENHANCING AND CHANGING THE PROGRAM

CHANGING THE TYPE OF QUESTION: the program, as it stands, produces addition and subtraction problems but this could be changed to cover any of the mathematical rules.

CHANGING THE RANGE OF NUMBERS: the program currently chooses random numbers for the questions in the range 0-20. The range of the *rnd* commands in lines 290 and 300 could be changed as required to make the questions easier or harder.

Chapter 9

String Variables

The last few chapters concentrated on the application of numeric variables and the various mathematic functions that can be used with them. You may recall that string variables are used to represent text or string information, and whilst the mathematical functions are not applicable, STOS offers an alternative set of commands specifically for string variables. In this chapter we shall look at the various commands available and in chapter 11 we shall put them to practical use to write another game.

Load the following:

```
10 rem STRING VARIABLES
20 rem PROGRAM = A:\STRINGS\NAME1
30 :
40 print "Enter your first name: "
50 input FIRST$
60 print "Enter your surname: "
70 input SURNAME$
80 :
90 print "Hello ";FIRST$;" ";SURNAME$. How are you"
```


Run the program and enter your name.

```
run
Enter your first name:
Joe
Enter your surname:
Bloggs
Hello Joe Bloggs. How are you
```

Lines 50 and 70 contain the familiar *input* command but look closely at the variable names that follow the command. Notice that they have a \$ symbol on the end. This indicates that the variable is a string variable rather than a numeric variable and that it represents text information rather than numerical information.

String is the name given to a series of characters within speech marks. You have already encountered strings when using the *print* command. For example:

```
print "SPACE INVADERS"
```

"SPACE INVADERS" would be referred to as a string.

String variables operate in exactly the same way as numeric variables. For example.

```
GAME$ = "PACMAN"
```

This would attach the variable name GAME\$ to one of the memory boxes and place the string "PACMAN" inside.

Looking back at the example program, the users first name is assigned to variable FIRST\$ and the users surname assigned to variable LAST\$. Line 90 contains a *print* command which incorporates variable and string information. This line is shown on the opposite page.

```
90 print "Hello ";FIRST$;" ";LAST$;". How are you ?"
```

Text and variables can be freely mixed within the *print* command provided that they are separated with semicolons. Line 90 therefore instructs the computer to:

- (1) display "Hello ".
- (2) fetch the contents of variable FIRST\$ and display them.
- (3) display " " (a space).
- (4) fetch the contents of variable LAST\$ and display them.
- (5) display ". How are you ?"

This is exactly the same as:

```
90 print "Hello ";  
91 print FIRST$;  
92 print " ";  
93 print LAST$;  
94 print ". How are you ?"
```

ADDITION AND SUBTRACTION

Whilst most of the mathematical functions are specific to numeric variables, STOS does allow the addition and subtraction of string variables.

Enter the following:

```
10 A$ = "Atari "  
20 B$ = "ST"  
30 C$ = A$ + B$  
40 print C$
```


Run the program and it will produce:

```
run
Atari ST.
```

Obviously text information cannot be added in the same sense as numeric variables but the result of the operation is to add the contents of variable B\$ to the end of variable A\$.

Load the following:

```
10 rem ASK USER FOR NAME 2
20 rem PROGRAM = A:\STRINGS\NAME2
30 :
40 print "Enter your first name"
50 input FIRST$
60 print "Enter your surname"
70 input SURNAME$
80 :
90 NAME$ = FIRST$ + " " + SURNAME$
100 print "Hello ";NAME$
```

Run the program and enter your name.

```
Enter your first name
John
Enter your surname
Smith
Hello John Smith
```

Lines 40-50 request the user to enter their first name and assign this to variable FIRST\$. Lines 60-70 request the user to enter their surname and assign this to variable SURNAME\$. Line 90 adds the first name and surname together inserting a blank space between the two. Line 100 then displays the word "Hello" followed by the name.

Notice the use of the semicolon to ensure that the name follows straight after the word "Hello".

String variables can also be subtracted.

Enter the following:

```
10 A$ = "stos basic program"
20 B$ = "s"
30 C$ = A$ - B$
40 print C$
```

Run the program.

```
run
to baic program
```

Notice that all occurrences of the letter "s" have been removed.

Now change line 80 so that variable B\$ contains the capital letter "S".

Enter the following:

```
20 B$ = "S"

list
10 A$ = "stos basic program"
20 B$ = "S"
30 C$ = A$ - B$
40 print C$
```

Run the program.

```
run
stos basic program
```


This time there is no change. This is because there are no occurrences of the capital letter "S" within variable A\$ and hence it remains unchanged.

In addition to single letters we can also subtract complete words.

As a final example change line 20 again.

Enter the following:

```
20 B$ = "basic"
```

```
list
```

```
10 A$ = "stos basic program"
```

```
20 B$ = "basic"
```

```
30 C$ = A$ - B$
```

```
40 print C$
```

Run the program.

```
run
```

```
stos program
```

This time the complete word "basic" has been removed".

FINDING THE LENGTH OF A STRING VARIABLE

The length of (the number of characters assigned to) a string variable can be found with the *len* command.

Enter the following:

```
A$ = "Hello World"
```

```
print len(A$)
```

11

The variable A\$ represents a string consisting of eleven characters and thus a value of eleven is returned. Note that spaces are included as characters.

PRODUCING STRINGS

Suppose we wanted to display a line. This could be achieved as shown below:

```
10 print "-----"
```

This is ok for a single line, but if lots of lines were required at various stages throughout the program, you would need a lot of *print* commands. An easier way would be to assign the line to a variable so that it could be printed by specifying the name of the variable. This is illustrated below:

```
10 L$ = "-----"  
20 print L$
```

Whenever a line is required the programmer simply prints variable L\$. This, you will probably agree, is a lot easier than having to print the complete line each time, but STOS simplifies the task even further.

Enter the following:

```
10 L$ = string$("-",30)  
20 print L$
```

and run the program.

The program produces exactly the same result as the previous two examples but uses the *string\$* command to create the line. The format of the *string\$* command is shown below:

```
string$ (X$,N)
```

This creates a string of N characters using the character specified by X\$. Line 10 of the program therefore creates a string of 30 x "-" to form the line.

UPPER AND LOWER CASE

The next two commands are very handy indeed. The *upper\$* and *lower\$* commands convert text information to upper (capitals) and lower case respectively.

Load the following:

```
10 rem CONVERTING CASE
20 rem PROGRAM = A:\STRINGS\CASE
30 :
40 A$ = "StOs"
50 print A$
60 print upper$(A$)
70 print lower$(A$)
```

Run the program and it will produce:

```
run
StOs
STOS
stos
```

The text has been converted .

The *upper\$* and *lower\$* commands can be very handy when analysing input from the user. Suppose we have reached a point in a program where the user needs to enter the word "YES" to continue. We need to ask the user to enter the information and then check that the information is valid. Consider the two segments of code listed below.

- (1) 10 print "Type in YES to continue"
 20 input A\$
 30 if A\$="YES" or A\$="yes" then print "I Will Continue"

- (2) 10 print "Type YES to continue"
 20 input A\$
 30 if upper\$(A\$)="YES" then print "I Will Continue"

The aim of both programs is to test the users input for the word "YES".

Example 1 requests the users input and assigns this to variable A\$. Line 30 then checks variable A\$ to see if it represents either "YES" or "yes". The check is made for both upper and lower case as the programmer may not know whether the Caps Lock key is on and hence the entered information could be in either upper or lower case. Whilst this segment of code would seem to operate correctly, it does have a serious fault. Suppose the user entered "YeS" or "yEs". Mixing upper and lower case letters in this way would cause the routine to fail unless we checked for every permutation of upper and lower case letters. Luckily this is not required as we can use the *upper\$* or *lower\$* command to overcome the problem

Example 2 uses the *upper\$* command within the *if..then* test. The contents of variable A\$ are converted to upper case before the test is made but the variable itself remains unchanged.

It is important that the programmer always check a users input for the expected and unexpected as it is quite common for a user to enter

information in an alternative format to that requested. Wrong information may be entered through a misunderstanding of the programs operation or may be entered by somebody simply wishing to see what happens!

SUBSTRINGS

There are times when the programmer may need to extract a part (or a substring) of another string. For example, suppose we had a list of peoples full names (e.g. Mr Martin Smith) and we wanted to produce a list with just the persons initial. We would have to extract the first letter of all the names to get the initials.

STOS offers three commands for extracting a substring from a main string and a fourth command for checking for the existence of one string within another. The commands for extraction are *left\$*, *mid\$* and *right\$* which allow information to be obtained from the left hand side, the middle, or the right hand side of the main string. The *instr* command is used to check for one string inside another.

Load the following:

```
10 rem SUBSTRINGS
20 rem PROGRAM = A:\STRINGS\SUBSTR
30 :
40 NAME$ = "Martin Luke Jones"
50 F$ = left$( NAME$, 6)
60 M$ = mid$( NAME$, 8, 4)
70 L$ = right$( NAME$, 5)
80 :
90 print F$
100 print M$
110 print L$
```

Run the program.

```
run
Martin
Luke
Jones
```

The program uses all three of the commands *left\$*, *mid\$* and *right\$*, so before studying the program, let us look at the format of these commands.

LEFT\$ extracts information from the left hand side of a string.

$$L\$ = \text{LEFT}\$(S\$, N)$$

where *S\$* is the string that we want to extract information from.

N is the number of characters we want to extract from the left hand side of *S\$*

MID\$ extracts information from the middle of a string

$$M\$ = \text{mid}\$(S\$, \text{START}, N)$$

where *S\$* is the string that we want to extract information from.

START is a number that specifies how far in to the string we should start extracting from.

N is the number of characters we want to extract.

RIGHT\$ extracts information from the right hand side of a string.

$$R\$ = \text{right}\$(S\$, N)$$

where `S$` is the string that we want to extract information from.

`N` is the number of characters that we want to extract from the right hand side of `S$`.

So let us look back at the program. Line 50 extracts 6 characters from the left hand side of the variable `NAME$` and assigns this new string to variable `F$`. Line 60 extracts 4 characters from the variable `NAME$` starting at character position 8 and assigns this new string to variable `M$`. Line 70 extracts 5 characters from the right hand side of variable `NAME$` and assigns this new string to variable `R$`. The program therefore extracts the first, middle and last names from the main variable `NAME$`.

The last command related to substrings is *instr* (in string) and this checks for the existence of one string inside another string.

Load the following:

```
10 rem THE INSTR COMMAND
20 rem PROGRAM = A:\STRINGS\INSTR
30 :
40 X$ = "cat dog mouse horse budgie fish"
50 :
60 print "Enter an animal"
70 input A$
80 A$ = lower$(A$)
90 :
100 I = instr (X$,A$)
110 print I
```

Run the program and enter the word horse.

```
run
Enter an animal
```

```
horse
15
```

Now run the program again and enter the word pig.

```
run
Enter an animal
pig
0
```

In the first example the variable X\$ is searched for the first occurrence of the string "horse". If the string is found, the position of the substring is assigned to variable I. In this case the substring "horse" starts at position 15 within the variable X\$. If the substring does not exist within the main string a value of zero is returned and this is illustrated in the second example when the word "pig" is entered.

Strings can be expressed as text or as a variable. For example:

```
10 S=instr("Mr Peter Jones","Peter")
```

and:

```
10 S$="Peter"
20 N$="Mr Peter Jones"
30 S=instr(N$,S$)
```

are both acceptable.

Instr can also be used within a print command.

Enter the following:

```
print instr("Programming","r")  
2
```

This produces the number 2 because the first occurrence of the letter "r" was found at position 2. Suppose now that we wanted to continue looking for other occurrences of the letter "r". STOS allows us to modify the *instr* command to specify the starting position for the search.

Enter the following:

```
print instr("Programming","r",3)  
5
```

This time the program searches for the letter "r" but starts the search at character position 3. The first "r" is therefore ignored and character position 5 is returned.

Chapter 10

User Input

We have already seen how the *input* command is used to assign information provided by the user to a variable and how this information can then be used within our program by specifying the variable name. In addition to the *input* command there are a number of other methods that can be used to derive user input. In this chapter we shall take a look at these methods and the ways in which they can be utilised within our programs.

The Atari ST computer offers three main sources of input - the keyboard, mouse and joystick.

Before the introduction of powerful computers like the ST, the keyboard was the sole source of input as mice and joysticks were not available. The keyboard can therefore be used for almost any application but there are many areas where it is more convenient and advantageous to use the mouse or a joystick.

Suppose that we were writing an art program that let the user draw pictures on the screen. We could control the pen using the cursor keys on the keyboard but this would be far from ideal as it would limit

movement in the directions left, right, up and down. The mouse, on the other hand, would be ideal as it would offer no restriction on the pens movement.

Suppose also that we were writing a game like Space Invaders where a gun is moved from left to right across the bottom of the screen and a fire button is used to blast the baddies out of the sky. The keyboard could be used, incorporating the left/right cursor keys for movement and the F key for fire. Whilst this would work ok, it would be far easier to use a joystick which is specifically designed for just this purpose. This being the case, it must be remembered that not all users may have a joystick and hence the option of joystick or keyboard operation could be provided.

When writing a program we need to think carefully about the input that we require and the easiest way for the user to actually provide this. It is important that we clearly specify the input format required from the user. The easiest way to learn about input is to look at each of the methods in turn so let's start with the keyboard.

THE KEYBOARD

The input command

In chapter 4 we saw a program that calculated the area of a rectangle and this is reproduced below.

$$\text{AREA} = \text{LENGTH} * \text{WIDTH}$$

WIDTH

LENGTH

```
10 rem CALCULATE THE AREA OF A RECTANGLE
20 rem PROGRAM: A:\VARIABLE\RECTANG
30 :
40 print "Enter the length"
50 input L#
60 print "Enter the width"
70 input W#
80 :
80 AREA# = L# * W#
90 print "The area of the rectangle is ";AREA#
```

When the program is run it produces:

```
Enter the length
? 5
Enter the width
? 4
The area of the rectangle is 20
```

In this, and all the other programs covered so far, we have used a *print* command followed by an *input* command to request information from the user. For example:

```
40 print "Enter the length"
50 input L#
```

Whilst this is perfectly acceptable, STOS allows the message to be displayed as part of the *input* command and hence a separate *print* command is not required. The above two lines can therefore be converted in to just one line as shown below:

```
40 input "Enter the length ";L#
```

The message is placed in speech marks directly after the *input* command and the variable is placed at the end of the line separated by

a semicolon.

The original area calculating program now becomes:

```
10 input "Enter the length ";L#  
20 input "Enter the width ";W#  
30 AREA# = L# * W#  
40 print "The area is ";AREA#
```

This program would produce:

```
Enter the Length 5  
Enter the Width 4  
The Area of the Rectangle is 20
```

Looking at the *input* commands on lines 10 and 20 you can see how the message is placed in speech marks and the variable is separated from this with a semicolon. Look carefully at the output from the two programs. The *input* commands in the first example displayed question marks but these have been omitted in the second program. When used on its own, the *input* command displays a question mark to indicate to the user that the computer is waiting for them to do something. When a message is included, STOS assumes that the message will prompt the user for the required information and hence there is no need to display the question mark. If you want to display a question mark then it should be included as part of the message. For example:

```
100 input "Would you like to play again ?" ;A$
```

The last program employs separate *input* commands to acquire the length and width of the rectangle but these can be combined in to one command as shown below:

```
10 input "Enter the Length, Width"; LENGTH, WIDTH  
20 AREA = LENGTH * WIDTH
```

```
30 print "Then area of the Rectangle is ";AREA
```

STOS allows a single *input* command to request multiple pieces of information and assigns these to different variables. Each of the variables are separated by a comma as shown in line 20. The information can be entered in one of two ways as the following examples illustrate.

(1) User enters data separated by comma.

```
Enter the Length, Width 5,4  
The Area of the Rectangle is 20
```

The user enters the two pieces of data separated by a comma and then presses the Return key. The information is assigned to the appropriate variables and the program continues.

(2) User presses the Return key after each entry.

```
Enter the Length, Width 25  
?? 12  
The Area of the Rectangle is 300
```

The user enters the length and presses the return key. STOS then displays two question marks to make the user aware that more information is required. The user enters the width and the program continues.

When using the *input* command in this form, it is important that the user is given specific instructions regarding the way in which the information should be entered. Although the required information is shown separated by a comma, this may still prove confusing to the user and hence another *print* command at say line 5 could actually state that the information must be separated by a comma.

It is probably better to use separate *input* commands so as to make it as easy as possible for the user.

The line input command

So far we have only used *input* with numeric variables but it can also be used in the same fashion with string variables. Suppose that we want to input the users address. We could either input and assign each line of the address to a separate variable or we could input and assign the whole address to one variable.

```
10 input "Enter address - line 1"; ADDR1$  
20 input "Enter address - line 2"; ADDR2$  
30 input "Enter address - line 3"; ADDR3$  
40 input "Enter address - line 4"; ADDR4$
```

The above program would request each line of the address and assign it to the variables. Let us now consider a program that assigns the whole address to a single variable.

Enter the following:

```
10 input "Enter address: ";A$
```

Run the program and enter the address as shown:

```
run
```

```
Enter address: 1 The Street, A Town, City, AB12 3CD
```

It is common practice to separate each line of an address with a comma as shown above. To check that the address has actually been assigned to the variable we shall print the variable.

Enter the following:

```
print A$  
1 The Street
```

The variable A\$ only contains the first part of the address. We have already seen that the *input* command uses a comma to separate each element of information, and when the comma is encountered, the computer thinks that the next piece of information should be assigned to the next variable. In this example there is only one variable specified and hence the rest of the information is lost. The computer therefore gets confused when we enter a comma as part of the text. To overcome this we have to use a variation of the *input* command, *line input*. Amend the original program to include this new command.

Enter the following:

```
10 line input "Enter address: ";A$
```

Run the program and enter the address:

```
run  
Enter address: 1 The Street, A Town, City, AB12 3CD
```

and now print the variable A\$:

```
print A$  
1 The Street, A Town, City, AB12 3CD
```

This time all of the information including commas has been assigned to the variable. Therefore, if you think that the user may enter some information containing commas, you must use the *line input* command rather than just *input*.

The input\$ command

Suppose we were writing a security program where the user had to enter a 4 digit password before they were allowed access to the system. We need to ask the user for the password, check to see if it is valid and then either allow or deny access accordingly. The following program would perform this operation:

```
10 rem PASSWORD PROTECTION
20 :
30 rem SET THE PASSWORD
40 ACCESS = 9345
50 :
60 rem ASK USER FOR PASSWORD
70 print "Please enter 4 digit password"
80 input P
90 :
100 rem CHECK THE PASSWORD
110 if P = ACCESS then print "ACCESS ALLOWED" else
print "ACCESS DENIED"
```

We have already seen that a program must be as user friendly as possible. The above program is fine but there are two areas that could be improved. First we ask the user to enter a 4 digit password but the *input* command does not allow us to limit the number of entered characters and will accept more than four characters. The user could, even though requested otherwise, enter a 10 digit password. The other thing to consider is that somebody else overlooking the computer would be able to see the password as it was typed in. The input could be hidden by setting the pen colour the same as the paper colour but there is a quicker and easier way to improve both the areas mentioned.

Load the following:

```
10 rem PASSWORD PROTECTION
20 rem PROGRAM = A:\INPUT\PASSWORD
30 :
40 key off: mode 0
50 :
60 rem SET THE PASSWORD
70 ACCESS = 9345
80 :
90 rem ASK USER FOR PASSWORD
100 print "Please enter 4 digit password"
110 P$ = input$(4)
120 :
130 rem CHECK THE PASSWORD
140 P = val(PASS$)
150 if P = ACCESS then print "ACCESS ALLOWED"
else print "ACCESS DENIED"
```

Run the program and enter the password 9345:

```
run
Please enter 4 digit password      [Enter 9345]
ACCESS ALLOWED
```

Now run the program again and enter the wrong password:

```
run
Please enter 4 digit password      [Enter 1234]
ACCESS DENIED
```

The program only allows access when the correct password is entered.

This time the program only allows 4 characters to be entered and does not display them on the screen.

Look carefully at line 110. The *input* command has been changed for the *input\$* command. This takes the following form:

X\$ = input\$(N)

where N specifies the number of characters required from the user.

Line 110 therefore accepts 4 characters from the user and assigns these to variable P\$. Note that the user does not have to press the return key as the program automatically moves on after the correct number of characters have been received. The *input\$* command can only be used with string variables and so line 140 converts the entered information into a numeric value using the *val* command. This value is then assigned to variable P and the program continues as before.

The *input\$* command, unlike *input*, does not display the entered text on the screen and is therefore suited to security applications and areas where only one or two characters are required. It is often used to obtain a simple Yes or No answer as shown below:

```
10 print "Would you like to play again (Y/N)"
20 PLAY$ = input$(1)
30 if upper$(PLAY$) = "N" then end
```

The user enters Y for YES or N for NO and the program acts accordingly.

The inkey\$ command

Another command very similar to *input\$* is *inkey\$*.

Load the following:

10 rem THE INKEY\$ COMMAND

```
20 rem PROGRAM = A:\INPUT\INKEY
30
40 clear key
50 :
60 while I$ = ""
70 I$ = inkey$
80 wend
90 :
100 print "Character = ";I$
```

Run the program and enter the letter "h".

```
run
h
Character = h
Ok
```

The above program waits for a key to be pressed and then displays the character on the screen. Line 70 contains the *inkey\$* command which checks to see if a key has been pressed. If a key has been pressed, the character is assigned to the variable *I\$*. *Inkey\$*, unlike *input* and *input\$*, does not stop and wait for the user to enter information. It makes a quick check of the keyboard and moves on, hence the reason for placing it within a loop which keeps repeating until a key is pressed.

The *inkey\$* command works by reading a keyboard buffer which is maintained by the computer. This is a small area of memory that stores each key press made by the user. When the *inkey\$* command is called, it looks at the buffer to see if a new character has arrived as the result of a key being pressed. The *clear key* command in line 40 clears the keyboard buffer of any old, unwanted information before the *inkey\$* command starts checking it.

The *inkey\$* command offers two advantages over the *input\$* command.

First it allows us to make checks for key presses without actually stopping the programs execution, and secondly it allows us to check for all the keys on the keyboard rather than just the alpha numeric keys. The *input\$* command can only accept characters that have an ascii code. These are the printable characters, letters, numbers, symbols, etc., and an alternative method is required to detect the other keys.

Run the program again and press the HELP key.

```
run
[ HELP ]
Character =
```

This time the key press is accepted but nothing is printed. The HELP key does not have an ascii code and hence cannot be printed. The *inkey\$* command detected the key press but cannot identify the actual key. To identify the key we have to consider the *scancode*.

SCANCODES

All of the keys on the keyboard are represented by a scancode. This is a code that the computer uses to identify which key has been pressed. When dealing with alpha-numerical information we do not need to worry about scancodes as the *input* and *input\$* commands can be used to input the information direct. The various other keys on the keyboard are not represented by ascii codes and hence the scancode must be analyzed to determine which key has been pressed.

Load the following:

```
10 rem SCANCODES
20 rem PROGRAM = A:\INPUT\SCANCODE
30 :
```

```
40 while I$ = ""
50 I$ = inkey$
60 wend
70 :
80 S = scancode
90 print "Scancode = ";S
```

Run the program and press the HELP key:

```
run
[ HELP ]
Scancode = 98
```

now run the program again and press the INSERT key:

```
run
[ INSERT ]
Scancode = 82
```

As you can see, each key has a different scancode and hence the key can be identified.

The program is similar to the last example except we have now followed the *inkey\$* command with a *scancode* command. Line 80 assigns the scancode of the pressed key to variable S and line 90 prints this.

The scancodes for various keys are listed below.

<u>KEY</u>	<u>SCANCODE</u>
ESC	1
TAB	15
BACKSPACE	14
DELETE	83

HELP	98
UNDO	97
INSERT	82
CLR/HOME	71
UP CURSOR	72
DOWN CURSOR	80
LEFT CURSOR	75
RIGHT CURSOR	77
F1 - F10	59 - 68
F11 - F20	84 - 93

The following program illustrates the way in which the key presses are detected within a practical application.

Load the following:

```
10 rem DETECTING KEYS USING THE SCANCODE
20 rem PROGRAM = A:\INPUT\KEYSCAN
30 :
40 while X$ = ""
50 X$ = inkey$
60 wend
70 :
80 A = asc(X$)
90 S = scancode
100 :
110 if A=0 and S=72 then print "UP ARROW"
120 if A=0 and S=80 then print "DOWN ARROW"
130 if A=0 and S=75 then print "LEFT ARROW"
140 if A=0 and S=77 then print "RIGHT ARROW"
150 if A=0 and S=98 then print "HELP"
160 if A=0 and S=97 then print "UNDO"
170 if A=0 and S=59 then print "F1"
```

Run the program a few times and press the arrow keys, help key,

undo key, etc.

The first part of the program is identical to that of the previous program. The *inkey\$* command is placed within a loop that keeps repeating until a key press is detected. When a key is pressed, the character will be assigned to variable X\$. If the pressed key does not have an ascii code then chr\$(0) is assigned to variable X\$. This indicates that a key has been pressed but it cannot be identified as no ascii code has been returned. In this instance we have to check the scancode to identify the key. Line 80 assigns the ascii code of the pressed key to variable A and line 90 assigns the scancode to variable S. Lines 110-170 check the scan codes and display a message indicating the appropriate key.

Line 170 of the program checks to see if the function key F1 has been pressed. The function keys are often used to allow the user to select various options, and with this in mind, the creators of STOS provided another handy function for checking the status of these keys.

THE FUNCTION KEYS

Selecting options using the function keys

If you look at the keyboard you will see that the function keys are numbered F1 - F10. These keys also have a second function denoted F11 - F20 which is accessed by holding down the shift key at the same time as pressing the function key. For example F12 is obtained by pressed SHIFT+F2.

The function keys allow options to be selected or information to be entered using a single key stroke. We have already seen how these keys can be detected using the scancode, but if a program requires the function keys only, then it is easier to use the *fkey* function.

Fkey checks the state of the function keys and returns a value in the range 1-20 if a key has been pressed or 0 (zero) if no keys are pressed. Like the *inkey\$* command, *fkey* checks the keyboard buffer and does not halt program execution.

Load the following:

```
10 rem THE FUNCTION KEYS
20 rem PROGRAM = A:\INPUT\FKEYS
30 :
40 while F = 0
50 F = fkey
60 wend
70 :
80 print "You pressed function key ";F
```

Run the program and press one of the function keys. The program displays a number indicating which one of the function keys was pressed.

Lines 40-60 form a loop which continues all the time that *fkey* is equal to zero. When a function key is pressed, the value of *fkey* changes to indicate the pressed key and the loop ends. Line 80 then prints the number of the key that was pressed.

Let us see how this could be used in a practical application. Consider the following:

```
10 print "F1=LOAD F2=SAVE F3=QUIT"
20 :
30 while F=0
40 F=fkey
50 wend
60 :
70 on F goto .....
```

The user must select one of the three options by pressing one of the function keys. The loop formed by lines 30 and 50 continually repeats until a function key is pressed. Line 70 then uses an *on..goto* command to send program execution to the appropriate part of the program to service the selected task.

Using the function keys to enter strings

The last example used the function keys to select an option but they can also be used to store text information (strings). The current contents of all the function keys can be listed using the *keylist* command.

Enter the following:

keylist

STOS initially assigns default information to the keys but this can be changed by the programmer. These 'programmable keys' are very useful when entering a program and can also be used as a method of input within a program.

Information is assigned to the keys using the *key()* command.

Enter the following:

```
key(8) = "Function keys are very useful"  
key(9) = "run"  
key(10) = "list"
```

Press the function key F8 and the message will be displayed. Function keys F9 and F10 have been assigned more useful information which will aid the programmer when entering a program. Whenever a program listing is required, the programmer simply presses F10

instead of having to type in the word "list" each time. You could assign any commands to the function keys. For example, you may like to make F4 renumber a program as shown below:

```
key(4) = "renum"
```

Function key F1 always maintains a copy of the last entered command and thus keeps changing as you enter a program.

You have already seen how the function keys can be detected using the *fkey* command but they could also be used to enter information.

Load the following:

```
10 rem THE FUNCTION KEYS
20 rem PROGRAM = A:\INPUT\FKEYS2
30 :
40 rem ASSIGN FUNCTION KEYS
50 key(1) = "Load a File"
60 key(2) = "Save a File"
70 key(3) = "End the Program"
80 :
90 print "Enter option (F1 = Load F2 = Save F3 = End)"
100 input OPT$
110 :
120 rem CHECK USER INPUT
130 if OPT$ = "Load a File" then print "Load a File"
140 if OPT$ = "Save a File" then print "Save a File"
150 if opt$ = "End the Program" then print "End the
Program"
160 goto 90
```

Run the program and select the options using the F1, F2 and F3 keys.

Although the *fkey* command could be used to detect the actual key

presses direct, the above program illustrates the way in which the function keys can be used to represent string information.

That just about covers the keyboard. As you have seen, there are lots of commands available to the programmer and those used will depend upon the type of input required. Sometimes the program may be better to use the mouse so let's take a look at this.

THE MOUSE

The mouse consists of two main input elements. A roller underneath the mouse detects movement up, down, left and right, and two buttons on the top offer switched input. This combination of movement and buttons produces a very useful and powerful input tool, and STOS provides commands that allows us to analyze and act on this input.

Normally the type of program dictates the methods of input required. There is no point using the mouse just for the sake of it, but it is true to say that most programs can benefit in one form or another from using the mouse. The first two commands offered by STOS simply allow the mouse pointer to be activated or deactivated.

Move the mouse around and you should see the mouse pointer follow your movements. Now enter the following:

hide on

The mouse pointer should have disappeared from the screen. When writing a program that does not require the mouse it is best to hide the mouse pointer at the start of the program. This will ensure that the mouse pointer does not obscure other information that may be displayed on the screen.

The mouse pointer can be restored at any time using the *show on*

command as shown below:

show on

Move the mouse around and you will see that the mouse pointer is visible again.

Take a close look at the mouse pointer. As the name suggests, it is in the shape of a pointing arrow. The appearance of the mouse pointer can be changed to one of three standard shapes or to a custom shape defined by the programmer. The three standard shapes are listed below:

- 1 Arrow (this is the default)
- 2 A Pointing Hand
- 3 A Clock Face

The mouse pointer is changed using the *change mouse* command and this takes immediate effect.

Enter the following:

change mouse 2

and the mouse pointer immediately changes to a pointing hand.

Now enter:

change mouse 3

and the mouse pointer changes to a clock face.

Finally, change it back to an arrow again:

change mouse 1

When the mouse pointer is hidden it can still be moved around the screen and changed. We could therefore hide the mouse pointer, change the style and then show it in the new form.

Load the following:

```
10 rem CHANGE THE MOUSE POINTER
20 rem PROGRAM = A:\INPUT\MOUSE1
30 :
40 change mouse 1
50 wait key
60 hide on
70 change mouse 2
80 wait key
90 show on
```

Run the program and press any key.

The mouse pointer is changed to an arrow, and when a key is pressed, is hidden from view. Whilst hidden, the mouse pointer is changed to a pointing hand, and when another key is pressed, the mouse pointer re-appears in its new form.

Well that is enough about the pointer. Let us now take a look at movement.

Physical movements of the mouse are relayed to the screen and STOS offers commands to locate the position of the mouse pointer. This is very useful as it enables us to detect the actual movements made by the user. We could for example ask the user to select an item from an option list using the mouse. The mouse pointer position would be monitored by the programmer to ascertain the option selected.

Load the following:

```
10 rem MOUSE MOVEMENTS
20 rem PROGRAM = A:\INPUT\MOUSE2
30 :
40 rem SET SCREEN RESOLUTION
50 key off: mode 0
60 :
70 rem DISPLAY COORDINATES
80 locate 1,1
90 print string$(" ",20)
100 locate 1,1
110 print x mouse, y mouse
120 wait vbl
130 :
140 goto 80
```

Run the program and move the mouse pointer around. The mouse pointer coordinates are displayed at the top of the screen and are updated as the mouse is moved around.

The program uses two new functions *x mouse* and *y mouse*. These return the current X and Y coordinates of the mouse pointer. If you run the program you will notice that the x coordinates are in the range 0-319 and y coordinates are in the range 0-199 which is the range for low resolution as set in line 50. Try changing line 50 as shown below:

```
50 key off: mode 1
```

This sets medium resolution, and if you run the program again, you will notice that the X coordinates now extend to 639.

x mouse and *y mouse* are reserved variables that maintain the current position of the mouse pointer.

Whilst looking at new functions, line 120 contains the *wait vbl* (wait for vertical blank) command. Televisions and monitors display information using an electron beam that moves down the screen. If we try and update or write to the screen when the electron beam is moving, it can cause a jerky or un-synchronised image. The *wait vbl* command therefore halts program execution until the electron beam has completed the current screen. The vertical blank is the period in which the electron beam is switched off and moved from the bottom of the screen back to the top, and during this period no information is actually written to the screen. The command is normally associated with sprites but in this example stops the screen from flickering when the mouse coordinates are printed to the screen.

So far the mouse pointer has been free to move to any position on the screen. In our usual bid to make programs as user friendly as possible it would be nice to limit movement of the mouse to an area directly applicable to the operation in hand. STOS allows the movement of the mouse pointer to be limited using the *limit mouse* command. The *limit mouse* command allows a rectangular area to be specified and inhibits movement of the mouse pointer outside this boundary.

The format of the *limit mouse* command is shown below:

limit mouse X,Y to X1,Y1

where X and Y indicate the graphic coordinates of the top left corner of the rectangle and X1 and Y1 indicate the coordinates of the point diagonally opposite.

Enter the following:

limit mouse 100,50 to 300,100

Move the mouse around and you will notice that movement is limited. To restore movement of the mouse pointer enter the following:

limit mouse

You can now move the mouse pointer around the whole screen again.

As can be seen, the mouse movement is restored using the *limit mouse* command with no parameters.

This leaves one more area of the mouse to cover. On top of the mouse are two buttons. You have probably used the left mouse button already as this is required with the GEM desktop to select the various functions, but within STOS the programmer can make use of both the buttons.

Load the following:

```
10 rem MOUSE BUTTONS
20 rem PROGRAM = A:\INPUT\MOUSE3
30 :
40 key off : mode 0
50 :
60 print "Press the mouse buttons"
70 while M=0
80 M=mouse key
90 wend
100 :
110 if M=1 then print "LEFT BUTTON"
120 if M=2 then print "RIGHT BUTTON"
```

The *mouse key* command is used to analyze the mouse buttons. The status of the buttons is returned as a value as shown below:

<u>Returned Value</u>	<u>Status of Mouse Buttons</u>
0	No buttons pressed
1	Left button pressed

2	Right button pressed
3	Both buttons pressed

Line 80 of the program assigns the value returned by *mouse key* to the variable M. Note that you do not have to assign the value to a variable as *mouse key* can be used directly. Line 110 of the program could be changed, for example, to

```
110 if mouse key = 1 then print "Left mouse key"
```

We mentioned a moment ago that we can produce customised mouse pointers. The following program does much the same as the last program but uses a custom mouse pointer to display which mouse button is pressed.

Load the following:

```
10 rem CUSTOM MOUSE POINTER
20 rem PROGRAM = A:\INPUT\MOUSE4
30 :
40 key off: mode 0: flash off
50 palette $235, $773, $777, $555, $444, $730,
$700, $0
60 :
70 rem MONITOR MOUSE BUTTONS
80 change mouse 4
90 m=mouse key
100 if m=1 then change mouse 5: while m=1:wend
110 if m=2 then change mouse 6: while m=2:wend
120 if m=3 then change mouse 7: while m=3:wend
130 :
140 goto 80
```

Run the program and press the left mouse button, the right mouse button and then both the buttons. As you can see, your key presses

are echoed by the mouse pointer on the screen. This program makes use of the sprite facilities of STOS and is shown to illustrate what can be achieved with the mouse pointer. We shall look at sprites later so do not worry too much about the programs operation at this stage.

JOYSTICKS

Where would we be without the good old joystick. Mega zapping, alien blasting shoot-em-ups just would not be the same if limited to keyboard input.

We have already seen that very careful attention needs to be paid to the chosen method of user input. Whilst the joystick is the obvious input medium for many games, consideration should also be given to those without a joystick and hence an alternative input method should also be offered.

The joystick provides five detectable functions. The stick itself can be moved up, down, left and right and the fire button or trigger can be pressed or pulled.

Load the following:

```
10 rem JOYSTICK MOVEMENT
20 rem PROGRAM = A:\INPUT\JOY
30 :
40 rem MONITOR JOYSTICK
50 if jup then print "UP"
60 if jdown then print "DOWN"
70 if jleft then print "LEFT"
80 if jright then print "RIGHT"
90 if fire then print "FIRE"
100 goto 40
```

Run the program and move the joystick around.

The program monitors the joystick and prints a message describing each movement that is detected. Five functions are used to detect the status as listed below:

<u>Command</u>	<u>Status of Joystick</u>
jup	Joystick moved upwards
jdown	Joystick moved downwards
jleft	Joystick moved left
jright	Joystick moved right
fire	Fire button pressed

Each function can produce one of two results. Consider *jup*, if the joystick has been moved up then *jup* will be true and hence equal the value -1. If the joystick has not been moved then *jup* will equal 0. The test could therefore be made in a number of ways as shown below:

```
10 if jup = -1 then print "UP"
```

```
10 if jup = true then print "UP"
```

```
10 if jup < > 0 then print "UP"
```

```
10 if jup then print "UP"
```

[*true* is a reserved variable that represents the value -1]

These serve simply to indicate the different ways in which these functions can be used but the last example would be the favoured form.

STOS offers another function which allows all of these tests to be incorporated into one.

Enter the following:

```
10 rem DETECT JOYSTICK MOVEMENT
20 J = joy
30 print J
40 goto 20
```

Run the program and move the joystick around.

Line 20 introduces the *joy* function. The *joy* function reads the status of the joystick and provides the result as a binary number. If you run the program and operate the joystick you will find that different numbers are displayed on the screen. Listed below are the numbers that are returned depending on the current status of the joystick.

<u>VALUE</u>	<u>JOYSTICK STATUS</u>
0	No joystick operation
1	Joystick moved up
2	Joystick moved down
4	Joystick moved left
8	Joystick moved right
16	Fire button pressed
17	Joystick moved up and fire button pressed
18	Joystick moved down and fire button pressed
20	Joystick moved left and fire button pressed
24	Joystick moved right and fire button pressed

When using the joystick we can therefore use the *joy* function or the *jup*, *jdown*, *jleft*, *jright* and *fire* functions. For example, suppose that you wanted to detect a press of the fire button. Either of the following two programs would carry out this operation.

(1) 10 if joy = 16 then print "FIRE"

(2) 10 if fire then print "FIRE"

In a later chapter we shall actually write some simple games that use the joystick facilities described in this chapter.

READ AND DATA

The keyboard, mouse and joystick have one thing in common - they all require external operation to signal movement or enter data. There are times when the data is required within the program without having to be entered by the user. We have already seen how data can be assigned to variables but there is an alternative method that allows the data to be stored as a list and read in to variables as and when required. The following example illustrates this.

Load the following:

```
10 rem READ AND DATA COMMANDS
20 rem PROGRAM = A:\INPUT\READ
30 :
40 rem READ AND DISPLAY INFORMATION
50 read A$
60 print "A$ contains ";A$
70 read B
80 print "B contains ";B
90 read C$
100 print "C$ contains ";C$
110 :
120 rem DATA
130 data "ALPHA", 54, "BRAVO"
```

Run the program and it will produce the following:

run

A\$ contains ALPHA
B contains 54
C\$ contains BRAVO

Line 50 contains the *read* command which reads the first item of data from the data list at line 130. The data is assigned to the variable specified with the *read* command - in this case variable A\$. Line 60 then prints the variable.

Line 70 reads the next item of data from the list and assigns this to variable B, line 90 reads the next line of data and assigns it to variable C\$.

The *data* command is used to specify a list of information that can then be assigned to variables using the *read* command. Each time a *read* command is encountered the program extracts the next item from the list of data and assigns this to the specified variable. Notice that string data is placed within speech marks and that string and numeric data can be freely mixed. It is important that you read string data in to string variables and numeric data in to numeric variables. If you try and assign the wrong sort of data to a variable an error will be reported. An error will also be reported if you try and read more items than that contained within the line of data.

The *read* and *data* commands are most useful when used with variable arrays as they allow long lists of information to be assigned to the variables using a minimum of program code. For example, consider a program that assigns the names of ten people and lists them on the screen. This could be achieved as shown below:

```
10 rem DIMENSION VARIABLE ARRAY
20 dim N$(10)
30 :
40 rem ASSIGN DATA TO VARIABLES
50 N$(1) = "Mark"
```

```
60 N$(2) = "Jill"
70 N$(3) = "Susan"
80 N$(4) = "Luke"
90 N$(5) = "Zoe"
100 N$(6) = "Paul"
110 N$(7) = "Mike"
120 N$(8) = "Rebecca"
130 N$(9) = "Mary"
140 N$(10) = "Roger"
150 :
160 rem PRINT THE LIST
170 for A = 1 to 10
180 print N$(A)
190 next A
```

The same result could be achieved as shown below.

Load the following:

```
10 rem READ AND DATA COMMANDS
20 rem PROGRAM = A:\INPUT\LIST
30 :
40 rem DIMENSION VARIABLE ARRAY
50 dim N$(10)
60 :
70 rem ASSIGN DATA TO VARIABLES AND PRINT
80 for A = 1 to 10
90 read N$(A)
100 print N$(A)
110 next A
120 :
130 rem DATA
140 data "Mark", "Jill", "Susan", "Luke", "Zoe",
"Paul", "Mike", "Rebecca", "Mary", "Roger"
```


Run the program and the names will be displayed.

In the above examples we only used ten names but imagine how much time would be saved for a list of one hundred or even one thousand names.

That covers the input commands and we shall now write another game.

Chapter 11

Guess the Word Game

In this chapter we shall develop another game using the string operations discussed in chapter 9. We shall define the program operation and follow the complete design process through to producing the code and playing the game.

DEFINITION

We shall write a program that is similar to hangman. The computer will choose a word at random and the user must try and guess the word by entering one letter at a time. Each time the user enters a letter, the computer will check to see if the letter has already been used or whether it is contained within the hidden word. When all the letters have been correctly guessed, the computer will display the complete word. The program shall offer a set of fifty words.

Load the following:

```
10 rem GUESS THE WORD GAME
20 rem PROGRAM = A:\WORDGAME\WORD
30 :
40 rem SET SCREEN RESOLUTION
50 key off : mode 0
60 :
70 rem INITIALISE VARIABLES
80 dim W$(49),SOFAR$(20)
90 FOUND=0
100 :
110 rem READ THE WORD DATA
120 for A=0 to 49
130 read W$(A)
140 W$(A)=upper$(W$(A))
150 next A
160 :
170 rem CHOOSE A RANDOM WORD
180 R=rnd(49)
190 W$=W$(R)
200 L=len(W$)
210 :
220 rem DEFINE THE WORD
230 for A=1 to L
240 SOFAR$(A)="_"
250 next A
260 USED$=""
270 :
280 repeat
290 :
300 print "Word so far"
310 for A=1 to L
320 print SOFAR$(A);
330 next A
```

```
340 print
350 print "Letters used so far"
360 print USED$
370 print
380 print "Guess a letter"
390 GUESS$ = input$(1)
400 GUESS$ = upper$(GUESS$)
410 :
420 rem HAS INPUT LETTER ALREADY BEEN USED
430 if instr(USED$,GUESS$) then print "LETTER
ALREADY USED" : print "Press any key to continue" :
wait key : goto 530
440 :
450 rem ADD LETTER TO LETTERS USED LIST
460 USED$ = USED$ + GUESS$
470 :
480 rem IS THE LETTER IN THE WORD
490 for A = 1 to L
500 if GUESS$ = mid$(W$,A,1) then
SOFAR$(A) = GUESS$ : inc FOUND
510 next A
520 :
530 rem HAVE ALL THE LETTERS BEEN FOUND
540 cls
550 until FOUND = L
560 :
570 rem THE WORD HAS BEEN FOUND
580 print "THE WORD WAS ";W$
590 print "Well done you have found the word"
600 end
610 :
620 :
630 :
640 rem WORD DATA
650 data "shirt","hat","gloves","coat","shoes"
```



```
660 data "cup","saucer","plate","spoon","fork"
670 data "dog","cat","mouse","hamster","bird"
680 data "computer", "disk", "program", "list",
"memory"
690 data "mother", "father", "son", "daughter",
"granddad"
700 data "grass","flower","shed","field","swing"
710 data "television", "radio", "stereo", "satellite",
"speaker"
720 data "car","lorry","bike","bus","train"
730 data "apple","pear","orange","banana","peach"
740 data "toy","teddy","doll","puzzle","book"
```

Play the game a few times so that you can see exactly what it does.

We shall now look at the construction and operation of each section of the program.

DESIGN

- (1) Set screen resolution and initialise variables.
- (2) Read word data into an array.
- (3) Choose a word at random.
- (4) Define the word.
- (5) Ask user for guess and perform checks.
- (6) End the game.
- (7) Word data.

(1) SET SCREEN RESOLUTION AND INITIALISE VARIABLES

```
10 rem GUESS THE WORD GAME
20 rem PROGRAM = A:\WORDGAME\WORD
30 :
```

```
40 rem SET SCREEN RESOLUTION
50 key off : mode 0
60 :
70 rem INITIALISE VARIABLES
80 dim W$(50),SOFAR$(20)
90 FOUND=0
```

Lines 10 and 20 contain remarks indicating the nature/title of the program and its location on disk. Line 50 switches off the function key window and sets the screen resolution to low. Line 80 dimensions two variable arrays W\$() and SOFAR\$(). At the start of the program the fifty words will be read from data statements in to array W\$ and one of these will then be chosen at random. Variable array SOFAR\$() maintains each letter of the word and is dimensioned to twenty which allows a maximum word length of twenty letters. Do not worry too much about SOFAR\$() yet as all will be revealed in a moment. Line 90 clears variable FOUND which is used to maintain the number of letters found so far.

(2) READ WORD DATA IN TO ARRAY

```
110 rem READ THE WORD DATA
120 for A=0 to 49
130 read W$(A)
140 W$(A)=upper$(W$(A))
150 next A
```

The words are now read in to the array W\$(). Line 120 initiates a *for..next* loop in the range 0 - 49, line 130 reads the word data in to the array W\$(), line 140 converts the word to upper case and line 150 continues the loop. Notice the way in which the loop variable A is used in line 130 to indicate each element of the array W\$(). Notice also that the *for..next* loop has been assigned the range 0-49 rather than 1-50. The reason for this will become apparent during the next

stage of the program.

(3) CHOOSE A WORD AT RANDOM

```
170 rem CHOOSE A RANDOM WORD
180 R=rnd(49)
190 W$=W$(R)
200 L=len(W$)
```

The next stage is to choose one of the words that are now stored in variable array W\$(). The easiest way to do this is to choose a random number and use this to select the word. Line 180 selects a random number in the range 0 - 49 and assigns this to variable R. Line 190 uses the random number to assign the word from variable array W\$() to the variable W\$, and thus variable W\$ now contains the word that will be used for the game. Line 200 finds the length of the word using the *len* function and assigns this to variable L.

(4) DEFINE THE WORD

```
220 rem DEFINE THE WORD
230 for A=1 to L
240 SOFAR$(A)="_"
250 next A
260 USED$=""
```

The next operation is to set a variable array SOFAR\$() for maintaining the letters of the word. Each time the user makes a guess, the program must display the word with the letters in the correct position. For example, consider the word "SHOE". At the start of the game, when no letters have yet been guessed, the word definition would be as shown below:

and variable array SOFAR\$() would contain:

SOFAR\$(1) = " _ "	SOFAR\$(2) = " _ "
SOFAR\$(3) = " _ "	SOFAR\$(4) = " _ "

Suppose now that the user enters the letter O. The word definition would be updated and re-displayed as shown below:

_ _ O _

and variable array SOFAR\$() would now contain:

SOFAR\$(1) = " _ "	SOFAR\$(2) = " - "
SOFAR\$(3) = " O "	SOFAR\$(4) = " _ "

We therefore use the variable array SOFAR\$() to maintain each of the letters. At the start of the program all of the letter positions will be blank and hence the "-" symbol is used to indicate a blank position. Lines 230-250 form a *for..next* loop to set the opening contents of the variable array SOFAR\$() to " _ ".

We also need to maintain a list of all the letters that the user enters and variable USED\$ is used to store these. Line 260 therefore clears the variable USED\$ ready for use.

(5) ASK USER FOR GUESS AND PERFORM CHECKS

```
280 repeat
290 :
300 print "Word so far"
310 for A=1 to L
320 print SOFAR$(A);
330 next A
340 print
```



```
350 print "Letters used so far"
360 print USED$
370 print
380 print "Guess a letter"
390 GUESS$=input$(1)
400 GUESS$=upper$(GUESS$)
410 :
420 rem HAS INPUT LETTER ALREADY BEEN USED
430 if instr(USED$,GUESS$) then print "LETTER
ALREADY USED" : wait key : goto 530
440 :
450 rem ADD LETTER TO LETTERS USED LIST
460 USED$=USED$+GUESS$
470 :
480 rem IS THE LETTER IN THE WORD
490 for A=1 to L
500 if GUESS$=mid$(W$,A,1) then SOFAR$(A)=GUESS$
: inc FOUND
510 next A
520 :
530 rem HAVE ALL THE LETTERS BEEN USED
540 cls
550 until FOUND=L
```

Lines 300-330 display the word so far. Line 310 initiates a *for..next* loop in the range 1 to L (length of the word), line 320 displays the letter and line 330 continues the loop. Notice the semicolon at the end of line 320 to inhibit the line feed/carriage return so that the letters are printed on the same line one after the other so as to display the complete word.

Lines 350-360 display all the letters used so far.

Lines 380-400 request a guess from the user. Line 390 uses the *input\$* command to input a single letter. You may recall that *input\$* does not

require the user to press the RETURN key, and thus when a key is pressed, the program continues. The entered letter is assigned to variable GUESS\$ and line 400 converts this to upper case.

We now need to make a number of checks.

Line 430 checks to see if the user has already used the character before. The *instr* command is used to see if the character GUESS\$ exists within the list of used characters USED\$. If the letter does exist within the list of used letters then a "LETTER ALREADY USED" message is displayed, the program waits for a key to be pressed and program execution is passed to line 530 using the *goto* command. The *goto* command sends program execution directly to the specified line skipping any lines of code that may be in between. If the letter has not already been used then program execution passes on to line 440.

We now know that the letter has not already been used and so can add it to the list of letters used. This is performed by line 460 which adds the letter GUESS\$ to the end of the list USED\$.

We can now check to see if the letter is required for the word. The guessed letter GUESS\$ must be checked against each letter of the word to see if a match is found. Line 490 initiates a *for..next* loop in the range 1 to L (length of the word), line 500 uses the *mid\$* command to check the letter within the word W\$ against the guessed letter GUESS\$. If a match is found, the letter GUESS\$ replaces the "_" character within the variable array SOFAR\$() and the number of letters found (variable FOUND) is increased by one. Line 510 contains the *next* command which continues the loop until all the letters of the word have been checked.

Line 540 clears the screen and line 550 checks to see if all the letters have been found by comparing the number of letters found (FOUND) with the length of the word (L). If there are still more letters to be found, the program loops back to the *repeat* command at line 280. If

all the letters have been found then the program proceeds on to line 560.

(5) END THE GAME

```
570 rem THE WORD HAS BEEN FOUND
580 print "THE WORD WAS ";W$
590 print "Well done you have found the word"
600 end
```

This section is quite straight forward using *print* commands to display the text. Line 600 ends the program.

(6) WORD DATA

```
640 rem WORD DATA
650 data "shirt","hat","gloves","coat","shoes"
660 data "cup","saucer","plate","spoon","fork"
670 data "dog","cat","mouse","hamster","bird"
680 data "computer","disk","program","list","memory"
690 data "mother","father","son","daughter","granddad"
700 data "grass","flower","shed","field","swing"
710 data "television","radio","stereo","satellite","speaker"
720 data "car","lorry","bike","bus","train"
730 data "apple","pear","orange","banana","peach"
740 data "toy","teddy","doll","puzzle","book"
```

Lines 640-740 list the word data.

You may like to increase the number of words that the program uses. Can you work out how to do this ?

You may add as many new data statements as required but must adjust

the dimension of the variable array W\$() and also the range of the loop that reads the words in to the array. For example, if you want to use 1000 words you will need to make the following changes:

```
80 dim W$(999), SOFAR$(20)
120 for A = 0 to 999
180 R = rnd(999)
```

Well that concludes the word guessing game. It introduced some quite advanced use of string variables and arrays so spend some time studying the program. Try looking at the program listing whilst playing the game so that you can compare the operation with the code.

Chapter 12

Windows & Character Sets

So far we have considered the screen as a whole, displaying information at various positions using different colours, etc., but STOS allows the screen to be separated in to a number of smaller screens known as *windows*. Windows are one of the features which make the Atari ST range of computers so easy to use and you have probably already encountered the feature when looking at and copying disks. When the ST is switched on it displays the GEM desktop which relies on the use of windows for various operations. For example, when you click on one of the floppy disk drive icons, a window is opened and the contents of the disk displayed. The window can be re-sized, scrolled, moved to a new position on the screen and closed when you have finished. STOS windows look slightly different to the GEM windows but they still offer the same, if not more, flexibility and are very useful indeed for us programmers.

Imagine a window as a mini screen that lays on top of the main screen. We use the words 'lay on top' because a window can be opened on top of existing information without disturbing the

information that it is hiding. When the window is closed, the information that was previously hidden is restored, and hence the window appears to lay on top of the existing information.

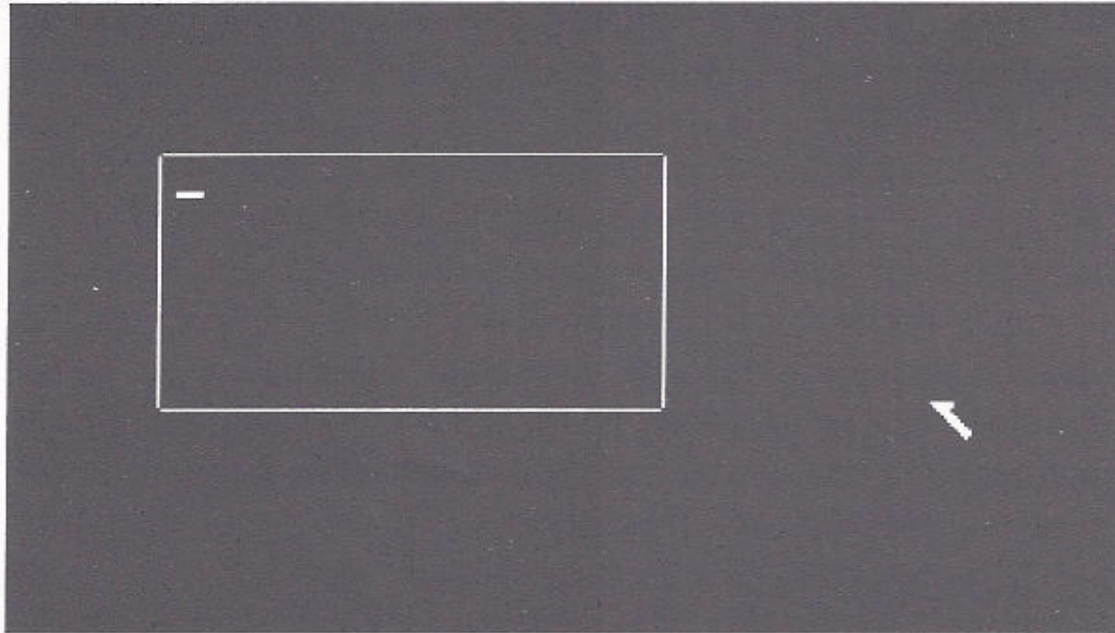
A maximum of thirteen windows can be opened and managed simultaneously.

OPENING A WINDOW

Load the following:

```
10 rem OPEN A WINDOW
20 rem PROGRAM = A:\WINDOWS\OPENWIN
30 :
40 mode 0: key off
50 windopen 1,5,5,19,10
```

Run the program and it should produce a window as shown below:



Now try pressing the Return key a few times and you will notice that the text cursor is limited to the window. If you try moving the mouse pointer you will see that this is still free to roam the entire screen as the window commands do not effect the mouse. When a window is opened it becomes active and STOS assumes that any further operations should be carried out inside the window. We shall see later how to switch control between windows, but for now, press the UNDO key twice to reset the editor and hence clear the window.

Windows are created or opened using the *windopen* command. The format of the *windopen* command is shown below:

`windopen N, X, Y, W, H, BORDER, CHARSET`

- N:** allocates a number by which the window can be identified. STOS allows thirteen separate windows and hence this value can range from 1 - 13.
- X/Y:** the text coordinates of the top left hand corner of the window. X indicates the position across the page and Y indicates the position down the page. Remember that low resolution offers 40x25 and medium / high resolution offers 80x25.
- W:** indicates the width of the window.
- H:** indicates the height of the window.
- BORDER:** indicates the style of border around the outside of the window. There are 16 different styles available and these are assigned the values 0 - 15. The BORDER parameter is optional, and if omitted, a standard plain border is used.
- CHARSET:** indicates the character set. STOS allows different character sets or styles of text to be used within windows but do not worry about this yet as we shall take a

detailed look later. The CHARSET parameter is optional, and if omitted, the standard character set is used.

BORDERS

The window in the last program had a standard, single line border as no border parameter was specified. This may well suit many programs but STOS also allows fancy borders to be used. Fancy borders are specified by a number in the range 0-15. There are thus fifteen different styles available plus border style zero which produces the window without a border.

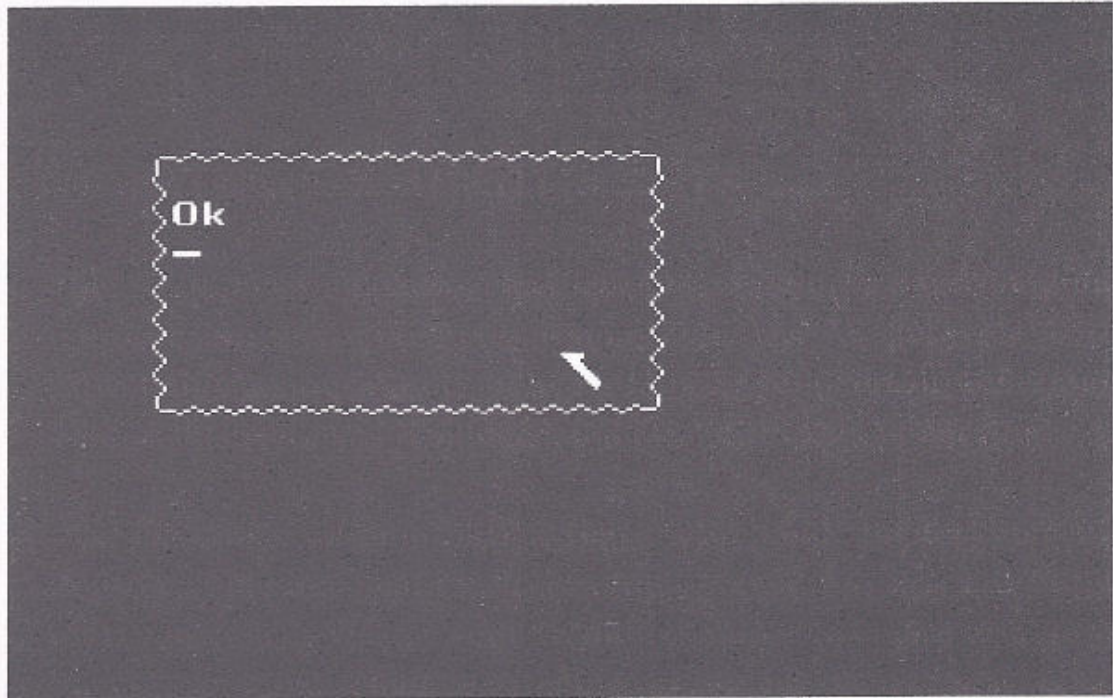
Load the following:

```
10 rem OPEN A WINDOW 2
20 rem PROGRAM = A:\WINDOWS\OPENWIN2
30
40 key off: mode 0
50 windopen 1,5,5,19,10,10
```

Run the program and it should produce a window as shown on the opposite page.

Border style five has been included as part of the *windopen* command and this produces a 'zig-zag' type effect.

When a window is opened the border style is selected as part of the *windopen* command but this can be changed at any time using the *border* command. You may, for example, wish to highlight a certain window by changing its border half way through your program. We shall now produce a program that illustrates all of the border styles.



Load the following:

```
10 rem WINDOW BORDERS
20 rem PROGRAM: A:\WINDOWS\BORDERS
30 :
40 rem SET SCREEN RESOLUTION LOW
50 key off : mode 0
60 :
70 rem OPEN A WINDOW
80 windopen 1,1,1,39,20
90 :
100 rem DISPLAY BORDERS
110 for COUNT = 1 to 15
120 border COUNT
130 clw
140 print "This is border ";COUNT
150 print
160 print "Press a key to see next border"
170 wait key
180 next COUNT
```


Run the program and press any key to cycle through the various border styles.

Look very carefully at the program.

Line 80 opens a window which is 39 characters wide by 20 characters high, and the top left corner of the window is positioned one character from the left hand side of the screen and one character down from the top of the screen.

Lines 110-180 form a *for..next* loop to display all fifteen of the border styles. The *border* command in line 120 changes the border of an existing window and the *clw* (clear window) command at line 130 clears the window. This is very similar to the *cls* (clear screen) command except that it only effects the active window rather than the entire screen.

All of the borders are shown at the end of this chapter.

CLOSING WINDOWS

One of the nice things about windows is that they can be opened 'on top' of existing information, and when closed again, STOS automatically restores the original information. This is illustrated by the next program.

Load the following:

```
10 rem OPENING AND CLOSING WINDOWS
20 rem PROGRAM = A:\WINDOWS\WINCLOSE
30 :
40 key off: mode 0
50 :
60 rem DISPLAY INFORMATION
```

```
70 locate 1,10
80 print "Existing information on the screen"
90 wait key
100 :
110 rem OPEN WINDOW
120 windopen 1,0,5,30,8,7
130 wait key
140 :
150 rem CLOSE WINDOW
160 windel 1
```

Run the program and the message will be displayed across the centre of the screen. Press a key and a window will open which covers the message. Press another key and the window will be closed. Notice that, when the window is closed, the original message is restored. Let us briefly look at the program itself.

Line 40 switches off the function key window and selects low screen resolution.

Lines 70-90 display the message and wait for the user to press a key.

Once a key is pressed, lines 120-130 open a window and wait for the user to press a key again.

Once a key is pressed, line 160 closes the window using the *windel* command. Following the *windel* command is the number of the window that is to be deleted or closed.

Therefore, when a window is closed any information hidden under the window is automatically restored.

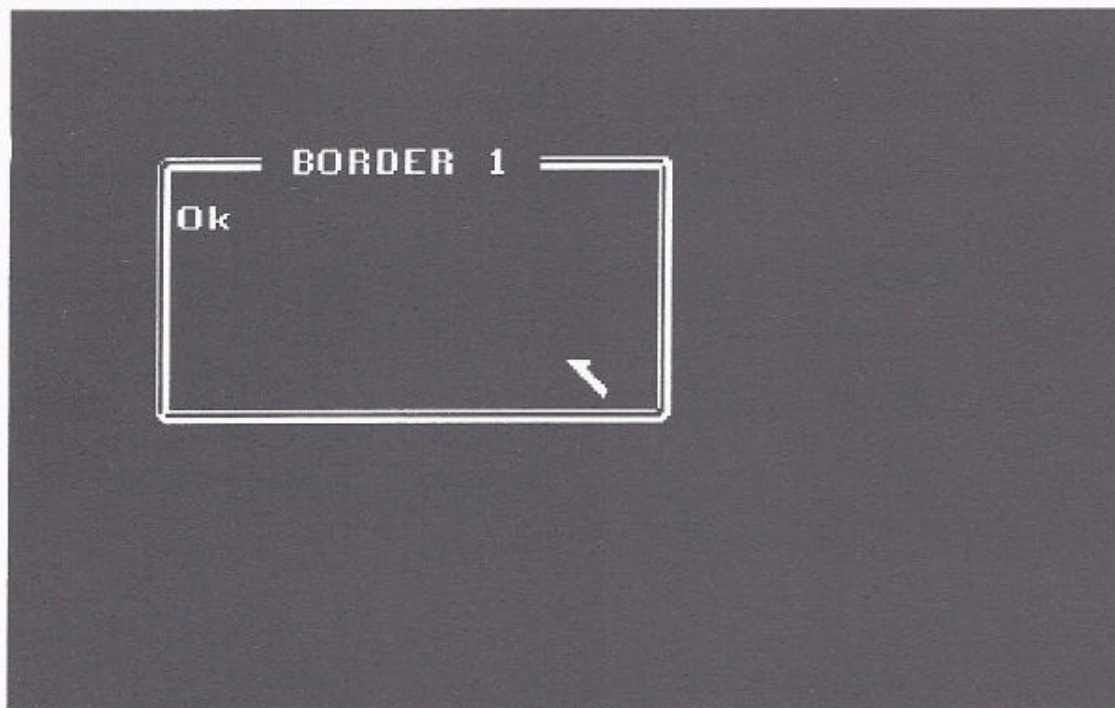
ADDING A TITLE

STOS allows a title to be assigned to each open window. This is displayed as part of the border at the top of the window and is assigned using the *title* command.

Load the following:

```
10 rem ADDING A WINDOW TITLE
20 rem PROGRAM = A:\WINDOWS\TITLE
30 :
40 key off: mode 0
50 windopen 1,5,5,19,10,4
60 title " BORDER 4 "
```

Run the program and you will see a window as shown below:



MULTIPLE WINDOWS

We know that STOS can display thirteen separate windows

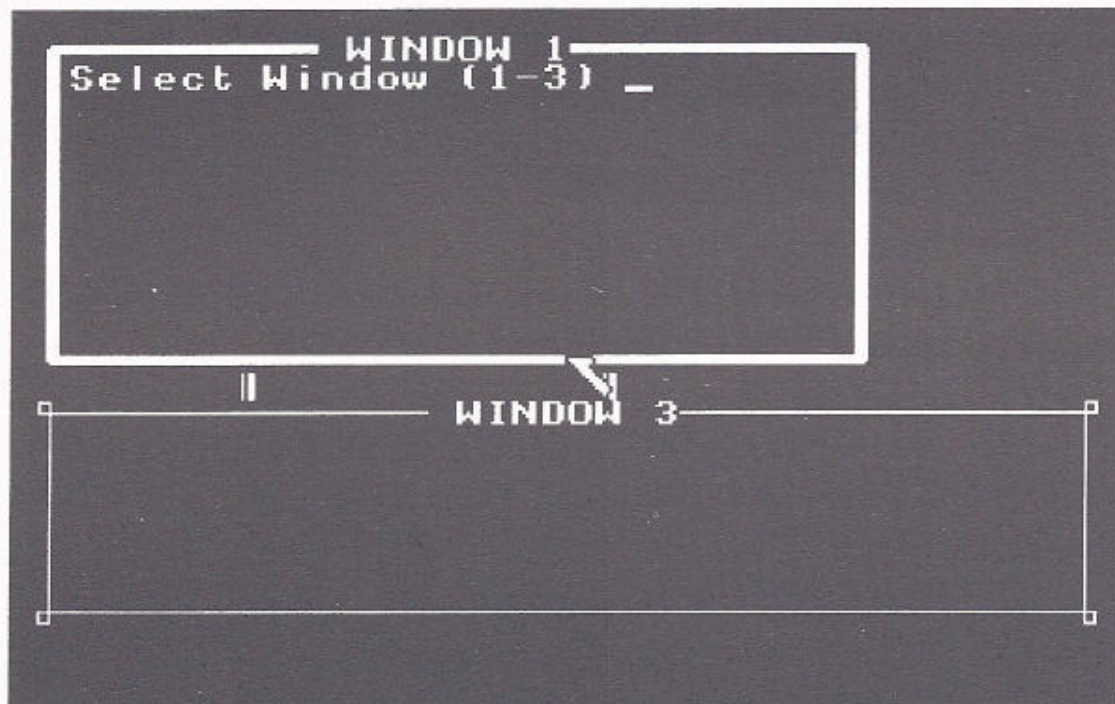
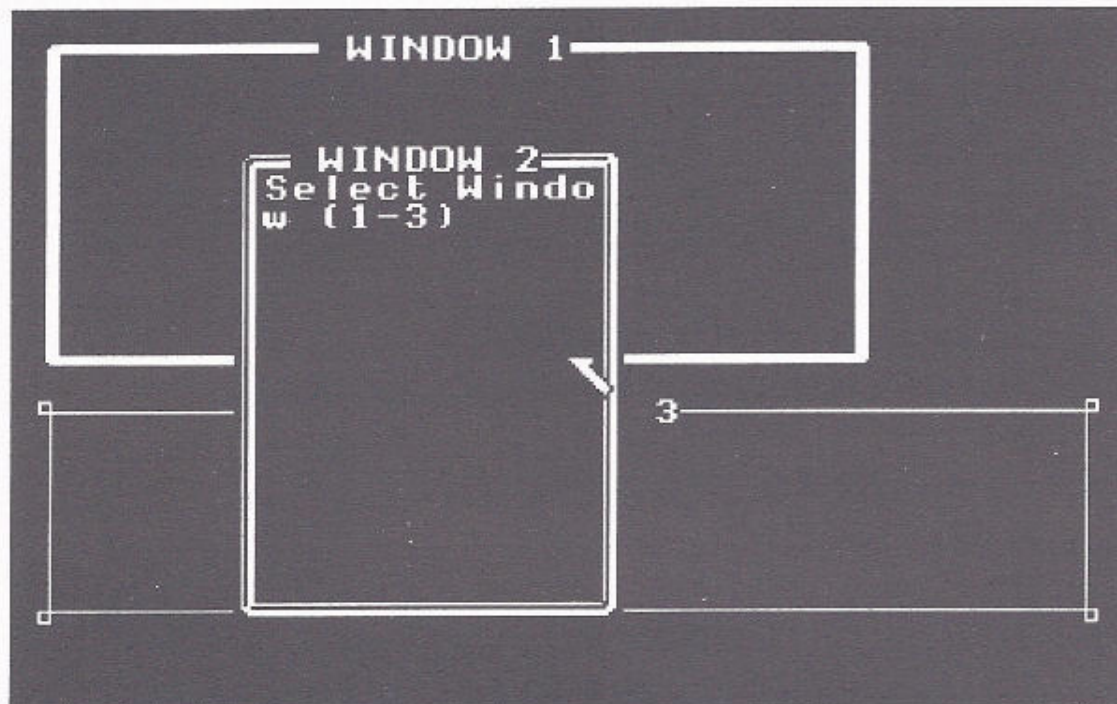
simultaneously but we need a way of specifying and switching the windows. For example, one window may be hidden behind another, and before it can be used it will have to be activated or brought to the front.

Load the following:

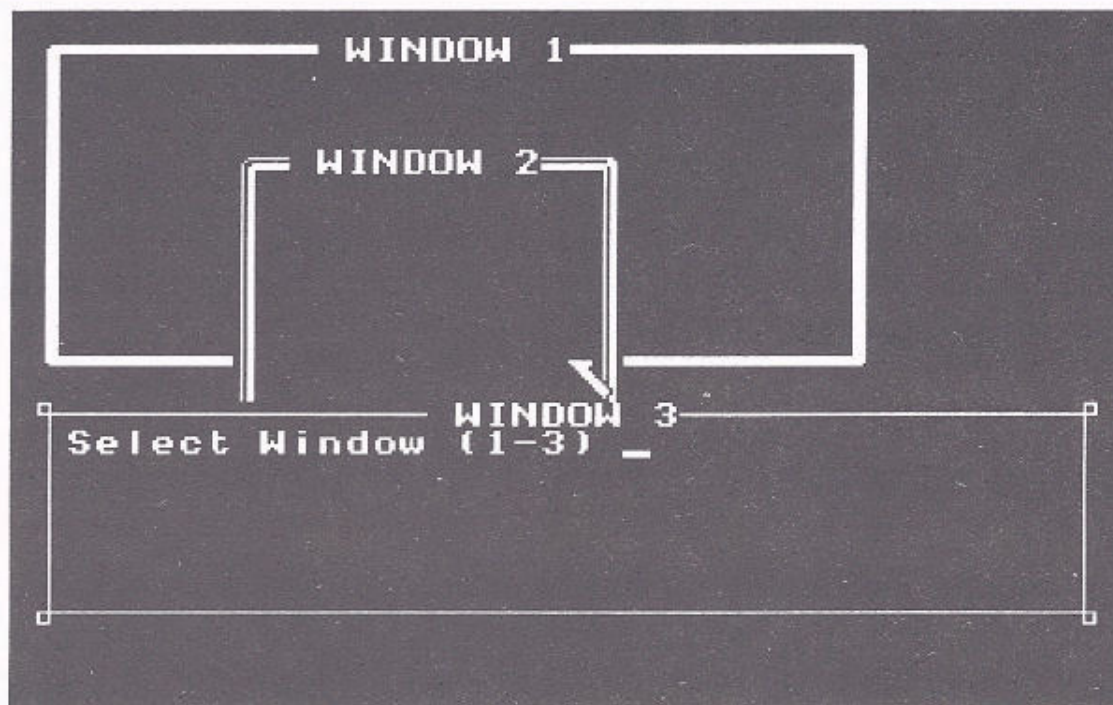
```
10 rem MULTIPLE WINDOWS
20 rem PROGRAM: A\WINDOWS\MULTI
30 :
40 rem SET SCREEN RES LOW
50 key off : mode 0
60 :
70 rem OPEN WINDOWS AND ADD TITLES
80 windopen 1,1,1,30,12,2
90 title " WINDOW 1 "
100 windopen 2,8,5,16,17,4
110 title " WINDOW 2 "
120 windopen 3,1,14,38,8,12
130 title " WINDOW 3 "
140 :
150 rem ASK USER TO ENTER WINDOW NUMBER
160 input "Select Window (1-3)"; WIN
170 :
180 rem CHECK THAT INPUT IS WITHIN RANGE
190 if WIN < 0 or WIN > 3 then 160
200 :
210 rem SWITCH WINDOWS
220 clw
230 window WIN
240 :
250 rem GO BACK FOR ANOTHER GO
260 goto 160
```

The program creates three windows each of which overlaps or hides

part of the other windows. This is shown below:



The user is asked to enter a window number between 1 and 3 and this then becomes the current window. Notice that the selected window is brought to the front of the screen hiding any other information that



may be in the way.

The user enters number 1 and window 1 is activated.

The user enters number 2 and window 2 is activated.

The user enters number 3 and window 3 is activated.

The *window* command in line 230 is used to actually activate the window. The window is redrawn so that it appears to move to the surface, hiding any other information underneath. Another command very similar to *window* is *qwindow*. This activates the specified window but does not redraw it. The window therefore stays hidden but any information printed to the window will be displayed. This can cause the display to become corrupt as can be demonstrated if you change line 230 to:

```
230 qwindow WIN
```

The *qwindow* command should therefore only be used when the window you wish to activate is not covered by any other windows or information. If the window is hidden from view, the *window*

command must be used so that the window is redrawn. The STOS manual states that *qwindow* works faster than *window* which makes sense when you consider that no redrawing is required. Whilst this may be the case, the difference in speed is unlikely to make much difference to most programs and if you always use *window* in preference to *qwindow* this will guarantee correct operation regardless of the position and state of the windows.

MOVING WINDOWS

Suppose that, once a window has been opened, you wish to move it to a new position. You could close the window and open a new one at the required position, but this would require quite a lot of program code as the original contents of the window would also have to be restored. Luckily STOS offers a far easier way of moving windows around in the form of the *windmove* command. The *windmove* command moves the current window and its contents to a new position as specified below:

windmove X,Y

The operation is carried out on the current window where X and Y specify the top left hand corner of the new position. If the window you wish to move is not active then it must be activated using the *window* command before the *windmove* operation is attempted.

Load the following:

```
10 rem MOVING WINDOWS
20 rem PROGRAM: A:\WINDOWS\WINDMOVE
30 :
40 rem SET SCREEN RES LOW
50 key off : mode 0
60 :
```

```
70 rem OPEN WINDOW
80 windopen 1,1,1,20,5
90 title " I KEEP MOVING "
100 print "Press any key"
110 :
120 rem WAIT FOR USER TO PRESS KEY
130 wait key
140 :
150 rem MOVE WINDOW TO RANDOM POSITION
160 X=rnd(19)
170 Y=rnd(17)
180 windmove X,Y
190 :
200 goto 130
```

Run the program and press any key on the keyboard.

The program creates a window and then moves it to a new position each time that a key is pressed.

The window is opened in line 80. Line 90 adds a title and line 100 prints some text in the window. Line 130 waits for the user to press a key. When a key is pressed, lines 160 and 170 choose a new, random position for the window. Line 180 uses the *windmove* command to move the window and its contents to the new position. Line 200 sends the program back for another go. To end the program press CONTROL+C.

COLOURED WINDOWS

Windows are not limited to black and white and the *pen* and *paper* commands can be used to produce some very professional effects. Try the following programs to see some of the effects that are possible:


```
(1) 10 rem COLOURED WINDOWS
    20 rem PROGRAM: A:\WINDOWS\WINCOL1
    30 :
    40 rem SET SCREEN RES AND COLOURS
    50 key off : mode 0 : flash off
    60 palette $0,$777,$700,$007
    70 BLACK=0 : WHITE=1 : RED=2 : BLUE=3
    80 :
    90 rem OPEN WINDOW
    100 windopen 1,1,1,35,10,12
    110 pen WHITE
    120 paper RED
    130 print "COLOURED WINDOW"
```

This program opens a window and displays a white message on a red background. Notice how the colour index numbers have been assigned to variables (line 70) so that the colours can be referred to by name rather than number.

```
(2) 10 rem COLOURED WINDOWS
    20 rem PROGRAM: A:\WINDOWS\WINCOL2
    30 :
    40 rem SET SCREEN RES AND COLOURS
    50 key off : mode 0 : flash off
    60 palette $0,$777,$700,$007
    70 BLACK=0 : WHITE=1 : RED=2 : BLUE=3
    80 :
    90 rem OPEN WINDOW
    100 paper RED
    110 windopen 1,1,1,35,10,12
    120 pen WHITE
    130 print "COLOURED WINDOWS"
```

This time the paper colour is changed before the window is opened.

This results in the whole background of the window changing to the paper colour.

```
(3) 10 rem COLOURED WINDOWS
    20 rem PROGRAM: A:\WINDOWS\WINCOL3
    30 :
    40 rem SET SCREEN RES AND COLOURS
    50 key off : mode 0 : flash off
    60 palette $0,$777,$700,$007
    70 BLACK=0 : WHITE=1 : RED=2 : BLUE=3
    80 :
    90 rem OPEN WINDOW
    100 paper RED
    110 windopen 1,1,1,35,10,12
    120 paper BLUE
    130 clw
    140 pen WHITE
    150 print "COLOURED WINDOW"
```

This program produces a window with a blue background and a red border. The window is opened with the paper colour set to red and thus the window is opened with a red background, but line 120 immediately changes the paper colour to blue and line 130 executes a *clw* command. The *clw* (clear window) command is similar to the *cls* (clear screen) command except that it only clears the active window rather than the whole screen. The window is therefore cleared to blue. The *clw* command only effects the background inside the window and hence the border still remains red.

```
(4) 10 rem COLOURED WINDOWS
    20 rem PROGRAM: A:\WINDOWS\WINCOL4
    30 :
    40 rem SET SCREEN RES AND COLOURS
```



```
50 key off : mode 0 : flash off
60 palette $0,$777,$700,$007
70 BLACK=0 : WHITE=1 : RED=2 : BLUE=3
80 :
90 rem OPEN WINDOW
100 paper RED
110 pen BLUE
120 windopen 1,1,1,35,20,15
130 print "COLOURED WINDOWS"
```

This example uses border style 15 which produces a chequered effect with the pen and paper colours. By setting the pen and paper colours before the window is opened, we produce a nice red and blue checked border.

The range of border styles and colours allow a vast selection of effects to be achieved. Try experimenting for yourself and remember that each window can have its own colours.

Load the following:

```
10 rem COLOURED WINDOWS
20 rem PROGRAM: A:\WINDOWS\WINCOL5
30 :
40 rem SET SCREEN RES AND COLOURS
50 key off : mode 0 : flash off
60 palette $0,$777,$700,$7
70 BLACK=0 : WHITE=1 : RED=2 : BLUE=3
80 :
900 rem OPEN TWO WINDOWS
100 windopen 1,1,1,19,22,4
110 windopen 2,21,1,19,22,8
120 :
130 rem DISPLAY INFO IN WINDOWS
140 window 1
```

```
150 paper RED: pen WHITE
160 print "I am window 1"
170 window 2
180 paper WHITE: pen BLUE
190 print "I am window 2"
```

This opens two windows each with different pen and paper attributes.

WINDOW SCROLLING

You have probably noticed, whilst programming in STOS, that the screen scrolls upwards whenever the text cursor reaches the bottom of the page. We have all seen the credits at the end of television programs where the directors and actors details move up the screen. The computer screen works in the same way with old information scrolling off the top of the screen as new information appears at the bottom. You can see the effect of this by continually pressing the Return key. When displaying information within windows we can instruct STOS to switch this scrolling on and off. If scrolling is switched on, existing information 'roles' out the top of the window to make way for the new information. If scrolling is switched off, the text cursor jumps back up to the top of the window rather than scrolling the contents. The following examples illustrate this.

Load the following:

```
10 rem SCROLLING ON
20 rem PROGRAM: A:\WINDOWS\SCROLL1
30 :
40 rem SET SCREEN RES
50 key off : mode 0
60 :
70 rem OPEN WINDOW
80 windopen 1,1,1,20,15,11
```



```
90 :  
100 rem SWITCH ON WINDOW SCROLLING  
110 scroll on  
120 :  
130 rem DISPLAY INFORMATION  
140 for A = 1 to 30  
150 print A  
160 wait 10  
170 next A
```

Run the program and it will list numbers 1-30. When the bottom of the window is reached, the old information scrolls upwards and off the top of the window making way for the new information displayed at the bottom of the window.

Now load the following:

```
10 rem SCROLLING OFF  
20 rem PROGRAM: A:\WINDOWS\SCROLL2  
30 :  
40 rem SET SCREEN RES  
50 key off : mode 0  
60 :  
70 rem OPEN WINDOW  
80 windopen 1,1,1,20,15,11  
90 :  
100 rem SWITCH ON WINDOW SCROLLING  
110 scroll off  
120 :  
130 rem DISPLAY INFORMATION  
140 for A = 1 to 30  
150 print A  
160 wait 10  
170 next A
```

This is exactly the same as the previous program except line 110 which now switches off scrolling. Each time the cursor reaches the bottom line of the window it is moved to the top again. The only problem is that the new information simply writes over the existing information and hence the display becomes corrupt. In practice we would have to clear the window before starting to display new information.

As a default STOS assumes window scrolling to be set ON although it is still good practice to include a *scroll on* command.

When window scrolling is on, STOS offers another two commands which allow us to force the contents of the window to be scrolled. These are *scroll up* and *scroll down* and are used to scroll the window, as the names suggest, up and down.

Scroll up moves everything above the text cursor up one line. Note that anything currently displayed on the top line of the window will be scrolled off the top.

Scroll down moves the area below the text cursor down one line. Note that anything displayed on the bottom line of the window will be scrolled off the bottom.

Load the following:

```
10 rem SCROLLING COMMANDS
20 rem PROGRAM A:\WINDOWS\SCROLL3
30 :
40 rem SET SCREEN RES + HIDE MOUSE POINTER
50 key off : mode 0
60 hide on
70 :
80 rem OPEN WINDOW AND DISPLAY MESSAGE
90 windopen 1,1,1,30,23,4
```



```
100 print "Scrolling up and down"
110 :
120 rem SCROLL MESSAGE DOWN
130 locate 0,0
140 for A = 1 to 20
150 scroll down
160 next A
170 :
180 rem SCROLL MESSAGE UP
190 locate 0,20
200 for A = 1 to 20
210 scroll up
220 next A
230 :
240 rem GO BACK FOR ANOTHER GO
250 goto 130
```

Run the program and you will see that it continually scrolls a message up and down within the window.

A window is opened and a message displayed. We already know that the *print* command terminates with a line feed and carriage return and hence line 100 will leave the text cursor on the next line underneath the message. To scroll the message down we first need to move the text cursor back to the line containing the message and this is carried out using a *locate* command at line 130. The window is then scrolled down twenty times using the *scroll down* command within a *for..next* loop. An important point to note is that the scroll commands do not move the text cursor. This means that, although we have scrolled the screen twenty times, the text cursor is still positioned at the top of the screen. We therefore need to position the text cursor below the message at the bottom of the screen before we can use the *scroll up* command. Line 190 performs this re-positioning using the *locate* command and lines 200-220 scroll the message up the window twenty times. Line 250 sends the program back to line 130 where the whole

process starts again.

CHARACTER SETS

In addition to the pen and paper attributes, each window can also incorporate its own character set. A character set is a set of graphical images that are used to display letters, numbers, etc. that we type in on the keyboard. Although the standard characters are acceptable, we can make our programs look really professional and exciting by incorporating different styles of text on the screen.

Before anything can be displayed on the screen, STOS must load a character set. When STOS is loaded it automatically installs three character sets in to memory; one each for low, medium and high resolution. These character set files can be found in the STOS folder and are named 8X8.CRO (low resolution), 8X8.CR1 (medium resolution) and 8X16.CR2 (high resolution). Additional character sets must be loaded by the programmer and can then be assigned to specific windows as part of the *windopen* command.

MEMORY BANKS

STOS allows all sorts of external information to be loaded and used within our programs - character sets, pictures, sprites, 3D graphics, music, etc. and STOS must maintain and make this information available as and when required. It does this using *memory banks*.

Memory banks are sections of memory that are set aside by STOS for maintaining all of this external data. There are fifteen memory banks as shown over the page.

<u>CATEGORY</u>	<u>FOR STORING</u>	<u>BANK</u>	<u>TYPE</u>
Sprites	Sprite Information	Bank 1	Permanent
Icons	Icon Information	Bank 2	Permanent
Music	Music Information	Bank 3	Permanent
3D	3D Extension Info	Bank 4	Permanent
Set	Character Set Info	Banks 1-15	Permanent
Screen	Picture Information	Banks 1-15	Temporary
Datascreen	Picture Information	Banks 1-15	Permanent
Work	Workspace	Banks 1-15	Temporary
Data	General Data	Banks 1-15	Permanent
Menu	Menu Information	Bank 15	Temporary
Program	Machine Code Info	Banks 1-15	Either

The above chart indicates the type of data that can be maintained in the memory banks. There are some categories of data that are automatically saved in a particular memory bank. For example, sprite data is always maintained in memory bank 1 but the programmer can define any of the banks to store specific information.

Notice also that there are two types of memory bank - permanent and temporary. Permanent memory banks, once assigned, are saved along with the program but temporary banks are cleared each time that the program is run.

Do not worry too much about memory banks at this stage. In later chapters we shall be looking at screen, datascreen and sprite data and all will become perfectly clear.

LOADING A CHARACTER SET

Before loading a character set we need to reserve a memory bank to hold it. When reserving memory banks for character sets we have to specify the size of the bank required. The easiest way to establish the

size is to look at the directory of the disk containing the character set. You can do this by typing DIR "*.mbk". As recommended in the STOS User Guide, you should round the value up to the nearest 1000 bytes to guarantee having enough space. Next we have to reserve the data bank that is going to hold the character set data and then the character set can be loaded. Looking back at the memory bank chart you will see that any of the memory banks 1-15 can be used to store character sets (SET) and that these will be permanent.

Disk 1 contains a new character set named "DATAFONT.MBK" which is inside the windows folder, so let's load this.

(1) Establish the size of the character set file

Place disk 1 in to drive A and enter the following:

```
dir "a:\windows\*.mbk"
```

This will display a list of all the files with the ".MBK" extension. The ".MBK" extension indicates that the file contains memory bank information.

```
Drive A, path: \WINDOWS  
DATAFONT.MBK    2322
```

2322 bytes used

The number at the end of the file name indicates the size of the file and you will see that the character set file has a size of 2322 bytes.

(2) Round the file size up to the nearest, thousand bytes

In this case, 2322 is rounded up to 3000.

(3) Reserve a memory bank and load the file

Whilst any of the fifteen banks could be used, we shall use memory bank 5.

We now have to reserve the memory bank and inform STOS of the type of information that is to be stored within the bank. We do this using the *reserve as* command as shown below:

Enter the following:

reserve as set 5,3000

reserve as set indicates that the information is character SET information, the number 5 indicates the memory bank number and the number 3000 indicates the size of the bank.

The character set file can now be loaded in to the memory bank.

Enter the following:

load "a:\windows\datafont.mbk",5

The number at the end of the load command indicates the number of the memory bank that the file should be directed to - in this case memory bank 5.

The new character set is now safely stored in memory bank 5 and can be used within a program. To prove that the data has been loaded simply list the program.

Enter the following:

list

Reserved memory banks:

5 chr set S:\$0EF600 E:\$0EFF00 L:\$000900

All the reserved memory banks are shown at the end of the program listing. The hexadecimal numbers indicate the start/end positions of the memory bank in memory and the length of the memory bank. We are not concerned with these numbers as they play no part as regards our programming at present.

Looking back to the memory bank chart you will see that SET category banks are permanent. This means that the character set will now remain in memory and will be saved along with any programs that you save to disk. Therefore, the data only needs to be loaded once and can then be used whenever required during the program.

A permanent memory bank can only be erased using the *erase* command.

Enter the following:

```
erase 5  
list
```

This time nothing is listed as the data has been erased and the memory is free to be used again.

We shall now see how to use the character set data once it has been loaded.

DISPLAYING NEW CHARACTER SETS

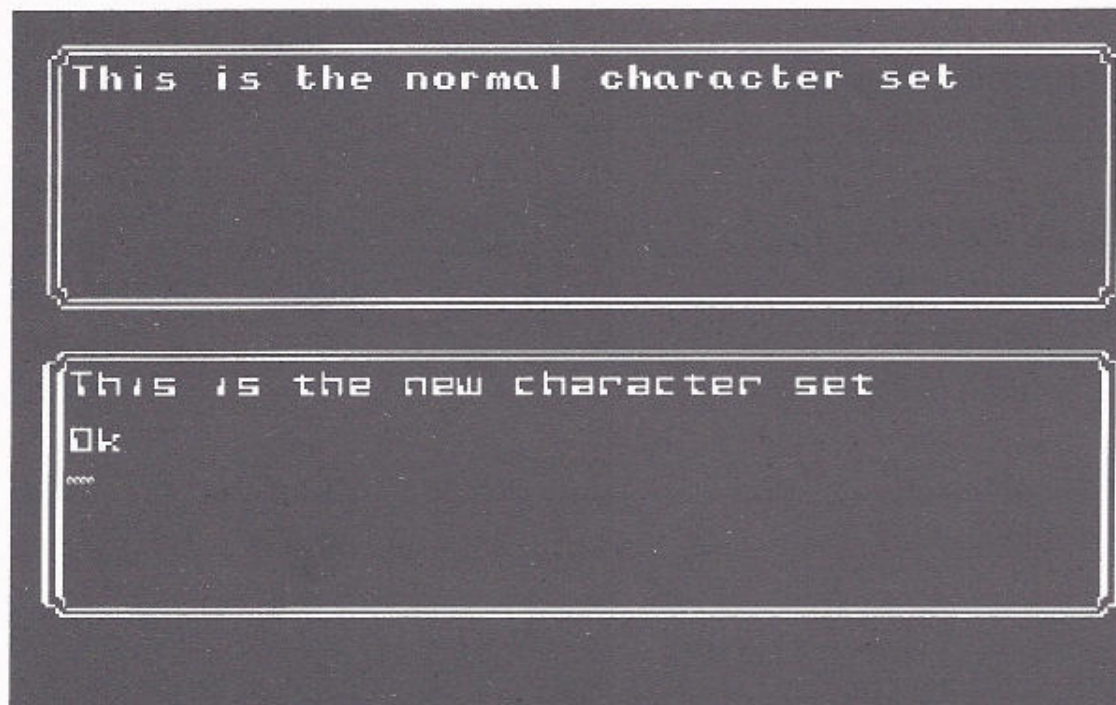
Load the following:

```
10 rem TWO DIFFERENT CHARACTER SETS
```



```
20 rem PROGRAM: A:\WINDOWS\CHARSET1
30 :
40 rem SET SCREEN RES
50 key off : mode 0
60 :
70 rem OPEN WINDOWS
80 windopen 1,0,0,40,10,7
90 print "This is the normal character set"
100 :
110 windopen 2,0,12,40,10,7,4
120 print "This is the new character set"
```

Run the program and you will see two windows as shown below:



Note that the character set data has been saved as part of the program and hence there is no need to load it again. If you are typing the program in direct you will need to load the character set data as previously shown.

The program opens two windows, one of which uses the original

system character set and the other which uses the new character set. The program is quite straight forward but look carefully at line 110 which contains the *windopen* command with the optional character set parameter on the end. Values 1-3 represent the system character sets and values 4-9 represent external character sets.

Remember also that windows can be opened without a border allowing the new character set to be used for general printing to the screen.

Load the following:

```
10 rem DISPLAY A NEW CHARACTER SET
20 rem PROGRAM: A:\WINDOWS\CHARSET3
30 :
40 rem SET SCREEN RES AND COLOUR PALETTE
50 key off : mode 0
60 :
70 rem OPEN WINDOW WITH NEW CHARACTER SET
80 windopen 1,0,0,40,25,0,4
90 :
100 rem DISPLAY TEXT
110 pen 12
120 print "This is a new character set which"
130 print "was designed using the STOS"
140 print "font editor."
```

LARGE TEXT

STOS, as previously mentioned, has three default character sets one of which is used for each screen resolution. Characters are constructed from a series of dots or *pixels* arranged in a block and the higher the resolution the higher the number of pixels available. The high resolution characters are thus constructed from 8x16 (8 wide x 16 high) pixel blocks whilst the low resolution characters are constructed

from 8x8 (8 wide x 8 high) pixel blocks, and when high resolution characters are displayed on the low resolution screen, they appear to be twice the height. This is a nice effect for main headings and is illustrated by the following program.

Load the following:

```
10 rem DISPLAY LARGE TEXT
20 rem PROGRAM A:\WINDOWS\CHARSET2
30 :
40 rem SET SCREEN RES LOW
50 key off : mode 0
60 :
70 rem OPEN WINDOW
80 windopen 1,0,0,40,10,2,3
90 print "BIG TEXT"
```

Run the program and you will see that it opens a window and displays a message in enlarged text. Notice that character set 3 (high resolution) has been specified at the end of the *windopen* command in line 80.

MIXING CHARACTER SETS

STOS allows one character set to be assigned to each window. The required character set is specified as part of the *windopen* command and thus only one character set can be displayed within each window. Suppose though that we require three different character styles displayed on the screen. This can be achieved by opening three windows, one below the other, each with a different character set and no border. This would give the effect of direct printing to the screen and the user would not be aware that windows were in use.

Load the following:

```
10 rem MIXING CHARACTER SETS
20 rem PROGRAM: A:\WINDOWS\CHARSET4
30 :
40 rem SET SCREEN RES AND COLOUR PALETTE
50 key off : mode 0 : flash off
60 palette $0,$777,$700,$70
70 BLACK=0 : WHITE=1 : RED=2 : GREEN=3
80 :
90 rem OPEN WINDOWS
100 windopen 1,0,0,40,2,0,4
110 pen GREEN
120 centre "THE BEGINNERS GUIDE TO STOS BASIC"
130 windopen 2,0,3,40,5,0
140 pen WHITE
150 centre "A complete programming course for"
160 windopen 3,0,3,40,4,0,3
170 pen RED
180 centre "ATARI ST/STE"
190 :
200 wait key
210 cls
```

Three separate windows are opened, one below the other, each with border style zero (no border) and each with a different character set. Line 200 waits for the user to press a key and line 210 clears the screen. Notice that the windows have not been closed. The *cls* command automatically closes any open windows before clearing the screen and hence no further *windel* operations are required.

SPECIAL EFFECTS

You will probably agree that the various window, character set and

window scrolling facilities allow a vast range of effects to be achieved within our programs. We can combine various character sets, produce a range of fancy borders and scroll information around the screen, all of which can be used to produce some very nice title screens. The following program illustrates the sort of thing that could be achieved.

Load the following:

```
10 rem TITLE SCREEN USING WINDOWS
20 rem PROGRAM: A:\WINDOWS\CHARSET5
30 :
40 rem SET SCREEN RES AND COLOUR PALETTE
50 key off : mode 0 : flash off
60 palette $0,$777,$457,$756
70 BLACK=0 : WHITE=1 : BLUE=2 : RED=3
80 :
90 rem OPEN WINDOW AND DISPLAY INFORMATION
100 windopen 1,0,0,40,25,0,4
110 curs off : hide on
120 pen BLUE
130 locate 0,12
140 centre "The Beginners Guide to STOS Basic"
150 locate 0,14
160 centre "from MT Software"
170 locate 0,13
180 wait 100
190 :
200 rem SCROLL THE MESSAGE FROM CENTRE
210 for A=1 to 10
220 scroll up
230 scroll down
240 wait 10
250 next A
260 :
270 rem DISPLAY WELCOME MESSAGE
```

```
280 wait 50
290 pen RED
300 centre " *** WELCOME *** "
310 :
320 rem CLEAR THE SCREEN
330 wait 100
340 cls
```

The program combines a new character set with the scrolling commands to produce a title screen. Run the program to see the effect.

Line 100 opens the window which is the same size as the screen, has no border and uses an external character set. The character set is stored in memory bank 5 and has been saved on the disk along with the program.

Lines 130-160 display the title message. This consists of two lines which are positioned at lines 12 and 14 using *locate* commands.

Line 170 locates the cursor on line 13 which is in between the two lines of text. Line 180 waits for two seconds giving the user time to view the screen.

Lines 210-250 form a *for..next* loop and cause the *scroll up* and *scroll down* commands to be executed ten times. The text cursor has already been positioned between the two lines of text and thus each line is scrolled out from the centre of the screen.

Lines 280-300 wait for a second so that the user can see the effect of the scrolling and then display a welcome message in between the two lines of text which are now positioned at the top and bottom of the window.

Line 330 waits for two seconds and line 340 clears the screen and

closes the window.

CREATING CHARACTER SETS

Throughout this chapter we have been using an external character set stored on disk under the name of DATAFONT.MBK. You are probably wondering how we actually produced this file and how new character sets are produced for use within our programs.

STOS comes with a font designer which is loaded as an accessory and allows character sets to be designed, saved to disk and grabbed in to the current program. The font designer is supplied on the STOS Accessory disk under the name of FONTS.ACB. We shall not cover the actual operation of such accessories within this course as our main objective is to master the art of programming. Most of the accessories are fairly straight forward to operate and are adequately described in the STOS Users Guide.

The next two pages show the various border styles available.

WINDOW BORDER STYLES

This is BORDER 0

This is BORDER 2

This is BORDER 1

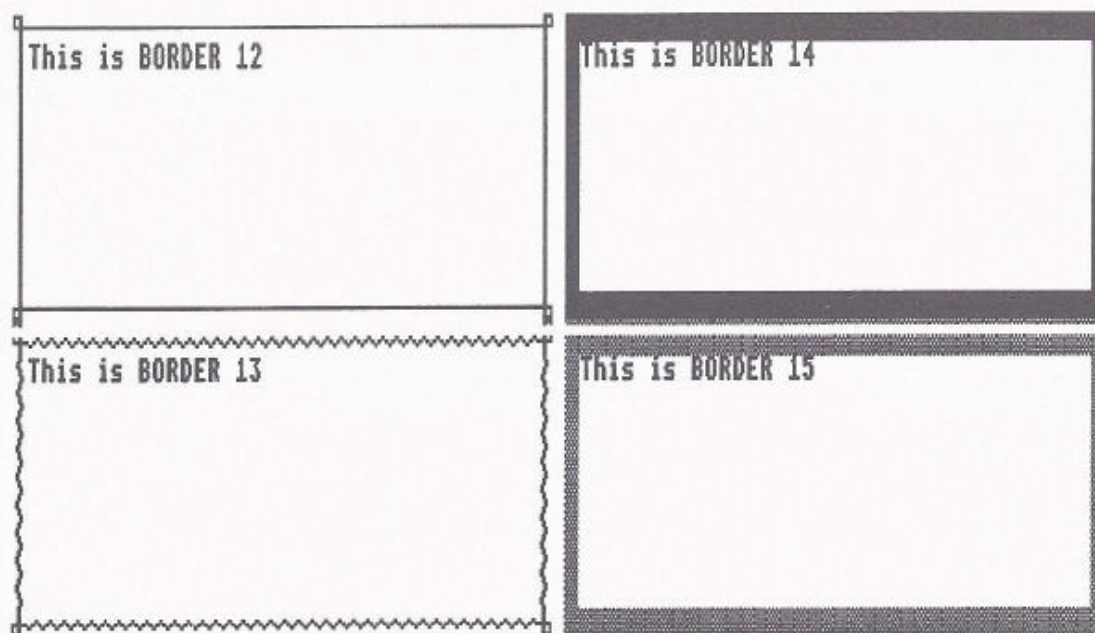
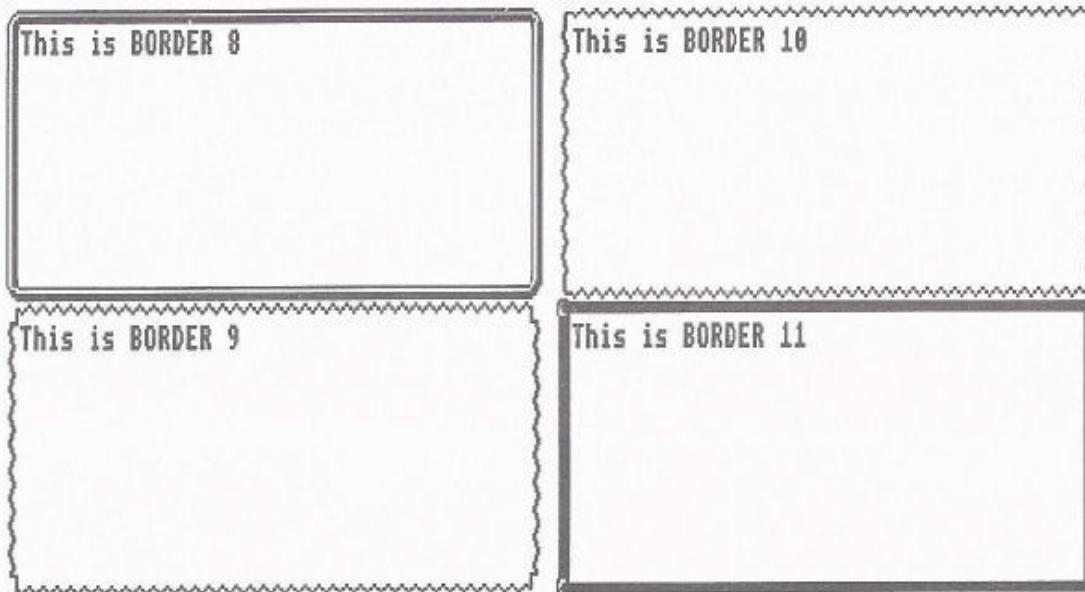
This is BORDER 3

This is BORDER 4

This is BORDER 6

This is BORDER 5

This is BORDER 7



Chapter 13

Graphics

So far we have only displayed characters on the screen but the ST is capable of far far greater things. The ability to draw pictures is known as graphics and the Atari ST is one the most advanced computers in this area. STOS provides a range of commands to allow us to utilise these facilities to the full.

Before continuing it is worth re-capping on screen resolution as it is important to select the correct resolution when designing a program. The following resolutions are available:

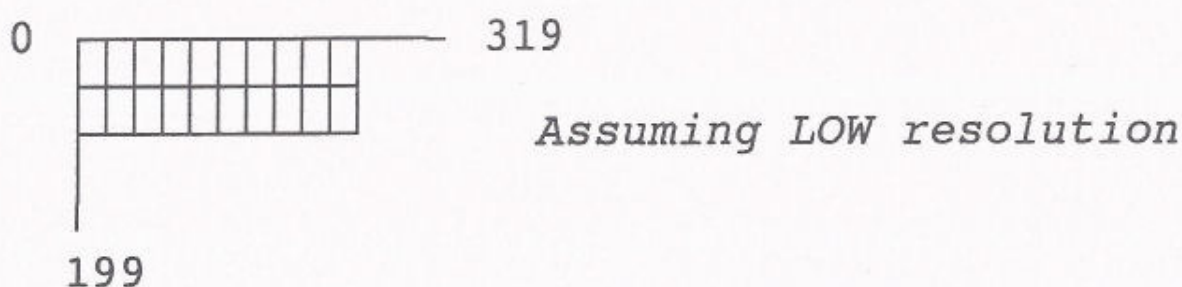
LOW RESOLUTION (MODE 0) offers a graphics resolution of 320 x 200 with 16 different colours displayed simultaneously.

MEDIUM RESOLUTION (MODE 1) offers a graphics resolution of 640 x 200 with 4 different colours displayed simultaneously.

HIGH RESOLUTION (MODE 2) offers a graphics resolution of 640 x 400 with only black and white available.

If you look closely at the computer screen you will see that it consists of a series of dots as shown in the diagram over the page. These dots

are known as *pixels* (picture elements) and the quality of the graphics is determined by the number of pixels available on the screen. When drawing on the screen we have to specify the required pixel using X and Y coordinates. The X coordinate specifies the position of the point across the screen and the Y coordinate specifies the position of the point down the screen. All measurements are taken from the top left hand corner (the *home* position) and hence this has coordinates X=0 and Y=0. These coordinates are collectively referred to as the graphics coordinates.



As can be seen, the graphics capability varies from mode to mode. Medium resolution offers 640 pixels across the screen by 200 pixels down the screen whilst low resolution only offers 320 pixels across the screen by 200 pixels down the screen. Therefore the graphics provided by medium resolution is better than that offered by low resolution. There is one drawback though, the higher the resolution, the lower the number of colours that are available.

When designing a program the programmer must decide on the screen resolution at the very outset of the program. A computer aided design (CAD) program for designing aircraft would require the highest resolution possible, but only a couple of colours. An art package on the other hand, for drawing pictures with lots of colours, would be better using low resolution. The desired screen mode is set using the *mode* command as seen in previous chapters.

PIXELS AND POINTS

The first graphic command offered by STOS is *plot*. This turns on a single point or pixel at the desired screen location. The colour of the point can also be specified.

Load the following:

```
10 rem THE PLOT COMMAND
20 rem PROGRAM = A:\GRAPHICS\PLOT
30 :
40 rem SET SCREEN RESOLUTION AND PALETTE
50 key off : mode 0 : flash off : hide on
60 palette $0,$700,$70,$7
70 BLACK=0 : RED=1 : GREEN=2 : BLUE=3
80 ink RED
90 :
100 rem PLOT THE POINTS
110 plot 160,100
120 plot 170,100,GREEN
130 plot 180,100,BLUE
140 plot 190,100
```

Run the program and look carefully at the centre of the screen.

The program produces a red dot followed by a green dot followed by a blue dot followed by another red dot.

Line 50 switches of the function key window, sets the screen to low resolution, switches off colour flashing and hides the mouse pointer.

Line 60 sets the colour palette.

Line 80 introduces the *ink* command. This is very similar to the *pen* command and sets the colour for drawing graphics. STOS maintains

two separate pen colours, one for text (*pen*) and the other for graphics (*ink*). The background or paper colour is common to both graphics and text, and as already seen, is set using the *paper* command.

Line 110 contains the *plot* command. This turns on the point or pixel at the screen location 160,100 in the colour previously set by the *ink* command, in this case red.

The format of the plot command is shown below:

plot X, Y, COLOUR

where X is the coordinate across the screen, Y is the coordinate down the screen and COLOUR is an optional parameter that specifies the colour index. If the COLOUR parameter is omitted, the point will be activated using the current ink colour.

Lines 120 and 130 contain further *plot* commands but this time the colour parameter is included. Remember that the colour parameter is optional, and if omitted, the current ink colour is used.

Line 140 displays the last dot in the current ink colour again.

LINES

Now suppose that we wish to draw a line. This could be accomplished by turning on a whole row of pixels using the *plot* command as shown below:

```
10 for X = 100 to 300
20 plot X,100
30 next X
```

This works ok, but STOS offers other commands specifically designed

for displaying lines.

Load the following:

```
10 rem THE DRAW COMMAND
20 rem PROGRAM = A:\GRAPHICS\DRAW1
30 :
40 rem SET SCREEN RESOLUTION AND PALETTE
50 key off : mode 0 : flash off
60 palette $0, $700, $70, $7
70 BLACK=0 : RED=1 : GREEN=2 : BLUE=3
80 :
90 rem DRAW LINES
100 ink red
110 draw 50,50 to 250,50
120 :
130 ink green
140 draw 300,50,300,150
150 :
160 ink blue
170 draw 20,20 to 310,180
```

Run the program and it will draw three lines using the *draw* command. The format of the *draw* command is shown below:

draw X,Y to X1,Y1

where X,Y are the coordinates of the start of the line and X1,Y1 are the coordinates of the other end of the line.

Referring back to the program, line 110 draws a line horizontally across the screen in red, line 140 draws a line vertically down the screen in green and line 170 draws a diagonal line in blue. The *draw* command does not allow the colour to be specified and the line is always drawn using the current ink colour.

Draw can also be used in another form by omitting the first pair of coordinates.

Load the following:

```
10 rem THE DRAW COMMAND
20 rem PROGRAM = A:\GRAPHICS\DRAW2
30 :
40 rem SET SCREEN RESOLUTION
50 key off : mode 0
60 :
70 rem DRAW BOX
80 draw 100,100 to 300,100
90 draw to 300,190
100 draw to 100,190
110 draw to 100,100
```

Run the program and you will see that it draws a box at the bottom of the screen.

When the first coordinates are omitted, the *draw* command draws a line starting at the last plotted point. Looking at the program above.

Line 80 draws a line from 100,100 to 300,100.

Line 90 draws a line from 300,100 to 300,190.

Line 100 draws a line from 300,190 to 100,190.

Line 110 draws a line from 100,190 to 100,100.

The next program illustrates the command further by displaying random coloured lines on the screen.

Load the following:

```
10 rem RANDOM COLOURED LINES
20 rem PROGRAM = A:\GRAPHICS\RANLINES
30 :
40 rem SET SCREEN RESOLUTION LOW
50 key off : mode 0 : flash off
60 :
70 rem PLOT INITIAL POINT
80 plot 100,100
90 :
100 rem DRAW RANDOM COLOURED LINES
110 repeat
120 ink rnd(15)
130 draw to rnd(319),(199)
140 wait 20
150 until mouse key
```

Run the program.

Line 80 plots an initial starting point at coordinates 100,100.

Lines 110-150 form a *repeat..until* loop which keeps executing until one of the mouse buttons is pressed. Line 120 chooses a random number in the range 0-15 and sets the ink colour, line 130 draws a line from the previously plotted point to a new set of random coordinates and line 140 halts program execution for 20/50ths of a second.

We could also draw lines using the mouse.

Load the following:

```
10 rem DRAW LINES USING THE MOUSE
20 rem PROGRAM = A:\GRAPHICS\MOUSEDR
```



```
30 :  
40 rem SET SCREEN RESOLUTION  
50 key off : mode 0 : flash off  
60 :  
70 rem PLOT INITIAL POINT  
80 plot 0,0  
90 :  
100 rem DRAW LINES  
110 repeat  
120 if mousekey = 1 then ink rnd(15) : draw to  
x mouse,y mouse  
130 until mouse key = 2
```

Whenever the left mouse button is pressed, a line is drawn to the position of the mouse pointer. The ink colour is selected at random and the program can be terminated by pressing the right mouse button. Holding the left mouse button down produces a nice multi-coloured drawing effect.

CUSTOMISED LINES

The *draw* command generates solid lines which are one pixel wide but STOS also allows us to define and use customised line styles. There are three attributes that can be specified, the thickness of the line, the style of line (e.g. dotted) and the endings of the line. The line style is defined using the *set line* command, the format of which is shown below:

```
set line MASK, THICKNESS, START, END
```

MASK is a bitmap that indicates the main style of the line. This is represented as a binary number by setting 16 digits to either '1' or '0'. Any points of the line that are to be displayed are set to 1. This really is quite straightforward as the following examples illustrate.

A STRAIGHT LINE would be represented by the following mask value:

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

A DOTTED LINE would be represented by the following mask value:

1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0

.....

The dots can be made larger using the following mask value:

1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0

So we can therefore generate any line style by setting the appropriate points to '1'.

THICKNESS specifies the thickness of the line and can range from a value between 1 for a thin line and 40 for a very thick line.

START and END specify the start and end style for the line. These are as follows:

- 0 = Standard Square Ending
- 1 = Arrowed Ending
- 2 = Rounded Ending

Once the line has been defined it can be used for drawing on the screen. The *draw* command previously encountered can only display the standard line and hence a new command *polyline* is used. This is

quite useful because, once a new line has been defined, you still have access to standard lines using the *draw* command. The *polyline* command also allows multiple lines to be drawn using the one command.

The following program demonstrates the various line styles available. When entering the line mask value ensure to precede the value with a percentage sign (%) to indicate that the number is in binary form. For example, for a solid line you would enter %1111111111111111.

Load the following:

```
10 rem THE POLYLINE COMMAND
20 rem PROGRAM = A:\GRAPHICS\POLYLINE
30 :
40 rem SET SCREEN RESOLUTION AND PALETTE
50 key off : mode 1
60 :
70 rem INPUT REQUIRED LINE PARAMETERS
80 input "Enter line mask";M
900 input "Enter line thickness (1-40)";THICK
100 input "Enter line starting end (1-3)";E1
110 input "Enter line ending end (1-3)";E2
120 :
130 rem DEFINE THE LINE
140 set line M, THICK, E1, E2
150 :
160 rem DRAW THE LINE
170 polyline 50,100 to 400,100
```

Run the program a few times selecting different settings for the line.

As previously stated, the *polyline* command allows multiple lines to be drawn using a single command. We could, for example, construct a triangle using just one command.

Load the following:

```
10 rem DRAW A TRIANGLE USING POLYLINE
20 rem PROGRAM = A:\GRAPHICS\TRIANGLE
30 :
40 rem SET SCREEN RESOLUTION AND PALETTE
50 key off : mode 0
60 ink 9
70 :
80 DEFINE LINE AND DRAW TRIANGLE
90 set line %1111111111111111, 5, 1, 1
100 polyline 50,150 to 250,150 to 150,50 to 50,150
```

Run the program and you will see that it produces a triangle.

POLYGONS

The last program used the *polyline* command to produce an outlined triangle but suppose that we wanted a solid triangle. STOS offers the *polygon* command which is a variant of the *polyline* command and causes the final shape to be filled with the current ink colour and paint style (more about this later).

Load the following:

```
10 rem DRAW SHAPES USING POLYGON
20 rem PROGRAM = A:\GRAPHICS\POLYGON
30 :
40 rem SET SCREEN RESOLUTION AND PALETTE
50 key off : mode 0
60 :
70 rem DRAW SHAPES
80 polygon 10,50 to 100,50 to 100,100 to 10,100 to
10,50
```



```
100 polygon 120,100 to 165,50 to 210,100 to
120,100
```

```
110 polygon 230,75 to 260,50 to 290,50 to 310,75
to 290,100 to 260,100 to 230,75
```

Run the program and it will produce three solid shapes on the screen.

Whilst the *draw*, *polyline* and *polygon* commands can be used to produce various shapes, STOS also provides a number of commands for drawing specific and commonly used shapes.

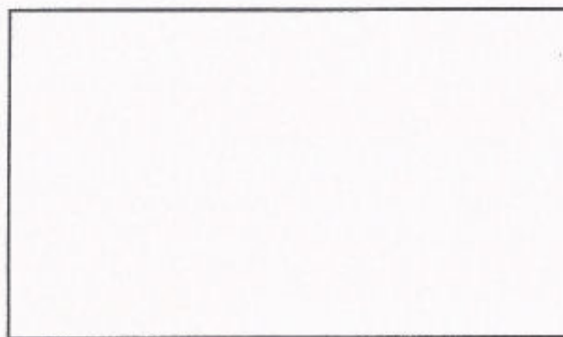
OUTLINE BOXES

We have seen two examples that draw a box, one using *draw* and the other using *polyline*. Both of these commands constructed the box by drawing 4 separate lines for each edge. STOS offers a much quicker way of producing boxes in the form of the appropriately named *box* command. The format of the *box* command is shown below:

`box X, Y to X1, Y1`

where X,Y are the coordinates of the top left hand corner of the box and X1,Y1 are the coordinates of the opposite corner of the box. This is illustrated below:

(X , Y)



(X1 , Y1)

Load the following:

```
10 rem DRAWING BOXES
20 rem PROGRAM = A:\GRAPHICS\BOX
30 :
40 rem SET SCREEN RESOLUTION
50 key off : mode 0
60 ink 13
70 :
80 rem DRAW BOX WITH SQUARE CORNERS
90 box 10,10 to 200,160
100 :
110 rem DRAW BOX WITH ROUNDED CORNERS
120 rbox 70,40 to 110,120
```

Run the program and two boxes will be displayed.

The first box is drawn by line 90. This uses the *box* command to produce a box with top left corner at coordinates 10,10 and bottom right hand corner at coordinates 200,160. Line 120 draws a second box but notice this time that the *box* command is preceded with the letter 'R'. The format of the *rbox* command is identical to that of the *box* command but it produces a box with rounded corners rather than the usual square corners. The second box has top left hand coordinates of 70,40 and bottom right hand coordinates of 110,120.

The *box* and *rbox* commands can produce some nice effects when used to place borders around the edge of the screen or around information that you wish to highlight.

Load the following:

```
10 rem CREATE BORDER USING RBOX
20 rem PROGRAM = A:\GRAPHICS\BORDER
30 :
```



```
40 rem SET SCREEN RESOLUTION AND PALETTE
50 key off : mode 0
60 colour 3,$700
70 :
80 rem DISPLAY MESSAGE
90 locate 8,10
100 print "LOADING...Please Wait"
110 :
120 rem DISPLAY BORDER
130 ink 3
140 rbox 56,72 to 238,93
```

Run the program to see the effect.

FILLED BOXES

The boxes produced with *box* and *rbox* are outline, that is they are produced from single lines and are not solid objects. STOS offers variants of these commands that allow the boxes to be filled in the current ink colour and paint style (more on this later).

Load the following:

```
10 rem FILLED BOXES
20 rem PROGRAM = A:\GRAPHICS\BOXFILL
30 :
40 rem SET SCREEN RESOLUTION
50 key off : mode 0
60 :
70 rem DRAW BOXES
80 ink 13
90 bar 20,20 to 100,50
100 ink 4
110 rbar 10,100 to 300,100
```

Run the program.

This produces two solid boxes on the screen, one with square corners and one with rounded corners. The format of the *bar* and *rbar* commands are exactly the same as that for *box* and *rbox*. So, just to recap, *box* and *rbox* produce outline boxes and *bar* and *rbar* produce solid or filled boxes.

LINE GRAPHS

Computer graphics are often used to represent data in the form of graphs and charts. Whilst data can be presented as a simple list, it is far clearer and easy to relate if presented in graphical form. We shall now write a program that takes a set of data and plots it on a graph.

Load the following:

```
10 rem LINE GRAPH
20 rem PROGRAM = A:\GRAPHICS\LINEGR
30 :
40 rem SET SCREEN RESOLUTION AND PALETTE
50 key off : mode 1 : flash off : curs off : hide on
60 palette $0,$777,$700,$70
70 BLACK=0 : WHITE=1 : RED=2 : GREEN=3
80 :
90 rem READ GRAPH DATA
100 dim INFO(12)
110 for COUNT=1 to 12
120 read INFO(COUNT)
130 next COUNT
140 :
150 rem PLACE GRAPH TITLE AT TOP OF SCREEN
160 pen RED
170 under on
```



```
180 centre "MEGA CORP - Sales Figures for 1991"
190 under off
200 :
210 rem DRAW THE GRAPH AXIS
220 set line %1111111111111111,5,0,1
230 polyline 30,180 to 30,10
240 polyline 30,180 to 600,180
250 :
260 rem PLACE MARKERS ON THE X AXIS
270 for X=30 to 510 step 40
280 draw X,180 to X,185
290 next X
300 locate 1,24
310 print " 0  JAN FEB MAR APR MAY JUN JUL
AUG SEP OCT NOV DEC";
320 :
330 rem PLACE MARKERS ON THE Y AXIS
340 MARK=0
350 for Y=180 to 30 step-15
360 draw 30,Y to 20,Y
370 locate 0, ytext(Y)
380 print MARK;
390 inc MARK
400 next Y
410 :
420 rem PLOT THE DATA
430 ink GREEN
440 plot 30,180
450 for COUNT=1 to 12
460 X=30+(COUNT*40)
470 Y=180-(INFO(COUNT)*15)
480 draw to X,Y
490 next COUNT
500 wait key
510 stop
```

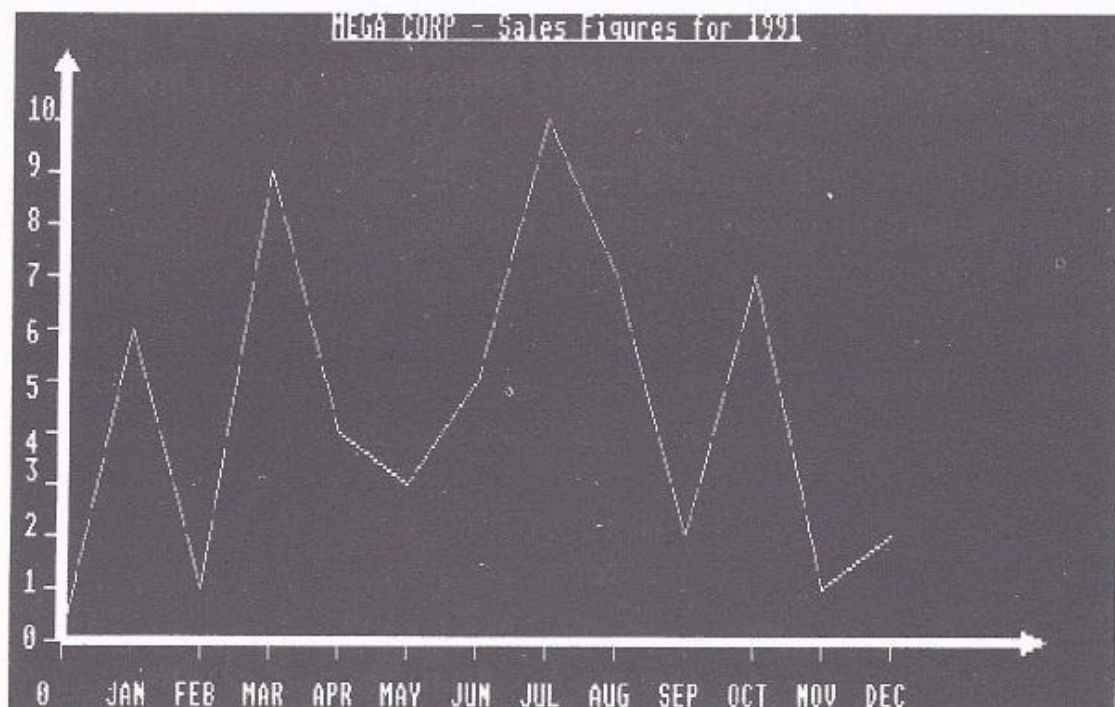
```
520 :  
530 rem DATA FOR GRAPH:  
540 data 6,1,9,4,3,5,10,7,2,7,1,2
```

Run the program.

The program consists of a series of small segments or modules that interface together to perform the overall task. The various operations of the above program are summarised below:

- (1) Set screen resolution and colour palette.
- (2) Read the graph data in to an array.
- (3) Display the graph axis and a title.
- (4) Plot the data on the graph.

We shall look at each of these in turn.



- (1) Line 50 switches off the function key window, switches to medium screen resolution, switches colour flashing off, switches the cursor off and hides the mouse pointer.

Line 60 assigns the colour palette and line 70 assigns a variable to each colour so that the colours can be selected by name rather than by index number.

(2) Lines 100-130 read the graph data in to a variable array. Line 100 dimensions the numeric variable array INFO(). Lines 110-130 form a *for...next* loop to read and assign the 12 data items to the array.

(3) Lines 160-190 display an underlined titled at the top centre of the screen in red.

Lines 210-240 draw the x and y axis of the graph. Line 220 uses the *set line* command to create a customised line style for the axis. A solid line is set with thickness 5, one square end and one arrowed end. Line 230 draws a line from the bottom left of the screen (30,180) to the top left of the screen (30,10) to form the Y axis. Line 240 draws a line from the bottom left of the screen (30,180) to the bottom right of the screen (600,180) to form the X axis.

Lines 270-310 place separation markers and labels along the x axis. This segment of code is reproduced below:

```
260 rem PLACE MARKERS ON THE X AXIS
270 for X=30 to 510 step 40
280 draw X,180 to X,185
290 next X
300 locate 1,24
310 print " 0   JAN FEB MAR APR MAY JUN JUL
AUG SEP OCT NOV DEC";
```

The graph displays data for each month of the year so the x-axis must be divided in to twelve segments (one for each month). Each value will be forty pixels wide as this allows twelve divisions to fit along the length of the line.


```
350 for Y=180 to 30 step-15
360 draw 30,Y to 20,Y
370 locate 0, ytext(Y)
380 print MARK;
390 inc MARK
400 next Y
```

The y-axis is divided in to 10 segments each of 15 pixels. The loop starts at coordinate 180 (the bottom of the y-axis) and works its way up the line in segments of 15 until it reaches coordinate 30. Line 360 draws a small line (10 pixels long) at each marker position. When constructing the x-axis we placed all the markers and then went back and placed the labels. On the y-axis the labels are placed as part of the *for..next* loop. Line 370 locates the text cursor at position $x=0/y=\text{position of the marker}$. Notice how the *ytext* function is used to convert the current y graphics coordinate to a text coordinate. The text cursor is therefore placed directly to the side of the marker and the label is printed, which in this case, is simply a number which is maintained by the variable MARK.

(4) Lines 430 - 490 plot the graph data. Line 430 sets the ink colour to green. Line 440 sets the starting position of the graph by plotting a point at coordinates 30,180 which is the zero value of the graph. Line 450 sets a *for..next* loop with a value of 1 to 12 to represent each month of the year. Line 460 and 470 calculate the graphics coordinates for the current graph data.

```
460 X=30+(COUNT*40)
470 Y=180-(INFO(COUNT)*15)
```

Line 460 calculates the X coordinate (the position across the graph). We know that the x-axis starts at coordinate $x=30$ and that it is separated in to segments of 40 pixels. Therefore we multiply the value of variable COUNT by 40 and add 30 to the result.

Line 470 calculates the y-coordinate (the position up the graph). We know that the y-axis starts at coordinate $y=180$ and that it is separated in to segments of 15. Therefore, we multiply the value of the data by 15 and subtract the result from 180.

Line 480 draws a line from the previous data coordinates to the current coordinates. When the graph is complete, line 500 waits for the user to press a key and line 510 terminates the program. The data for the graph is held at line 540. Try changing the values to see the effect that this has on the graph. Note that the program can only plot whole numbers in the range 1-10 but you may like to experiment changing the program to accept larger values and fractions, etc.

BAR CHARTS

The nice thing (or not quite so nice when trying to find errors!) with programming is that minor changes of the code can produce quite dramatic changes in the programs operation. For example, the previous program can be changed from producing line charts to bar charts by simply amending one line of code as shown below.

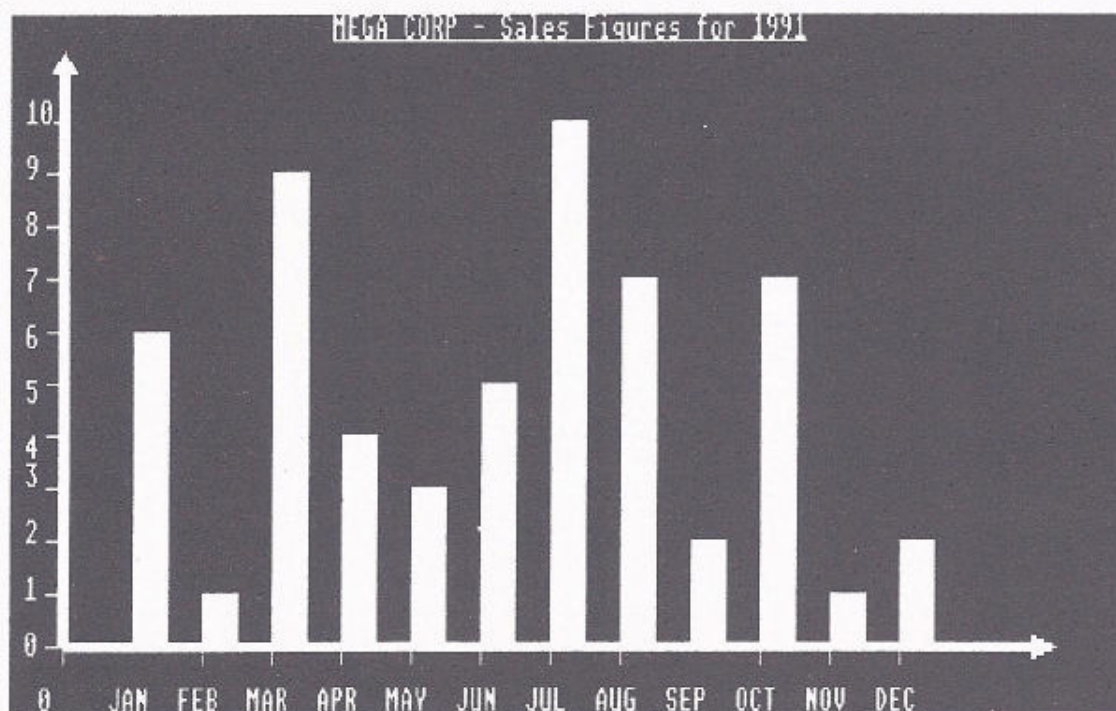
Enter the following:

```
480 bar X,Y to X + 20, 180
```

Run the program and it will now produce a bar chart as shown over the page.

These programs could be modified to suit any particular application by changing the data information and the scale/labelling of the axis. To make it really flexible we could obtain the data using *input* commands and then calculate the scale, etc of the axis.

Well that is enough of lines and charts. We shall now move on and



look at some more graphics commands.

DRAWING FILLED CIRCLES

In the same manner as boxes, STOS allows outlined and filled circles to be produced. It actually goes one step further than this by allowing segments of circles, ellipses and complete pie charts to be produced. Let's start by looking at simple circles.

Load the following:

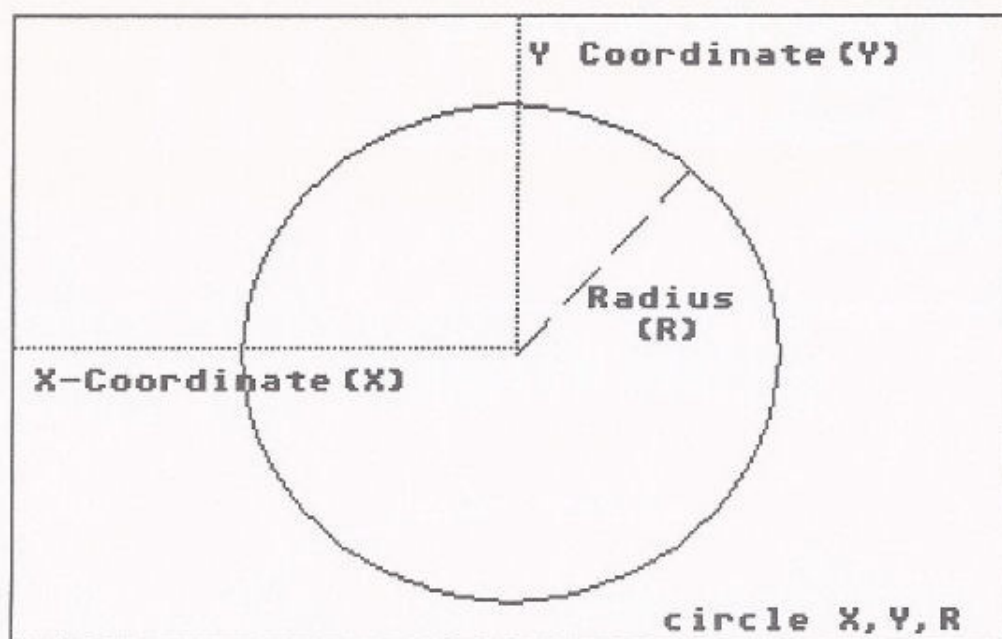
```
10 rem FILLED CIRCLES
20 rem PROGRAM = A:\GRAPHICS\CIRCLE
30 :
40 rem SET SCREEN RESOLUTION
50 key off : mode 0
60 :
70 rem DRAW CIRCLES
80 ink 12
```

```
90 circle 100, 100, 80
100 ink 13
110 circle 200, 150, 30
```

Run the program and you will see two, solid, circles displayed.

The format of the *circle* command is shown below:

`circle X, Y, R`



where X,Y are the coordinates of the centre of the circle and R is the radius of the circle. The radius of a circle is the distance from the centre to the outside edge. The circle is displayed in the current ink colour.

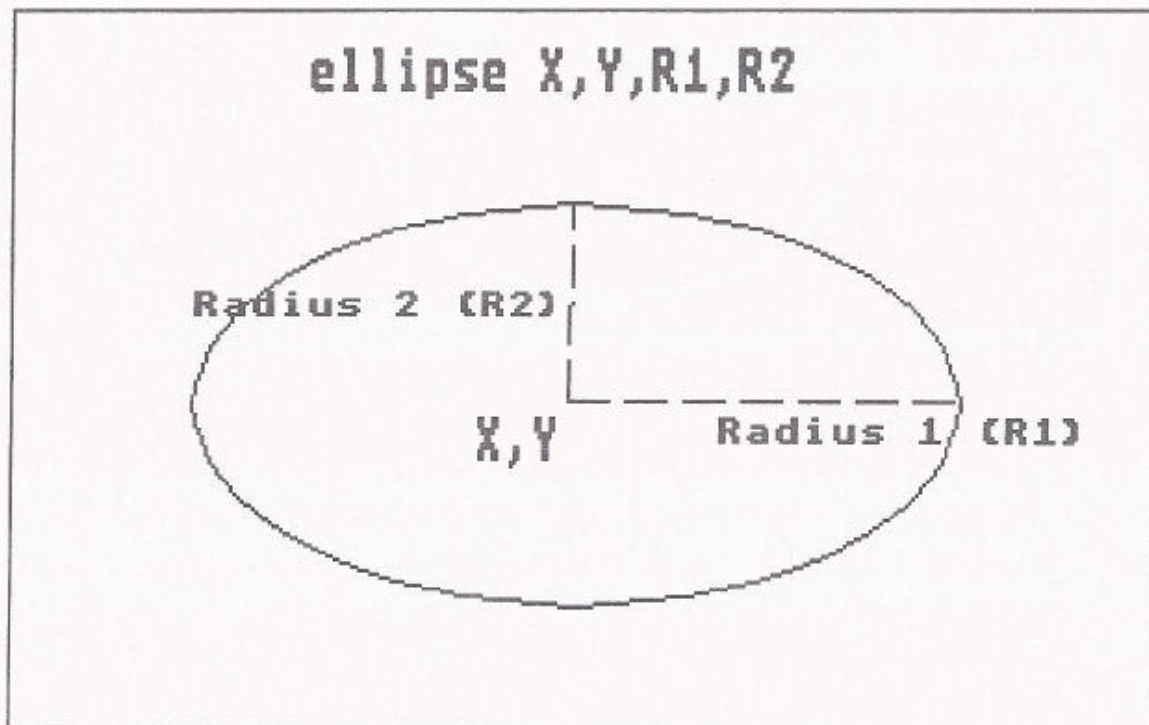
DRAWING FILLED ELLIPSES

Another function, very similar to *circle*, is *ellipse*.

Load the following:

```
10 rem FILLED ELLIPSES
20 rem PROGRAM = A:\GRAPHICS\ELLIPSE
30 :
40 rem SET SCREEN RESOLUTION
50 key off : mode 0
60 :
70 rem DRAW ELLIPSES
80 ink 14
90 ellipse 150, 50, 125, 40
100 ink 9
110 ellipse 150, 150, 30, 40
```

Run the program.



The format of the *ellipse* command is shown below:

`ellipse X, Y, R1, R2`

where X,Y are the graphic coordinates of the centre of the ellipse, R1 is the radius across to the left or right edge and R2 is the radius from the centre to the top or bottom edge.

The previous two commands, *circle* and *ellipse* are fine for solid shapes but suppose the shapes are required outlined. STOS offers four more commands for drawing outlined circles, ellipses and arcs.

CIRCULAR ARCS

Load the following:

```
10 rem THE ARC COMMAND
20 rem PROGRAM = A:\GRAPHICS\ARC
30 :
40 rem SET SCREEN RESOLUTION
50 key off : mode 0
60 :
70 rem DRAW AN ARC
80 arc 50,50,40,0,900
90 :
100 rem DRAW A SEMI CIRCLE
110 arc 50,100,40,0,1800
120 :
130 rem DRAW COMPLETE CIRCLE
140 arc 50,150,40,0,3600
```

Run the program.



Three arcs are drawn to produce a quarter of a circle, a semi circle and a complete circle as shown.

The format of the *arc* command is shown below:

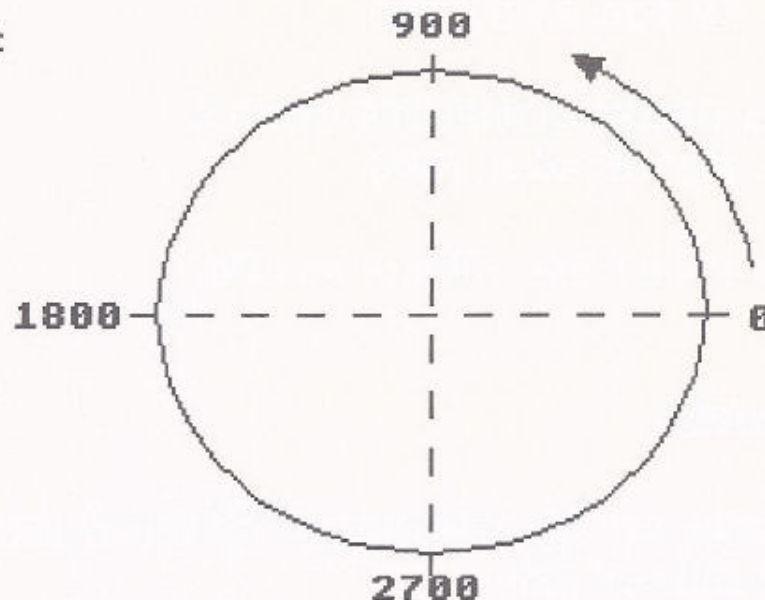
`arc X, Y, R, START, END`

where X,Y are the coordinates of the centre of the circle, R is the radius of the circle, START is the angle where the arc should start and END is the angle where the arc should stop.

Angles are normally measured in degrees with a complete circle being 360 degrees, but STOS measures angles in tenths of degrees and hence a complete circle is 3600 degrees. The best way to understand angles is to consider a clock face. All measurements are taken from the three o'clock position with this being angle 0. Angles are measured in an anti-clockwise direction from this point as shown in the diagram on the next page.

Looking back at the program, line 80 draws an arc which is one quarter of a circle in size, starting at the 3 o'clock position and ending at the 12 o'clock position (refer to the angle chart). Line 110 draws a semi-circle arc from position 3 o'clock to position 9 o'clock. Line 140 specifies a starting angle of 0 and an ending angle of 3600. This indicates a complete circle and hence a complete outline circle is produced. STOS does not offer a separate command for outline circles

ANGLES FOR USE WITH
ARC
EARC
PIE



and hence *arc* must always be used - remember that filled circles can be produced using the *circle* command.

We can also produce filled segments using the *pie* function. The format of the *pie* function is exactly the same as that for *arc*, and as the name suggests, it is ideally suited to the production of pie charts. We can alter the last program to display filled segments rather than just arcs by replacing the *arc* functions with *pie* functions.

Load the following:

```
10 rem THE PIE COMMAND
20 rem PROGRAM = A:\GRAPHICS\PIE
30 :
40 rem SET SCREEN RESOLUTION
50 key off : mode 0
60 :
70 rem DRAW AN ARC
80 pie 50,50,40,0,900
90 :
```



```
100 rem DRAW A SEMI CIRCLE
110 pie 50,100,40,0,1800
120 :
130 rem DRAW COMPLETE CIRCLE
140 pie 50,150,40,0,3600
```

Run the program and the arcs will now be solid.

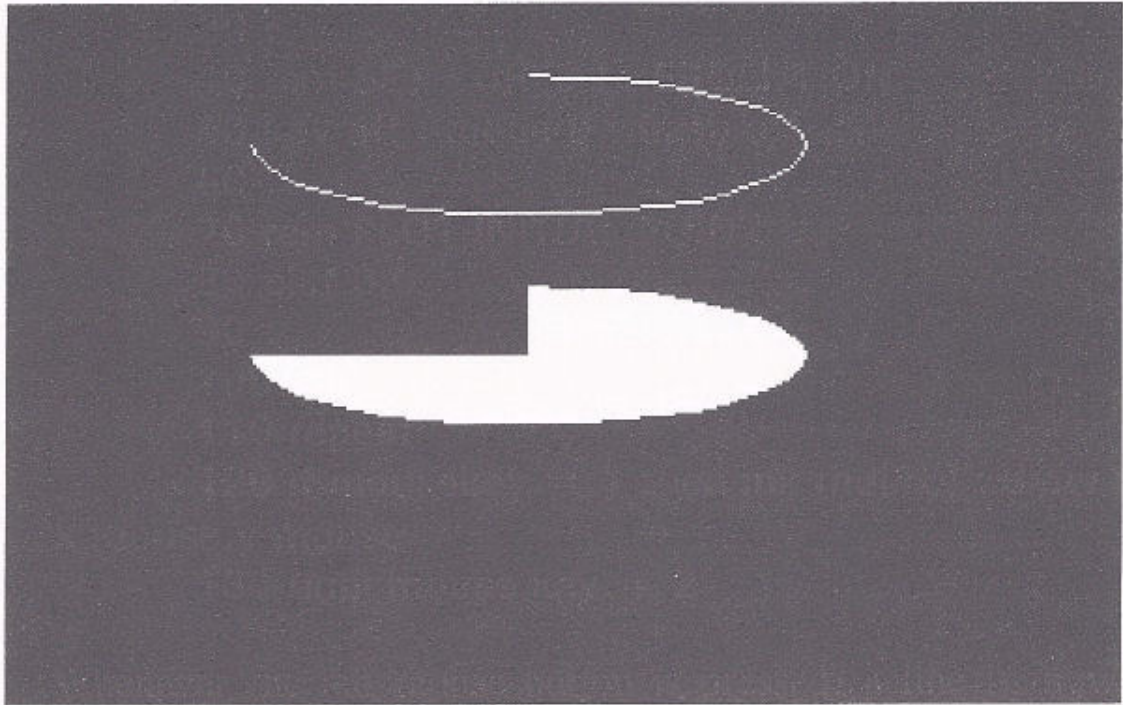
ELLIPTICAL ARCS

In addition to outline and solid circular arcs, STOS can also produce outline and solid elliptical arcs.

Load the following:

```
10 rem EARC AND EPIE
20 rem PROGRAM = A:\GRAPHICS\EARC
30 :
40 rem SET SCREEN RESOLUTION
50 key off : mode 0
60 :
70 rem DRAW OUTLINE ARC
80 earc 150, 40, 80, 20, 1800, 900
90 :
100 rem DRAW FILLED ARC
110 epie 150, 100, 80, 20, 1800, 900
```

Run the program and it will produce the following:



The following program uses the *pie* command to produce a pie chart representing the amount of used and free space on a floppy disk.

Load the following:

```
10 rem DISK SPACE PIE CHART
20 rem PROGRAM = A:\GRAPHICS\DFREE
30 :
40 rem SET SCREEN RES
50 key off : mode 0 : hide on
60 colour 1,$700 : colour 3,$7
70 :
80 rem CHECK FREE DISK SPACE
90 D=dfree/200
100 :
110 rem DRAW PIE CHART
120 ink 1 : pie 100,110,60,0,D
130 ink 3 : pie 100,110,60,D,3600
140 :
150 rem LABEL PIE CHART
```



```
160 pen 1 : locate 0,0 : print "DISK FREE"
```

```
170 pen 3 : locate 0,1 : print "DISK USED"
```

Run the program and it will draw the pie chart.

If you have a single sided disk drive rather than a double sided disk drive, change line 90 to the following:

```
90 D=dfree/100
```

The *dfree* command checks the amount of free disk space on the current disk. We shall take a closer look at this command in a later chapter.

PAINTING THE SCREEN

We have already seen how to produce filled shapes but these have been filled with the current ink colour giving the effect of a solid object.

STOS allows shapes and areas of the screen to be filled or painted with various styles. There are thirty six different styles available comprising solid, dotted, lined and user-defined varieties. The paint pattern is set using the *set paint* command, the format of which is illustrated below:

```
set paint STYLE, PATTERN, BORDER
```

STYLE can range from 0 - 4 as detailed below:

<u>STYLE</u>	<u>EFFECT</u>
0	Do not paint.
1	Paint using current ink colour (solid).
2	Paint using one of 24 dotted patterns.

- 3 Paint using one of 12 lined patterns.
- 4 Paint using user-defined pattern.

PATTERN indicates the required pattern in the range 1-24 or 1-12 depending on the paint style selected.

BORDER can be set to either 0 (zero) or 1. When set to one, a border is drawn around the painted area using the current ink colour.

We shall now produce a program to display all of the dotted and lined paint styles.

Load the following:

```
10 rem DISPLAY PAINT STYLES
20 rem PROGRAM = A:\GRAPHICS\PAINT
30 :
40 key off : mode 0 : curs off : hide on : ink 10
50 :
60 rem DISPLAY DOTTED PAINT STYLES
70 for PATTERN = 1 to 24
80 set paint 2,PATTERN,1
90 bar 100,50 to 220,100
100 wait key
110 next PATTERN
120 :
130 rem DISPLAY LINED PAINT STYLES
140 for PATTERN = 1 to 12
150 set paint 3,PATTERN,1
160 bar 100,50 to 220,100
170 wait key
180 next PATTERN
```

Run the program and press any key on the keyboard to cycle through the various dotted and lined styles available.

Notice that the program uses the *bar* command to produce a painted or filled bar. We have already encountered this command in a previous section where it was used to produce solid bars on a bar chart. The paint style defaults to style zero which means that all of the commands *bar*, *rbar*, *circle*, *ellipse*, *pie*, *epie* and *polygon* will produce a solid object in the current ink colour unless a *set paint* command is used to change the default setting.

STOS, as mentioned previously, also allows user-defined fill patterns but it is unlikely that you will need to use these as the thirty six styles provided by STOS are quite adequate for the majority of applications. If you do want to design your own patterns then refer to the STOS Users Guide.

There is one final command that completes the graphics section of this course. The *paint* command allows any shape or area of the screen to be filled using the current ink and paint style. The format of the command is shown below:

paint X,Y

where X and Y represent the graphic coordinates of a point within the area to be filled.

As a final example, the following program combines a number of the graphics commands to draw a picture of a house.

Load the following:

```
10 rem DRAW HOUSE
20 rem PROGRAM = A:\GRAPHICS\HOUSE
30 :
40 key off : mode 0 : curs off : hide on
50 :
60 rem DRAW THE GRASS
```

```
70 ink 11
80 bar 0,180 to 319,199
90 :
100 rem DRAW THE MAIN HOUSE
110 ink 15
120 bar 50,80 to 250,180
130 :
140 rem DRAW THE DOOR AND WINDOWS
150 ink 10
160 bar 140,140 to 165,180
170 bar 70,100 to 90,120
180 bar 70,140 to 90,160
190 bar 210,100 to 230,120
200 bar 210,140 to 230,160
210 :
220 rem DRAW THE SUN
230 ink 12
240 circle 30,30,20
250 :
260 rem DRAW THE ROOF
270 ink 5
280 set paint 2,9,1
290 polygon 45,80 to 75,50 to 225,50 to 255,80
300 bar 200,50 to 220,20
```

The graphics commands offered by STOS are ideally suited to the production of simple pictures that might be found in educational type programs.

ZONES

Previously we covered the mouse as a means of input and saw how the *x mouse* and *y mouse* functions can be used to determine the position of the mouse pointer on the screen. We can take this one step

further by defining areas of the screen, known as zones, which can be tested for the presence of the mouse pointer or a sprite. We shall cover sprites in a later chapter so for now we are only interested in the mouse pointer. A maximum of 128 zones can be defined and tested, but by way of an example, we shall define three.

Load the following:

```
10 rem DEMONSTRATE ZONES
20 rem PROGRAM = A:\GRAPHICS\ZONES
30 :
40 rem SET SCREEN
50 key off : mode 0
60 :
70 rem DRAW BOXES
80 box 10,10 to 60,60
90 box 110,10 to 160,60
100 box 210,10 to 260,60
110 :
120 rem DEFINE ZONES
130 set zone 1,10,10 to 60,60
140 set zone 2,110,10 to 160,60
150 set zone 3,210,10 to 260,60
160 :
170 rem MONITOR THE ZONES
180 repeat
190 Z = zone(0)
200 locate 14,10
210 print "ZONE = ";Z
220 wait vbl
230 until mouse key = 1
```

Run the program and you will see three boxes displayed at the top of the screen. Move the mouse pointer in to each of the boxes and the zone number will be displayed. The program thus detects when the

mouse pointer enters one of the boxes and can be terminated by pressing the left mouse button.

The program works by setting a zone to cover the area of each box. The zones are monitored, and when the mouse pointer is detected, the zone number is printed to the screen. Let's look at the program.

Line 50 sets the screen resolution to low and lines 80-100 draw the three boxes.

Lines 130-150 define the three zones using the *set zone* command. The format of *set zone* is shown below:

```
set zone N,X,Y to X1,Y1
```

N is a number that is assigned to identify the zone, X/Y are the graphics coordinates of the top left hand corner and X1/Y1 are the graphics coordinates of the opposite corner.

Notice that the coordinates of the zones are exactly the same as those of the three boxes so that each zone covers the box.

Once defined, the zones can be checked. Lines 180-230 form a *repeat..until* loop which continues until the left mouse button is pressed. Line 190 contains the *zone* command. The number in brackets indicates the number of the sprite, which in the case of the mouse pointer, is zero. The command checks all of the defined zones and returns the number of the zone in which the mouse pointer is detected. If the mouse pointer is outside of all the zones a value of zero is returned.

Lines 200-210 display the current zone on the screen and line 220 contains a *wait vbl* command to stop the screen flickering.

Zones are ideal for the production of menu screens where the user can

select options using the mouse. Later in this course we shall develop and art program and shall use zones to produce such a menu system.

Chapter 14

The Screen

I doubt if you have reached this chapter without already having had a sneaky look! The thought of screens, sprites and games have probably proved too much as it is one of the most interesting and exciting areas available to the programmer. In the past, games tended to be developed using machine code as this was the only medium that guaranteed the speed and flexibility required, but things quickly changed with the introduction of STOS which has made a mighty name for itself as the true alternative to machine code. STOS offers a vast array of facilities for the games programmer which we shall cover in the next few chapters. Areas of the screen can be cut and pasted, screens can be flipped to produce animation, sprites can be zapped around the screen at high speed, collisions can be detected for super shoot-em-ups, animation sequences can be applied to sprites plus much much more.

PICTURE FORMATS

We saw in a previous chapter how the screen is divided in to pixels

which can be switched on and off to produce lines, boxes and pictures. The screen handling side of STOS is somewhat more advanced than this allowing a whole multitude of graphical effects to be achieved. Before moving on to these advanced facilities we need to understand how STOS controls screens and how sprites are displayed.

Pictures can be stored on disk in the same way as normal programs and data. We have seen previously how different types of file are assigned different file extensions and picture files are no exception. There are a number of picture formats available to the ST but the two most common of these are DEGAS and NEOCHROME. Most art packages offer DEGAS and many also offer NEOCHROME. Picture files are identified by the following file extensions:

- .PI1 = Degas format low resolution picture.
- .PI2 = Degas format medium resolution picture.
- .PI3 = Degas format high resolution picture.
- .NEO = Neochrome format which is a low resolution format.

STOS basic allows us to load these picture files and use them within our programs. This is a very very powerful facility indeed as the pictures can be produced using external art packages that offer far more facilities than the STOS graphic commands. Another interesting area regarding pictures is that of scanning. Various Companies now offer 'scanning' hardware and software that allow pictures to be 'grabbed' from magazines, books and video tape. The process of scanning converts the pictures in to a series of dots which can then be saved as a picture file on disk. This technology produces photographic images as the following programs illustrate.

Load the following:

```
10 DISPLAY A COLOUR PICTURE
20 PROGRAM = A:\SCREEN\PIC1
```

```
30 :  
40 key off : mode 0 : flash off  
50 :  
60 load "a:\pictures\pictures.pi1"  
70 :  
80 wait key
```

Run the program and a colourful picture will be displayed.

This picture was produced using an art package. The *wait key* command is included in line 80 so that you can view the picture - press any key to end the program.

The picture file has the extension .PI1 which indicates that the file is stored in Degas format and is designed for low resolution. Remember that low resolution allows sixteen colours.

Now load the following:

```
10 DISPLAY A DIGITISED PICTURE  
20 PROGRAM = A:\SCREEN\PIC2  
30 :  
40 key off : mode 0 : flash off  
50 :  
60 load "a:\pictures\village.pi1"  
70 :  
80 wait key
```

Run the program.

This picture has been scanned and looks more like a photograph than a computer generated picture. Whilst scanned pictures are pretty impressive, most games look better with nice colourful pictures which can be produced using any of the many art packages available for the ST. If you do not already have an art package, do not worry as we

shall be writing one in chapter 16 when we have covered all the techniques and concepts required to undertake such a project.

Once a picture has been loaded, STOS offers a range of facilities that can produce some really stunning effects. We shall look at these shortly but for now let us see how sprites fit in to the picture.

WHAT IS A SPRITE ?

A sprite is a small graphical unit that can move around 'on top of' or above another image that is displayed on the screen without effecting that image. For example, consider a shoot-em-up game where the aliens zap around the screen as you try and shoot them. The aliens are sprites that move around on a background screen. Another example of a sprite is the Atari mouse pointer. As you move the mouse around the mouse pointer follows your movement on the screen, floating on top of any information that is currently displayed.

CHANGING THE MOUSE POINTER SPRITE

To prove that the mouse pointer is a sprite, we can change the design of the sprite to say a space craft as illustrated below.

Load the following:

```
10 rem CHANGE THE MOUSE POINTER
20 rem PROGRAM = A:\SCREEN\MOUSE
30 :
40 rem SET SCREEN
50 mode 0
60 :
70 rem LOAD THE SPRITES IN TO MEMORY
80 load "a:\screen\invaders.mbk"
```

```
90 :  
100 rem CHANGE THE MOUSE POINTER  
110 change mouse 14
```

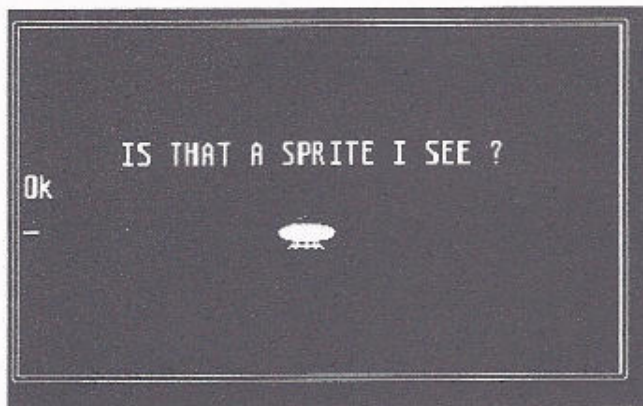
Run the program and you will see that the mouse pointer has changed to a space craft.

Look closely at line 80 and you will notice that the filename has the extension MBK. This indicates that this is a memory bank file and hence the data will be loaded in to a memory bank. STOS reserves memory bank 1 for sprites and hence the data is automatically loaded in to memory bank 1. Line 110 uses the *change mouse* command which changes the style of the mouse pointer to the specified sprite. The value following the command specifies the number of the sprite to be assigned. Numbers 1, 2 and 3 are reserved by STOS for arrow, pointing hand and clock face with numbers 4 and above indicating an external sprite. In this example we have specified number 14 which indicates sprite number 11 from the sprite bank ($14-3=11$).

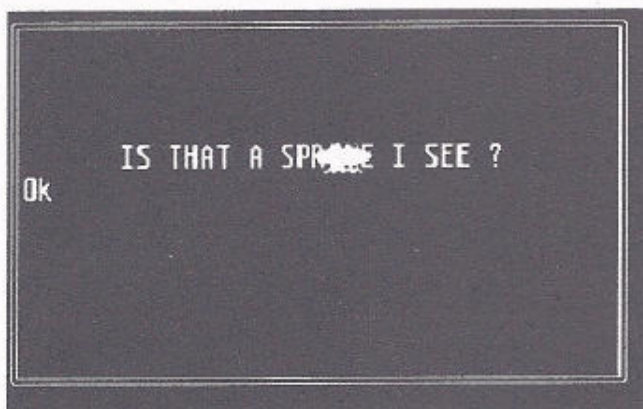
Do not worry too much about the sprite file and the memory banks at this stage as all will be revealed later.

THE SCREEN

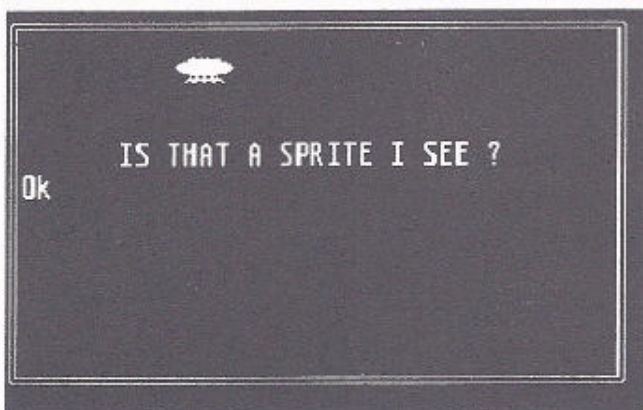
Although the ST is a clever beast, only one piece of information can be displayed on the screen at any one time. When a sprite is displayed it may cover information that was already displayed at that position. When the sprite is moved, STOS has to restore the information that was previously covered by the sprite. For example, consider a space ship sprite moving up the screen. Information covered by the space ship must be restored as it moves on. The diagram over the page illustrates this.



The sprite (in this case a space ship) is below the printed message.



The sprite moves up and covers the word SPRITE. Whilst parts of the word are still visible, the sprite has actually been written over the top of the word.



The sprite now moves to the top of the screen and the word SPRITE is visible again. The clever part of the operation is the way in which the original information (e.g. the word SPRITE) is restored when the sprite moves on. STOS keeps a copy of the background (or

the original information) which it re-copies to the main screen each time that the sprite moves. Whenever information is covered by a sprite, the information is still held in the background so that it may be recopied back to the original position when the sprite moves or is removed from the screen.

STOS therefore maintains two separate screens in memory - the PHYSICAL screen and the BACKground screen. Consider the

background screen to be lying underneath the physical screen. When a sprite is displayed it is only displayed on the physical screen and the background screen still contains the original information that is hidden by the sprite. The background screen is then used to redraw the original area of the screen when the sprite moves. In the last example we used the *load* command to load a picture file as shown below:

```
load "a:\pictures\village.pi1"
```

This loads the picture to both the PHYSICal and BACKground screens but we can specify that the picture file be loaded to the PHYSICal screen only.

Enter the following:

```
mode 0  
flash off  
load "a:\pictures.pi1",physic
```

and move the mouse around.

The picture is displayed, but when the mouse is moved around (remember that the mouse pointer is a sprite), you will see that the picture is 'wiped' allowing the original background screen to show through. The word PHYSIC has been added to the end of the *load* command above to specify the screen to which the picture should be directed. This means that the picture is displayed on the PHYSICal screen but not the BACKground screen. The BACKground screen still contains old information, and hence when the mouse pointer is moved, STOS copies the relevant parts of the background screen to the PHYSICal screen resulting in the picture information being overwritten. Another point to note is that the colour palette has not changed to match the picture. STOS cannot change the colour palette because the information on the BACKground screen will still use the original palette.

Enter the following:

```
mode 0  
load "a:\pictures\pictures.pi1"
```

If you move the mouse around now, the picture is not affected. This is because the picture has been loaded to both the physical and background screens, and hence when the mouse is moved, the uncovered areas are correctly restored from the background screen.

Enter the following:

```
back = physic
```

and move the mouse around the screen. Notice now, that as the mouse is moved, all of the mouse pointers remain on the screen. The BACKground screen has been set to equal the PHYSICAL screen and hence all information displayed on the PHYSICAL screen is also displayed on the background screen.

Many people find this concept of PHYSICAL and BACKground screens complicated so if you have not fully understood everything have a quick look back and try the examples again.

To make things a little more interesting STOS also offers a third screen known as the LOGICAL screen. The LOGICAL and PHYSICAL screens are normally one of the same - that is that all operations operate on both and hence no separation or separate consideration is required. There are times though when a separate screen can be very handy. For example, suppose we had a program that consisted of a main menu with lots and lots of options. Rather than redrawing the screen each time required, we could draw it once on the logical screen and simply switch it with the physical screen each time that the menu was required. This technique of screen switching, or 'flipping' as it is known, is used by a lot of games programmers to produce

animation. An image can be drawn on the LOGICAL screen which is hidden from view, and when the drawing is complete, the screen can be swapped with the PHYSICAL screen. The PHYSICAL screen is now hidden and can be used to develop the next image which can once again be flipped.

Let us look at a simple example to demonstrate this idea. We shall load two separate pictures, one to the PHYSICAL screen and one to the LOGICAL screen and then flip them.

Load the following:

```
10 rem SCREEN FLIPPING
20 rem PROGRAM: A:\SCREEN\FLIP
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off : hide on
60 :
70 rem LOAD PICTURE
80 load "a:\pictures\village.pi1"
90 :
100 rem HIDE LOGICAL SCREEN
110 logic = back
120 load "a:\pictures\vilage2.pi1"
130 :
140 rem WAIT FOR USER TO PRESS A KEY
150 wait key
160 :
170 rem FLIP THE LOGICAL AND PHYSICAL SCREENS
180 wait vbl : screen swap
190 :
200 rem GO BACK FOR ANOTHER GO
210 goto 140
```

Line 80 loads the first picture. At this stage the PHYSICAL and

LOGICAL screens are one of the same so the picture is effectively loaded to the PHYSICAL, LOGICAL and BACKground screens. Notice that, when a picture is loaded, its palette is also loaded so that the screen is displayed in the correct colours.

Line 110 sets the LOGICAL screen equal to the BACKground screen. This means that any operation on the logical screen will be underneath or at the back of the picture currently displayed and thus hidden from view. Line 120 loads a second picture in to the logical screen. Notice the word "logic" which has been added at the end of the *load* command. Remember that, if this is omitted, the picture will be loaded to all screens and displayed immediately. Both images are now in memory and line 150 waits for the user to press a key.

The *screen swap* command in line 180 carries out the switch. This simply swaps the PHYSICAL and LOGICAL screens. Notice the *wait vbl* (wait for vertical blank) command also on line 180 which delays the screen swap operation until the next vertical blank and thus ensures a smooth screen update. It is good practice to include *wait vbl* whenever you initiate a screen swap.

Moving on, line 210 sends execution back to line 140 where the process begins again. Therefore the screens will be flipped each time that a key is pressed.

When flipping the pictures take a close look at the colours of the second image. The second picture has been displayed using the same palette as the first. When loading an image in to a hidden screen, STOS cannot change the colour palette as this would have an immediate effect on the displayed image as well. The easiest way to overcome this problem is to use memory banks.

MEMORY BANKS

We have already seen that memory banks can store a variety of information and this includes complete pictures. Any of the fifteen banks can be reserved to hold picture information on a permanent or temporary basis. When designing a program we need to decide which pictures to make permanent (stored along with the program) and which pictures to make temporary. Let us look at a couple of examples to illustrate the difference between temporary and permanent memory banks.

Load the following:

```
10 rem SLIDE SHOW
20 rem PROGRAM = A:\SCREEN\SLIDESH
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off : hide off
60 :
70 rem RESERVE MEMORY BANKS FOR SCREENS
80 reserve as screen 12
90 reserve as screen 13
100 reserve as screen 14
110 reserve as screen 15
120 :
130 rem LOAD PICTURES IN TO MEM BANKS
140 load "a:\pictures\pictures\cat.pi1",12
150 load "a:\pictures\pictures\car1.pi1",13
160 load "a:\pictures\pictures\horse.pi1",14
170 load "a:\pictures\pictures\heron.pi1",15
180 :
190 rem DISPLAY PICTURES
200 for COUNT = 12 to 15
210 get palette (COUNT)
220 screen copy COUNT to physic
```


230 wait key

240 next COUNT

This program loads 4 different pictures and then displays each one in turn. When a picture is displayed, press any key to move on to the next.

Before information can be loaded in to a memory bank we have to reserve the bank and define the required status. Lines 80-110 reserve four memory banks with *screen* status. Therefore banks 12-15 are now temporary, screen banks. Lines 140-170 load a picture in to each of the reserved memory banks. The numbers at the end of the *load* commands indicate the memory bank that the picture should be sent to. Line 200 sets up a *for..next* loop to display the four pictures. Line 210 introduces another new command, *get palette*, and the format of this is shown below:

get palette (N)

where N specifies the number of the memory bank.

This command changes the current colour palette to suit the picture stored in the specified bank. The *get palette* command is normally issued before copying a picture from a memory bank to the PHYSICAL screen so that the picture is displayed in the correct colours. Line 220 copies the picture from the appropriate memory bank to the PHYSICAL screen using the *screen copy* command and hence the picture is immediately displayed. The *screen copy* command is very powerful and we shall look at this in detail shortly. Line 230 waits for a key to be pressed and line 240 continues the loop.

In the last example the memory banks were reserved and the screens loaded within the program. Memory banks reserved as 'screen' are temporary and must therefore be loaded each time that the program is run. An alternative method is to reserve the banks as

DATASCREENS which are permanent banks and are thus automatically saved along with the program.

Enter the following:

```
reserve as datascreen 11  
load "a:\pictures\cat.pi1",11
```

The screen is now in the memory bank and can be displayed on the screen.

Enter the following:

```
mode 0  
get palette (11)  
screen copy 11 to physic
```

Notice that, unlike character sets, we do not have to specify the size of the memory bank.

To conclude this section let us just recap on the two types of screen. SCREEN memory banks are temporary and must be loaded each time the program is run. DATASCREEN memory banks are permanent and hence do not need to be loaded as part of the program. You can use direct commands to load these screens and they will always be available. For example, suppose we wanted a title screen to be permanently stored. Before entering the main program we could enter:

```
reserve as datascreen 14  
load "title.pi1",14
```

The screen is now available and will be stored as part of our program. No further reserving or loading is required within the program.

FADING PICTURES

In the last section we saw how to load pictures in to memory banks and switch them in and out of view. By copying the contents of a memory bank to the PHYSICAL screen, for example:

```
screen copy 11 to physic
```

the new picture is displayed immediately on the screen. This is a very powerful facility but you have probably seen commercial games where the screens gradually fade in and out rather than just appearing. This effect really can make a program look professional and exciting and can be achieved quite easily with STOS using the *fade* command.

The *fade* command can be used in one of three ways so let's look at some examples.

(1) The first method is to slowly blend all of the colours to black and hence achieve a gradual fading effect. The format of the *fade* command for this is shown below:

```
fade SPEED
```

where SPEED indicates the fade speed (more on this below).

Load the following:

```
10 rem FADE THE SCREEN
20 rem PROGRAM: A:\SCREEN\FADE1
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off
60 :
70 rem LOAD A PICTURE
80 load "a:\pictures\pictures\xmas.pi1"
```

```
90 :  
100 rem WAIT FOR USER TO PRESS A KEY  
110 wait key  
120 :  
130 rem FADE THE SCREEN  
140 fade 10 : wait 10*7
```

Run the program. When the picture is displayed press any key.

The above program loads a picture and waits for the user to press a key. When a key is pressed the picture is faded from the screen. Line 80 loads the picture file, and as no screen or memory bank have been specified, it is loaded directly to both the PHYSICal and BACKground screens with the colour palette being adjusted to match the picture.

The *fade* command in line 140 is followed by the number 10. This number indicates the speed of the fade and specifies the number of vertical blanks that should occur between each change of the colour palette. Line 150 contains the *wait* command which causes the computer to stop and wait for a specified amount of time. The amount of time, as usual, is specified in 50ths of a second.

The *fade* command works under interrupt and hence the program continues execution whilst the fade continues in the background. If the next line of our program was to print new information to the screen this would interfere with the fade and hence would not work properly. The *wait* command therefore halts the program until the fade has completely finished. The period of time required for the complete fade is calculated as follows:

wait FADE SPEED * 7

the speed specified with the *fade* command is multiplied by 7.

Try changing the speed of the fade to see the different effects that can

be achieved. To speed the fade up try changing line 140 as shown below:

```
140 fade 2 : wait 2*7
```

or to slow the fade down try:

```
140 fade 50 : wait 50*7
```

(2) The second form of the *fade* command allows the current colour palette to be slowly blended to the new palette of a screen stored in a specific memory bank. The format of the command is as follows:

```
fade SPEED to MEMBANK
```

where SPEED specifies the speed of the fade and MEMBANK specifies the memory bank containing the new palette.

Load the following:

```
10 rem FADE A PICTURE TO MEM BANK PALETTE
20 rem PROGRAM: A:\SCREEN\FADE2
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off
60 :
70 rem LOAD PICTURE IN TO MEMORY BANK
80 reserve as screen 14
90 load "a:\pictures\pictures.pi1",14
100 :
110 rem LOAD MAIN PICTURE
120 load "a:\pictures\car1.pi1"
130 :
140 rem WAIT FOR USER TO PRESS KEY
```

```
150 wait key
160 :
170 rem FADE TO MEMORY BANK 14 PALETTE
180 fade 10 to 14 : wait 10*7
```

Lines 80-90 reserve a memory bank and load a picture in to it. Line 120 loads a picture directly on to the screen. When the user presses a key, the *fade* command at line 180 slowly blends the colour palette to the new colours in memory bank 14. The original picture does not look as good but it does demonstrate the effect.

(3) The third format of the command is the most powerful as it allows any colour to be faded or blended to any other colour. The format of the command is as follows:

fade SPEED, C1, C2, C3, etc.

where SPEED is the speed of the fade, C1 is the new colour (specified as \$RGB) for colour index 1, C2 is the new colour for colour index 2, etc.

This is a very nice effect for displaying title screens and messages.

Load the following:

```
10 rem FADE INDIVIDUAL COLOURS
20 rem PROGRAM = A:\SCREEN\FADE3
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off
60 :
70 rem SET COLOUR INDEX 1 TO BLACK
80 colour 1,$0
90 :
```



```
100 rem OPEN WINDOW-NO BORDER AND LARGE  
CHARACTERS
```

```
110 windopen 1,0,0,39,12,0,3
```

```
120 :
```

```
130 rem HIDE CURSOR AND MOUSE POINTER
```

```
140 hide on : curs off
```

```
150 :
```

```
160 rem PRINT TEXT
```

```
170 centre "Featuring the amazing"
```

```
180 print
```

```
190 centre "STOS"
```

```
200 print
```

```
210 centre "FADE COMMAND"
```

```
220 :
```

```
230 rem FADE COLOUR INDEX 1 TO PINK
```

```
240 fade 6,,$704 : wait 6*7
```

```
250 wait key
```

```
260 :
```

```
270 rem FADE BACK TO BLACK AGAIN
```

```
280 fade 6,,$0 : wait 6*7
```

The above program displays a large message on the screen which appears (or fades) out of the background. Pen index 1 is initially set to black, and because the message is printed on the black background, it is invisible. The *fade* command at line 240 fades colour index 1 through to pink and the message appears on the screen. Line 250 waits for a key press and line 280 fades the message back to black again. Remember the *wait* commands that follow all the *fade* commands. Just as a reminder take a look at line 110. A window is opened with border style 0 (zero) and character set 3. Border style zero results in no border being displayed and character set 3 is the high resolution character set which appears as double height on the low resolution screen.

Load the following:

```
10 rem FADE ON A SINGLE LINE
20 rem PROGRAM = A:\SCREEN\FADE4
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off
60 :
70 rem SET COLOUR INDEX 1 TO BLACK
80 colour 1,$0
90 rem OPEN WINDOW
100 windopen 1,0,0,39,12,0,3
110 :
120 rem HIDE CURSOR AND MOUSE POINTER
130 hide on : curs off
140 :
150 centre "Welcome to....."
160 fade 10,,,$700 : wait 10*7
170 fade 10,,,$0 : wait 10*7
180 :
190 clw
200 centre "The Beginners Guide to"
210 fade 10,,,$70 : wait 10*7
220 fade 10,,,$0 : wait 10*7
230 :
240 clw
250 centre "STOS BASIC"
260 fade 10,,,$7 : wait 10*7
270 fade 10,,,$0 : wait 10*7
280 :
290 clw
300 centre "from MT Software"
310 fade 10,,,$143 : wait 10*7
```

This example fades in and out different messages on a single line with

each message displayed in a different colour.

Load the following:

```
10 rem FADE MULTIPLE LINES
20 rem PROGRAM = A:\SCREEN\FADE5
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off
60 palette $0,$0,$0,$0
70 :
80 rem OPEN WINDOW - NO BORDER - CHARACTER
SET 3
90 windopen 1,0,0,39,12,0,3
100 :
110 rem HIDE MOUSE POINTER AND CURSOR
120 hide on : curs off
130 :
140 rem PRINT MESSAGES
150 pen 1
160 centre "The Beginners Guide to"
170 pen 2
180 print : print
190 centre "STOS BASIC"
200 pen 3
210 print : print
220 centre "from MT SOFTWARE"
230 :
240 rem FADE MESSAGES
250 fade 14,,,$421 : wait 14*7
260 fade 14,,,,$572 : wait 14*7
270 fade 14,,,,,$257 : wait 14*7
280 :
290 rem FADE THE BACKGROUND
300 fade 20,$7 : wait 20*7
```

310 fade 20,\$700 : wait 20*7

This final example prints a three line message down the screen and then fades each line out of the background in a different colour. Finally the background is also faded.

We could also use the *fade* command within the slide show program that we saw previously to fade the pictures in and out.

Load the following:

```
10 rem SLIDE SHOW USING FADE COMMAND
20 rem PROGRAM: A:\SCREEN\SLIDESH2
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off : hide on
60 fade 1 : wait 1*7
70 :
80 rem RESERVE MEMORY BANKS
90 reserve as screen 12
100 reserve as screen 13
110 reserve as screen 14
120 :
130 rem LOAD PICTURE FILES IN TO BANKS
140 load "a:\pictures\car1.pi1",12
150 load "a:\pictures\car2.pi1",13
160 load "a:\pictures\car3.pi1",14
170 :
180 rem DISPLAY THE PICTURES
190 for COUNT = 12 to 14
200 screen copy COUNT to physic
210 fade 10 to COUNT : wait 10*7
220 wait 50
230 fade 10 : wait 10*7
240 next COUNT
```


Notice the *fade* command in line 60. This fades all of the colours to black instantly. We could use the *palette* command but this requires more typing:

```
fade 1
```

produces the same result as:

```
palette $0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0
```

Try experimenting with the *fade* command as it allows a vast range of effects to be achieved which can add real sparkle to your programs.

Well, if you thought the *fade* command was good then just wait until you see what the *appear* command has to offer.

MAKING PICTURES APPEAR

Load the following:

```
10 rem MAKE A PICTURE APPEAR
20 rem PROGRAM: A:\SCREEN\APPEAR1
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off : hide on
60 :
70 rem LOAD PICTURE IN TO MEMORY BANK 14
80 reserve as screen 14
90 load "a:\pictures\pictures.pi1",14
100 :
110 rem MAKE PICTURE APPEAR
120 get palette (14)
130 appear 14,30
140 :
```

```
150 rem PRESS A KEY TO END THE PROGRAM
160 wait key
```

The *appear* command makes a picture appear using one of seventy nine, yes seventy nine, different effects.

The format of the *appear* command is:

```
appear X,Y
```

Where X represents the screen to appear and can be a memory bank, PHYSIC, LOGIC, etc. The Y parameter is optional and specifies which one of the seventy nine effects you require. If this is omitted, the effect is chosen by STOS at random. Effects 1-72 result in the new picture completely replacing the existing display whilst effects 73-79 leave the new picture slightly different.

Let's modify the last program so that we can try different effects.

Load the following:

```
10 rem MAKE PICTURE APPEAR
20 rem PROGRAM: A:\SCREEN\APPEAR2
30 :
40 rem SET SCREEN
50 key off : mode 0 : hide on
60 :
70 rem LOAD PICTURE IN TO MEM BANK 14
80 reserve as screen 14
90 load "a:\pictures\pictures.pi1",14
100 get palette (14)
110 :
120 rem WHAT APPEAR STYLE DO YOU REQUIRE
130 cls
140 input "Enter value for appear (1-79)";STYLE
```



```
150 :  
160 rem MAKE PICTURE APPEAR  
170 flash off  
180 appear 14,STYLE  
190 :  
200 rem WAIT FOR USER TO PRESS A KEY  
210 wait key  
220 :  
230 rem GO BACK FOR ANOTHER GO  
240 goto 130
```

Try experimenting with different values and make a note of ones that you especially like so that you can use them at a later date in your own programs.

In the previous sections we looked at two slide show programs. The first simply displayed the pictures one after the other whilst the second used the *fade* command to fade the pictures in and out. To conclude the series of slide show programs we shall now produce a third program that uses the *appear* command.

Load the following:

```
10 rem SLIDE SHOW USING APPEAR COMMAND  
20 rem PROGRAM: A:\SCREEN\SLIDESH3  
30 :  
40 rem SET SCREEN  
50 key off : mode 0 : flash off : hide on  
60 :  
70 rem RESERVE MEMORY BANKS  
80 reserve as screen 12  
90 reserve as screen 13  
100 reserve as screen 14  
110 :  
120 rem LOAD PICTURE FILES IN TO BANKS
```

```
130 load "a:\pictures\car1.pi1",12
140 load "a:\pictures\car2.pi1",13
150 load "a:\pictures\car3.pi1",14
160 get palette (12)
170 :
180 rem DISPLAY THE PICTURES
190 for COUNT = 12 to 14
200 appear COUNT
210 wait key
220 next COUNT
```

Run the program and press any key to see each of the three pictures appear using a random appear effect.

ZOOMING & REDUCING

We have already seen some of the very impressive screen operations that STOS is able to perform. The ability to fade screens and make new screens appear makes our programs look very professional, but so far we have always considered the screen as a whole. STOS offers a range of commands that allow us to manipulate small sections of the screen and we shall start with the *zoom* and *reduce* commands. The *zoom* command allows a specified section of the screen to be enlarged to any other specified size. The matching command *reduce* allows a specified section of the screen to be reduced to any specified size. Let's take a look these in operation.

This first program loads a picture on the screen which contains a further sixteen, smaller pictures. When the user presses a key, one of the smaller pictures (the mouse) is zoomed to fill the entire screen.

Load the following:

```
10 rem ZOOMING A SECTION OF THE SCREEN
```



```
20 rem PROGRAM = A:\SCREEN\ZOOM
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off
60 :
70 rem LOAD PICTURE
80 load "a:\pictures\pictures.pi1"
90 :
100 rem WAIT FOR USER TO PRESS KEY
110 wait key
120 :
130 rem ZOOM THE SCREEN
140 zoom 2,52,77,97 to 0,0,319,199
```

The format of the *zoom* command is as follows:

`zoom SCREEN1, X, Y, X1, Y1 to SCREEN2, X2, Y2, X3, Y3`

where SCREEN 1 and SCREEN 2 indicate the source and destination screens, X, Y, X1, Y1 indicate the coordinates of the area of the source screen that is to be enlarged and X2, Y2, X3, Y3 indicate the coordinates of the position in to which the image will be expanded. SCREEN 1 and SCREEN 2 may refer to memory banks and may also be omitted if you want the operation performed on the current screen.

Using the optional screen parameters we can load a picture in to a memory bank and then zoom a section on to the physical screen.

Load the following:

```
10 rem ZOOM SCREEN USING MEM BANK
20 rem PROGRAM = A:\SCREEN\ZOOM2
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off
```

```
60 :  
70 rem LOAD PICTURE  
80 reserve as screen 15  
90 load "a:\pictures\car2.pi1",15  
100 get palette(15)  
110 :  
120 rem ZOOM PICTURE  
130 zoom 15,90,65,220,130 to physic,0,0,319,199
```

Run the program and you will see a picture of a car. Line 130 recovers the picture from the memory bank, zooms it and displays it on the physical screen. The advantage of this technique is that the user only sees the final zoomed image and does not see the original picture as per the last example.

The ZOOM facility is ideally suited to the development of art packages that allow areas to be enlarged so that fine and detailed editing may be carried out. It can also be used for creating large titles as the following example illustrates.

Load the following:

```
10 rem ZOOM A MESSAGE  
20 rem PROGRAM = A:\SCREEN\MESSAGE  
30 :  
40 rem SET SCREEN  
50 key off : mode 0  
60 :  
70 rem DISPLAY MESSAGE  
80 print "HIGH SCORES"  
90 :  
100 rem ZOOM  
110 zoom 0, 0, xgraphic(11), ygraphic(1) to  
0,0,310,75
```


The program works fine but the initial small message may be seen by the user before it is zoomed, and this can make a program look rather unprofessional. The answer is to carry out the operation in a way that is hidden from the user.

Load the following:

```
10 rem ZOOM MESSAGE USING LOGICAL SCREEN
20 rem PROGRAM = A:\SCREEN\MESSAGE2
30 :
40 rem SET SCREEN
50 key off : mode 0
60 :
70 rem HIDE THE LOGICAL SCREEN
80 logic = back
90 :
100 rem DISPLAY MESSAGE
110 print "HIGH SCORES"
120 :
130 rem ZOOM MESSAGE ON TO PHYSICAL SCREEN
140 zoom logic,0,0,xgraphic(1),ygraphic(11) to physic
0,0,310,75
```

The program works by printing the required message in the background, out of sight of the user. This message is then zoomed and displayed on the PHYSICAL screen. The large message therefore appears as if it were simply printed on the screen.

The required text is printed to the LOGICAL screen where it is hidden from the user. line 80 sets the LOGICAL screen to the BACKGROUND and line 110 prints the message.

Line 140 contains the *zoom* command but notice this time that we have also specified the screens. The command therefore zooms the image and copies it from the LOGICAL screen to the PHYSICAL

screen. This ability to zoom across different screens really does allow some powerful effects to be achieved.

Well that has covered the *zoom* command but what about the opposite command *reduce*. *Reduce* offers the opposite effect to *zoom*, allowing images to be reduced in size and placed at a new position on the screen. Let's look at an example.

Load the following:

```
10 rem REDUCE A SECTION OF THE SCREEN
20 rem PROGRAM = A:\SCREEN\REDUCE
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off
60 :
70 rem LOAD "PICTURE
80 load "A:\PICTURES\XMAS.PI1"
90 :
100 rem REDUCE THE PICTURE
110 reduce to 0,0,160,100
```

A picture is loaded and then reduced to the top left hand corner of the screen. Look carefully at the *reduce* command in line 100. The format of the command is as follows:

reduce SCREEN1 to SCREEN2, X, Y, X1, Y1

where SCREEN1 is a screen containing the image to be reduced, SCREEN 2 is the destination screen where the re-sized image is to be displayed, X/Y specify the position of the new image and X1/Y1 specify the size of the new image.

In the above example we have not specified any screens so the operation is carried out on the current screen. The coordinates after

the *reduce* command are 0,0,160,100. This specifies that the top left hand corner of the image is to be placed at coordinates 0,0 and that the new image is to be 160 pixels across by 100 pixels down. This is one quarter of the main screen and hence the image is reduced to a quarter of its original size.

Try experimenting with various size coordinates and you will notice that the image can be 'stretched' and 'squashed' considerably changing the original image. For example, try changing line 110 to:

```
110 reduce to 0,0,300,100
```

Notice how the image has been expanded or stretched.

By using the optional screen parameters we can reduce an image before actually displaying it to the user.

Load the following:

```
10 rem REDUCE A HIDDEN SECTION OF SCREEN
20 rem PROGRAM = A:\SCREEN\REDUCE2
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off
60 :
70 rem LOAD PICTURE IN TO MEM BANK
80 reserve as screen 14
90 load "a:\pictures\cat.pi1",14
100 get palette (14)
110 :
120 rem REDUCE PICTURE AND DISPLAY
130 reduce 14 to physic,0,0,160,100
```

The image is loaded in to memory bank 14 where it is invisible to the user. The *reduce* command at line 120 then reduces the image and

copies it to the PHYSICAL screen where it becomes visible to the user.

We shall now look at the most powerful of all the screen manipulation commands - *screencopy*. After this we shall have a break and write a complete game based on the subjects learnt in this chapter.

COPYING SECTIONS OF THE SCREEN

The *screencopy* command really is one of the most powerful commands offered by the STOS Basic programming language. It allows any section of a screen to be copied to a new position on the original, or a different screen. By copying a number of different images on top of each other we can achieve some quite complicated animations.

Let's start by extending the last example program. In this program we loaded a screen, reduced its size and then displayed it in the top left hand corner of the screen. Suppose that we wanted to display the same picture in all four corners of the screen. When the picture has been displayed once, we can simply copy it to the other three corners.

Load the following:

```
10 rem REDUCING THE SCREEN
20 rem PROGRAM = A:\SCREEN\REDUCE3
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off
60 :
70 rem LOAD PICTURE TO MEM BANK
80 reserve as screen 14
90 load "a:\pictures\cat.pi1",14
100 get palette (14)
110 :
```



```
120 rem REDUCE PICTURE AND DISPLAY
130 reduce 14 to physic,0,0,160,100
140 screen copy physic,0,0,160,100 to physic,161,0
150 screen copy physic,0,0,160,100 to physic,0,101
160 screen copy physic,0,0,160,100 to
physic,161,101
```

USING SCREENCOPY TO PRODUCE ANIMATION

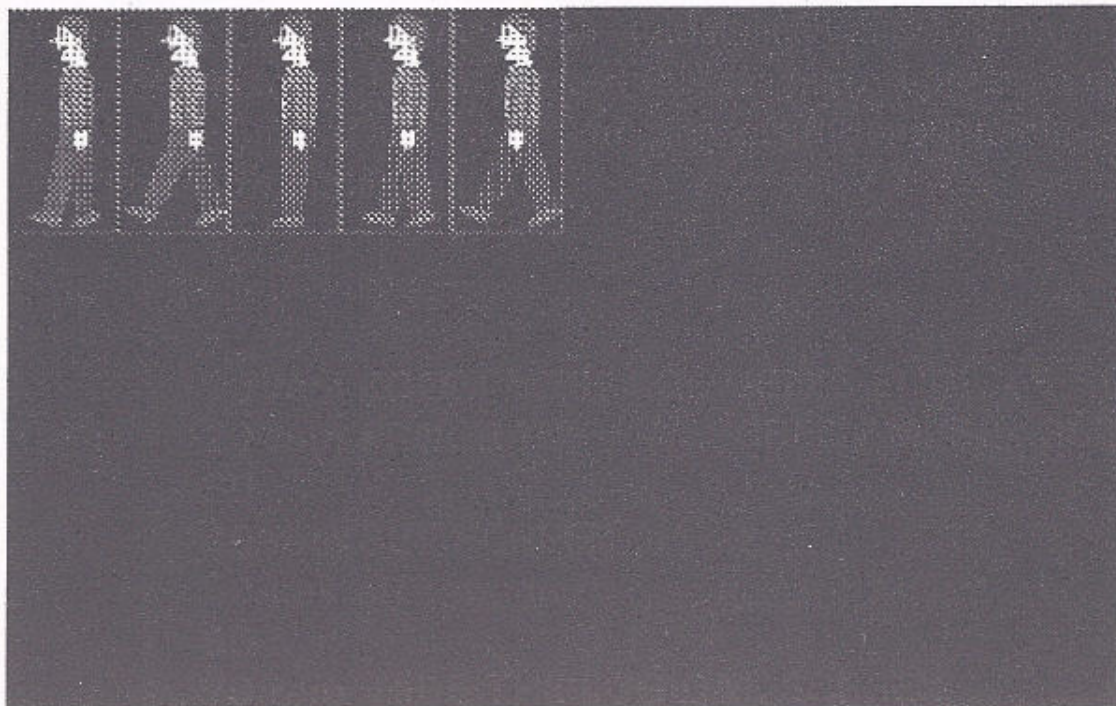
The next program serves to demonstrate how *screencopy* can be used to produce animation. It also shows how the command can copy across completely different screens. To produce the effect of animation we have to copy a number of pictures or 'frames' on top of each other. Our example is going to make a man walk on the spot in the centre of the screen and to start we need to produce a picture containing all of the frames. Disk 1 that accompanies this course already has such a picture as shown on the next page.

The picture consists of 5 separate images which, when continuously copied on top of each other in a cyclic fashion, will generate the effect of the man walking. Each frame is 32 pixels wide (X axis) by 64 pixels high (Y axis). If you consider the frames to be numbered from 1-5 across the page, the coordinates required by the *screencopy* command are as follows:

```
IMAGE 1: 0,0,32,64
IMAGE 2: 32,0,64,64
IMAGE 3: 64,0,96,64
IMAGE 4: 96,0,128,64
IMAGE 5: 128,0,160,64
```

The next program carries out this copying of images.

Load the following:



```
10 rem ANIMATION USING SCREENCOPY
20 rem PROGRAM = A:\SCREEN\ANIMATE
30 :
40 rem SET SCREEN
50 key off : mode 0 : curs off : hide on
60 :
70 rem LOAD SCREEN IN TO MEMORY BANK
80 reserve as screen 15
90 load "a:\pictures\walk.pi1",15
100 get palette (15)
110 :
120 rem ANIMATE THE IMAGES
130 wait vbl
140 screen copy 15,64,0,96,64 to physic,128,68
150 wait 7
160 wait vbl
170 screen copy 15,0,0,32,64 to physic,128,68
180 wait 7
190 wait vbl
200 screen copy 15,32,0,64,64 to physic,128,68
```



```
210 wait 7
220 wait vbl
230 screen copy 15,0,0,32,64 to physic,128,68
240 wait 7
250 wait vbl
260 screen copy 15,64,0,96,64 to physic,128,68
270 wait 7
280 wait vbl
290 screen copy 15,96,0,128,64 to physic,128,68
300 wait 7
310 wait vbl
320 screen copy 15,128,0,160,64 to physic,128,68
330 wait 7
340 wait vbl
350 screen copy 15,96,0,128,64 to physic,128,68
360 wait 7
370 goto 130
```

Lines 70-100 reserve a memory bank, load the picture and change the current colour palette to suit the picture.

Lines 120 to 360 carry out the animation by copying each of the images from bank 15 to the PHYSICAL screen. Run the program and compare the displayed images to those on the screen containing all of the images. Considering the individual images to be numbered from left to right, the sequence of frames is 3-1-2-1-3-4-5-4.

SCREEN SCROLLING

In the next chapter we shall develop a complete game using the animation techniques just discussed, but before moving on, there is one more area to cover - screen scrolling. STOS allows the complete screen or a part of the screen to be scrolled. You may have seen one of the many demonstration disks that are available for the Atari

computer. These are produced by computer enthusiasts to show off the features of the computer and often include a line of text that roles or scrolls across the screen. Another example of scrolling is the credits that are seen rolling up the television screen at the end of films, etc. It does not matter whether you want to produce scrolling text or complete scrolling pictures, they can all be achieved using STOS.

We shall now look at a number of examples that will illustrate the use of the scroll commands and the effects that can be achieved using same.

VERTICAL SCREEN SCROLLING

Load the following:

```
10 rem SCROLL SCREEN UP VERTICALLY
20 rem PROGRAM = A:\SCREEN\SCROLL1
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off : curs off : hide on
60 :
70 rem LOAD PICTURE TO MEM BANK 15
80 reserve as screen 15
90 load "a:\pictures\cat.pi1",15
100 get palette (15)
110 :
120 rem DEFINE SCROLLING
130 def scroll 1,0,0 to 320,200,0,-1
140 :
150 rem SCROLL THE SCREEN
160 for Y=0 to 199
170 screen copy 15,0,Y,320,Y+1 to physic,0,199
180 wait vbl : scroll 1
190 next Y
```


Run the program and you will see a picture of cats scroll up from the bottom of the screen.

We start by defining the type of scrolling that we require. We can specify the area of the screen to be scrolled, the number of pixels that the screen should be scrolled and whether the scrolling should be horizontal or vertical. These parameters are specified using the *def scroll* command, the format of which is shown below:

```
def scroll N, X, Y, X1, Y1, XSTEP, YSTEP
```

N indicates the number of the scrolling zone. A maximum of 16 zones or areas of the screen can be defined and these are identified by the number N that is assigned by the programmer.

X, Y, X1, and Y1 indicate the graphics coordinates of a rectangular area of the screen that we wish to scroll. X/Y indicate the top left corner and X1/Y1 indicate the bottom right corner.

XSTEP specifies the number of pixels by which the screen or segment of the screen should be scrolled in the horizontal direction. If this is a positive value the screen will be scrolled from left to right, and if it is a negative value the screen will be scrolled from right to left.

YSTEP specifies the number of pixels by which the screen or segment of the screen should be scrolled in the vertical direction. If this is a positive value the screen will be scrolled from top to bottom, and if this is a negative value the screen will be scrolled from bottom to top.

We can therefore produce horizontal scrolling by specifying a value for XSTEP, we can produce vertical scrolling by specifying a value of YSTEP and we can produce a combination of the two, by setting both XSTEP and YSTEP.

Getting back to the program, line 130 defines the scrolling and is reproduced below.

```
130 def scroll 1,0,0 to 320,200,0,-1
```

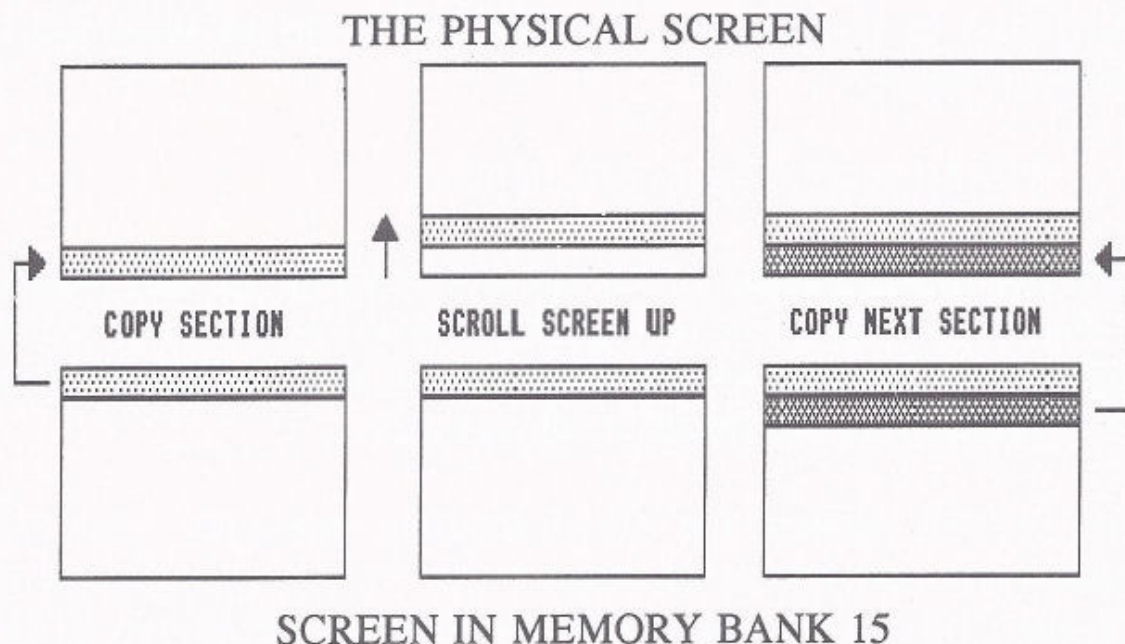
This defines scrolling zone 1. The coordinates 0,0 to 320,200 indicate the area of the screen to be scrolled, which in this case, is the complete low resolution screen. XSTEP is set to zero, YSTEP is set to -1 and this makes the screen scroll upwards.

Lines 150-190 perform the actual scrolling operation. Line 160 initiates a *for..next* loop in the range 0-199. This range covers all of the graphic pixels from the top of the screen to the bottom of the screen. Line 170 copies a section of the screen in memory bank 15 to the PHYSICAL screen and line 180 waits for the vertical blank before scrolling the screen using the *scroll* command. The number following the *scroll* command indicates the zone to be scrolled as specified by the *def scroll* command.

The trickiest part of the operation is the screen copy operation in line 170. This is made clearer in the diagram shown over the page.

A strip is copied from the top of the picture in memory bank 15 to the bottom of the PHYSICAL screen. The physical screen is then scrolled upwards to make way for the next line. The *for..next* loop causes this operation to repeat until the complete picture has been scrolled on to the PHYSICAL screen.

NOTE: The *screen copy* command could be used on its own to produce a similar scrolling facility as the program above. The only problem is that we would have to copy larger areas of the screen and this can lead to an inconsistent scrolling speed and a jerky display.



Load the following:

```

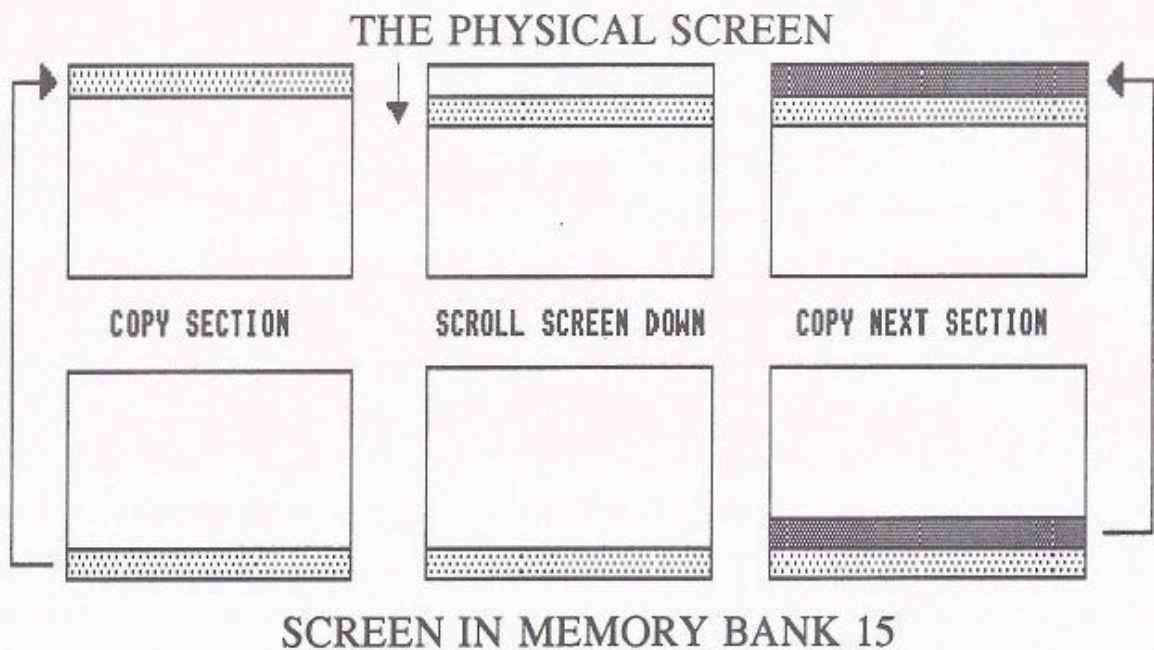
10 rem SCROLL SCREEN DOWN VERTICALLY
20 rem PROGRAM = A:\SCREEN\SCROLL2
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off : curs off : hide on
60 :
70 rem LOAD PICTURE TO MEM BANK 15
80 reserve as screen 15
90 load "a:\pictures\horse.pi1",15
100 get palette (15)
110 :
120 rem DEFINE SCROLLING
130 def scroll 1,0,0 to 320,200,0,1
140 :
150 rem SCROLL THE SCREEN
160 for Y=0 to 199
170 screen copy 15,0,200-Y,320,200 to physic,0,0
180 wait vbl : scroll 1

```

190 next Y

Run the program.

This time a picture scrolls down from the top of the screen. Notice in line 130 that a positive value has been specified for YSTEP and thus the screen scrolls downwards. The program uses the same principle as the previous program and is further illustrated in the diagram below:

**HORIZONTAL SCREEN SCROLLING**

Pictures can also be scrolled in from the left and right hand side of the screen

Load the following:

```
10 rem SCROLL SCREEN FROM RIGHT TO LEFT
20 rem PROGRAM = A:\SCREEN\SCROLL3
30 :
```



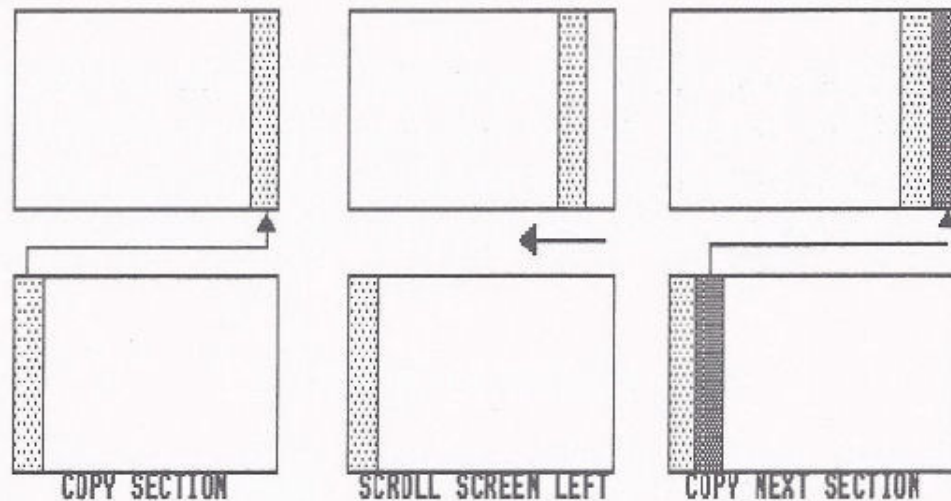
```
40 rem SET SCREEN
50 key off : mode 0 : flash off : curs off : hide on
60 :
70 rem LOAD PICTURE TO MEM BANK 15
80 reserve as screen 15
90 load "a:\pictures\cat.pi1",15
100 get palette (15)
110 :
120 rem DEFINE SCROLLING
130 def scroll 1,0,0 to 320,200,-16,0
140 :
150 rem SCROLL THE SCREEN
160 for X=0 to 319 step 16
170 wait vbl : scroll 1
180 screen copy 15,X,0,X+16,200 to physic,320,
-16,0
190 wait 5
200 next X
```

Run the program and you will see that the picture scrolls in from the right hand side of the screen. Notice that the scrolling is a lot jerkier this time. This is due to the fact the *screen copy* command always rounds down the X coordinate to the nearest sixteen and hence we have to scroll 16 pixel blocks rather than single pixel blocks that we used for vertical scrolling.

Look carefully at the scrolling definition in line 130. The XSTEP parameter has been specified as a negative number. The diagram over the page illustrates the way in which the sections of screen are copied and scrolled.

And finally we can scroll in from the left hand edge of the screen.

THE PHYSICAL SCREEN



SCREEN IN MEMORY BANK 15

Load the following:

```

10 rem SCROLL SCREEN FROM LEFT TO RIGHT
20 rem PROGRAM = A:\SCREEN\SCROLL4
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off : curs off : hide on
60 :
70 rem LOAD PICTURE TO MEM BANK 15
80 reserve as screen 15
90 load "a:\pictures\cat.pi1",15
100 get palette (15)
110 :
120 rem DEFINE SCROLLING
130 def scroll 1,0,0 to 320,200,16,0
140 :
150 rem SCROLL THE SCREEN
160 for X=0 to 319 step 16
170 wait vbl : scroll 1
180 screen copy 15,320-X,0,320,200 to physic,0,0

```



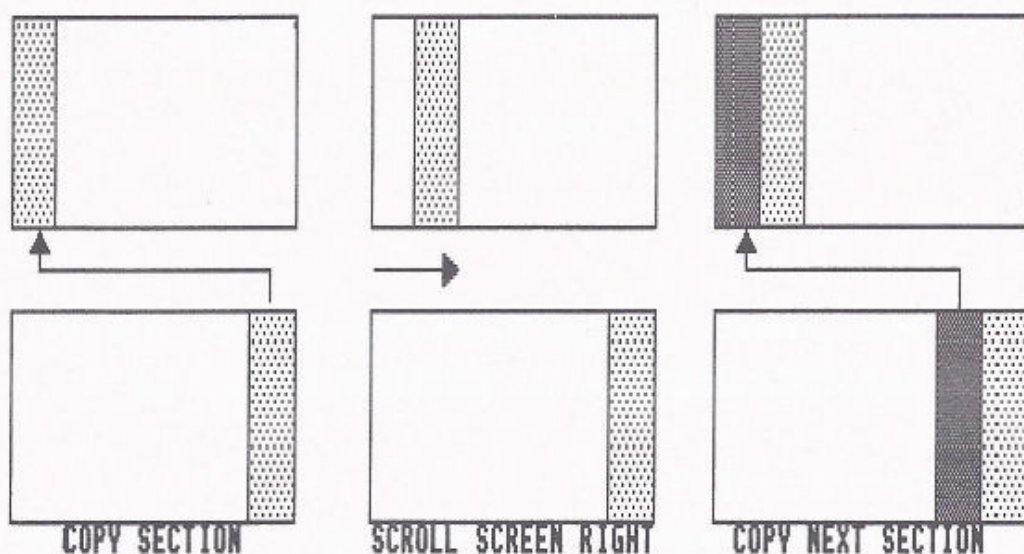
```

190 wait 5
200 next X

```

Run the program and you will see the picture scroll in from the left hand side of the screen. Notice this time that a positive value for XSTEP has been used within the scroll definition at line 130. To see how the *screen copy* command in line 180 works, refer to the diagram below:

THE PHYSICAL SCREEN



SCREEN IN MEMORY BANK 15

DIAGONAL SCROLLING

Scrolling directions can be freely mixed and can produce a variety of interesting effects. The next example combines horizontal and vertical scrolling.

Enter or load the following:

```

10 rem COMBINED SCROLLING
20 rem PROGRAM = A:\SCREEN\SCROLL5

```

```
30 :  
40 rem SET SCREEN  
50 key off : mode 0 : flash off : curs off : hide on  
60 :  
70 rem LOAD PICTURE IN TO MEM BANK 15  
80 reserve as screen 15  
90 load "a:\pictures\xmas.pi1",15  
100 get palette (15)  
110 :  
120 rem REDUCE TO TOP LEFT CORNER  
130 reduce 15 to physic,0,0,160,100  
140 :  
150 rem DEFINE SCROLLING  
160 def scroll 1,0,0 to 320,200,2,2  
170 :  
180 rem SCROLL THE PICTURE  
190 for A=1 to 35  
200 wait vbl : scroll 1  
210 next A
```

The program loads a picture, reduces it to the top left corner of the screen and then scrolls it down to the centre of the screen. Run the program to see the effect.

Lines 70-100 load the picture in to memory bank 15 and change the current palette to suit it. Note that nothing has been displayed on the physical screen yet and hence it is still blank.

Line 130 uses the *reduce* command to reduce the screen in memory bank 15 and copy the reduced section to the PHYSICAL screen where it appears in the top left hand corner.

Line 160 defines scrolling zone 1. Notice that both parameters XSTEP and YSTEP have been set to a value of 2. Therefore, when the scrolling is activated, the screen will scroll from left to right by 2

pixels and downwards by 2 pixels.

In a moment we shall look at methods for creating scrolling text lines but first we shall look at one more example which uses four separate scrolling zones to produce a fascinating effect.

Load the following:

```
10 rem BARREL SCROLL EFFECT
20 rem PROGRAM = A:\SCREEN\SCROLL6
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off : curs off : hide on
60 :
70 rem LOAD PICTURE TO MEM BANK 15
80 reserve as screen 15
90 load "a:\screen\pictures\pictures.pi1",15
100 get palette (15)
110 :
120 rem DEFINE SCROLLING
130 def scroll 1,0,30 to 320,106,0,-4
140 def scroll 2,0,20 to 320,33,0,-3
150 def scroll 3,0,14 to 320,22,0,-2
160 def scroll 4,0,11 to 320,16,0,-1
170 :
180 rem SCROLL THE SCREEN
190 for A=0 to 199 step 4
200 screen copy 15,0,A,320,A+4 to physic,0,102
210 scroll 4
220 scroll 3
230 scroll 2
240 scroll 1
250 :
260 wait 1
270 next A
```

```
280 :  
290 goto 190
```

Run the program and be amazed !

The program produces a barrel scroll - that is it appears to roll the picture around on the screen. It uses the same principle as the previous programs in that sections of the screen in memory bank 15 are copied to the PHYSICal screen. The difference here though is that four separate scroll operations are used. If you wish to trace the exact operation, draw the screen out on paper and work out where the scrolling zones lay.

To achieve the barrel effect the top of the picture is scrolled at a slower speed to the rest of the picture and the picture appears to compress - this is what produces the effect of rotation.

There are a vast range of effects that can be achieved with the *scroll* command so do not be afraid to experiment.

SCROLLING TEXT LINES

The STOS scroll commands can be used to produce scrolling text lines similar to those seen on demonstration disks or those seen on revolving signs that are often found in shop windows. Let us look at an example.

Load the following:

```
10 rem SCROLLING TEXT LINE  
20 rem PROGRAM = A:\SCREEN\SCROLL7  
30 :  
40 rem SET SCREEN  
50 key off : mode 0 : curs off : hide on
```



```
60 :
70 rem DEFINE SCROLLING AREA
80 Y=ygraphic(10) : Y1=ygraphic(11)
90 def scroll 1,0,Y to 320,Y1,-4,0
100 :
110 rem DEFINE TEXT TO SCROLL
120 T$="Scrolling messages are easily produced using
the STOS Basic SCROLL command..... The Beginners
Guide to STOS Basic "
130 :
140 rem SCROLL THE TEXT
150 for L=1 to len(T$)
160 locate 39,10
170 print mid$(T$,L,1)
180 wait vbl : scroll 1
190 wait vbl : scroll 1
200 next L
```

Run the program and you will see a message scroll across the centre of the screen.

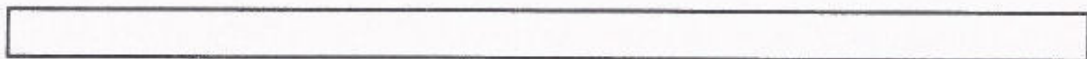
You can change the message by amending line 120.

Lines 90 defines the scroll zone and is reproduced below:

```
90 def scroll 1, 0, Y to 320, Y1,- 4, 0
```

This defines scrolling zone 1 as a box with the following coordinates:

0, ygraphic(10)

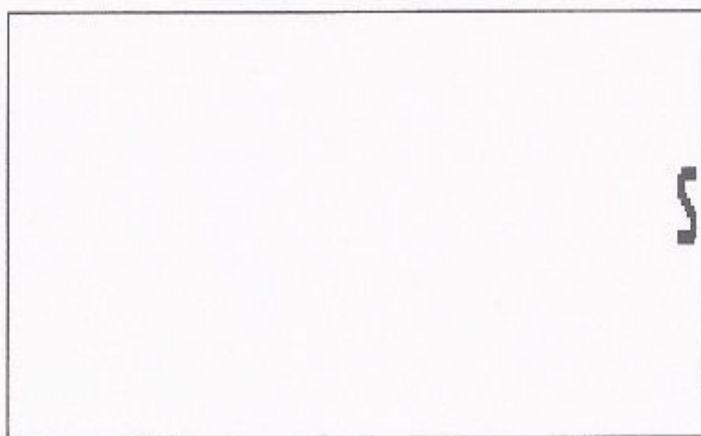


320, ygraphic(11)

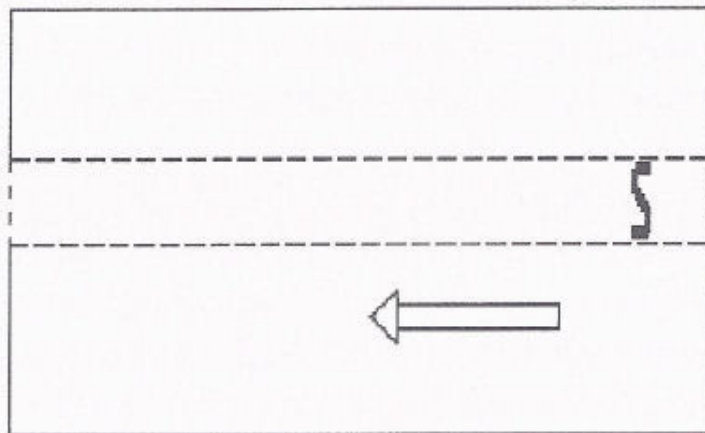
The program scrolls a single line of text at a position ten lines down

from the top of the screen. The *def scroll* command requires the graphics coordinates rather than the text coordinates and thus line 80 converts the text coordinates to graphic coordinates using the *ygraphic* function.

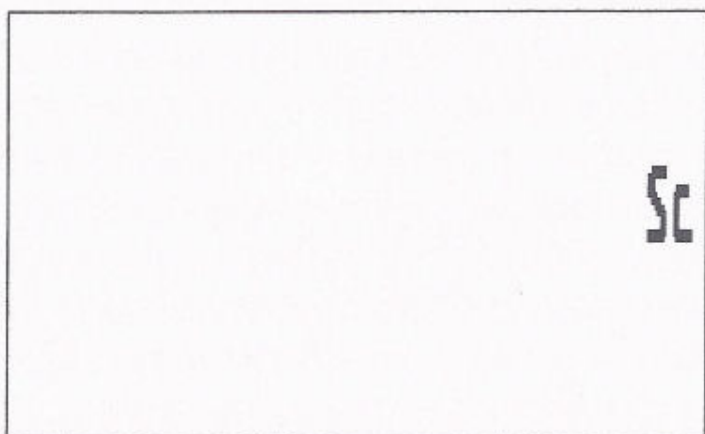
Lines 140-200 perform the scrolling. Line 150 initiates a loop which will continue for each character of the text string to be displayed. Notice how the *len* command is used to determine the length of the text string. Line 160 locates the text cursor at position 39,10 (which is at the end of line 10) and line 170 prints the next character from the text screen. Notice how the individual characters are extracted from the text string using the *mid\$* function. Line 180 waits for the vertical blank and then scrolls the screen. Line 190 then repeats this operation and line 200 sends the program back to display the next character, etc. You may recall from previous chapters that characters (such as the letter "A") are 8 pixels wide, so why do we scroll the screen twice by four pixels when we could simply scroll it once by eight pixels? The reason is that it produces a smoother scroll. As you start writing your own programs you will find various 'quirks' where theory is not always quite right in practice. The following shows the exact operation of the program:



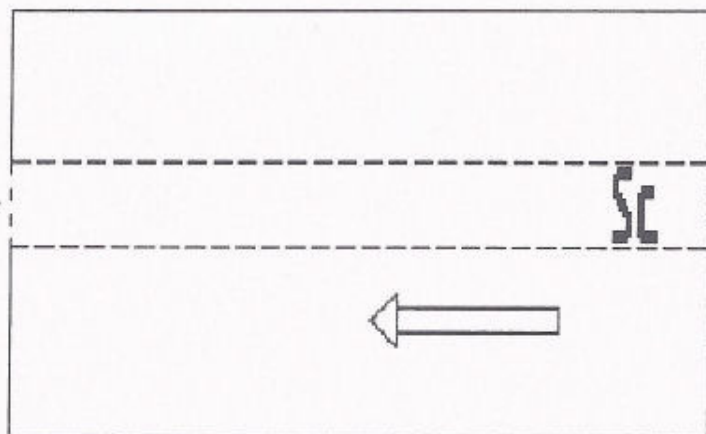
The first character "S" is placed on the screen.



The section of screen is scrolled as shown by the dotted outline and the letter "S" thus moves to the left.



The next letter is placed at the end of the line - in this case the letter "c"



The screen is then scrolled again and both letters now move to the left making room for the next letter.

This operation continues until all of the letters have been displayed.

The previous program can be expanded to produce yet another effect.

Load the following:

```
10 rem SCROLLING TEXT 2
20 rem PROGRAM = A:\SCREEN\SCROLL8
30 :
40 rem SET SCREEN
50 key off : mode 0 : curs off : hide on
60 :
70 rem DEFINE AREA OF SCREEN FOR SCROLLING
80 def scroll 1,0,ygraphic(10) to 320,ygraphic(11),-4,0
90 def scroll 2,0,0 to 320,200,0,-2
100 :
110 rem DEFINE TEXT TO SCROLL
120 dim T$(3)
130 T$(1) = "WELCOME TO....."
140 T$(2) = "The Beginners Guide to STOS Basic"
150 T$(3) = "from MT Software"
160 :
170 rem SCROLL THE TEXT
180 for A = 1 to 3
190 pen A + 10
200 :
210 rem SCROLL MESSAGE ACROSS THE SCREEN
220 for L = 1 to len(T$(A))
230 locate 39,10
240 print mid$(T$(A),L,1)
250 wait vbl : scroll 1
260 wait vbl : scroll 1
270 next L
280 wait 25
290 :
300 rem SCROLL MESSAGE UP THE SCREEN
310 for X = 1 to 10
```



```
320 wait vbl
330 scroll 2
340 next X
350 :
360 wait 25
370 next A
```

Run the program.

This time the program uses two scrolling zones which are defined in lines 80 and 90. Scrolling zone 1 is the same as the last example and scrolls a single line from right to left across the screen. The second scrolling zone scrolls the complete screen up by 2 pixels.

Lines 110-150 define the text that we wish to display. Three lines of text are displayed and these are assigned to three variables of an array T\$(3).

Lines 170-370 perform the scrolling operation. Lines 210-270 scroll a single line of text across the screen in the same manner as the previous example. Lines 300-340 then scroll the screen upwards making room for the next line of text. The program continues until all three lines have been displayed.

There is one final screen handling technique to look at before moving on and using these techniques to produce another game.

EXPANDING BOXES

We have recently seen how pictures can be directed to a particular screen, e.g. PHYSIC, BACK and LOGIC, but the graphics commands normally operate on both the PHYSICal and BACKground screens together. This means, that when we issue a *box* command, the box is drawn on both screens. This facility is known as *autoback* because the

commands are automatically written to the BACKground screen. There are times when we may not want to write to the background screen and thus the facility can be switched on and off using the *autoback on* and *autoback off* commands.

The following two programs demonstrate the effect of this.

Load the following:

```
(1) 10 rem AUTOBACK ON
     20 rem PROGRAM = A:\SCREEN\AUTO_ON
     30 :
     40 rem SET SCREEN
     50 key off : mode 0
     60 :
     70 autoback on
     80 box 10,10 to 100,50
     90 :
    100 wait key
    110 screen copy back to physic
```

Run the program and a box will be displayed. Press a key and the BACKground screen will be copied on to the PHYSICal screen. Notice that the screen remains unchanged.

In this example autoback is switched on and thus graphic commands will be written to both the PHYSICal and BACKground screens. The BACKground screen therefore contains the same information as the PHYSICal screen. Let's now try it with autoback switched off.

Load the following:

```
(2) 10 rem AUTOBACK OFF
     20 rem PROGRAM = A:\SCREEN\AUTO_OFF
     30 :
```



```
40 rem SET SCREEN
50 key off : mode 0
60 :
70 autoback off
80 box 10,10 to 100,50
90 :
100 wait key
110 screen copy back to physic
```

Run the program and a box will be displayed, but when a key is pressed, the box will disappear. This time autoback is switched off and so the box is only written to the PHYSICal screen - the BACKground screen does not contain the box. Line 110 copies the BACKground screen to the PHYSICal screen and thus the box disappears. We can use this technique to produce expanding boxes.

Expanding boxes are a feature found on the GEM desktop and in many art packages. The GEM desktop allows a group of files to be selected by expanding a rectangle across the icons and art programs allow shapes to be drawn using much the same method. The next program demonstrates the effect.

Load the following:

```
10 rem DRAW AN EXPANDING BOX
20 rem PROGRAM = A:\GRAPHICS\EXPAND
30 :
40 rem SET SCREEN
50 key off : mode 0
60 :
70 rem DRAW TO PHYSIC ONLY
80 auto back off
90 :
100 rem WAIT FOR USER TO PRESS MOUSE BUTTON
110 repeat : until mouse key = 1
```

```
120 :  
130 rem GET MOUSE COORDINATES  
140 X=x mouse : Y=y mouse  
150 X1=X : Y1=Y  
160 :  
170 rem HAS USER TO RELEASED MOUSE BUTTON  
180 repeat : until mouse key = 0  
190 :  
200 rem DRAW THE EXPANDING BOX  
210 while mouse key < > 1  
220 box X,Y to X1,Y1  
230 X1=x mouse : Y1=y mouse  
240 while X1=x mouse and Y1=y mouse and mouse  
key = 0 : wend  
250 screen copy back to physic  
260 wend  
270 :  
280 rem DRAW FINAL BOX  
290 auto back on  
300 box X,Y to X1,Y1
```

Run the program and click the left mouse button. Move the mouse pointer around and you will see that an expanding box is produced. Click the left mouse button again and the box will be permanently drawn on the screen at the current size. We shall use this technique within an art program in chapter 16 so let's see how it works.

Line 50 sets the screen.

Line 80 contains the *autoback off* command and causes subsequent graphic commands to be limited to the PHYSICal screen only.

Line 110 waits for the user to press the left mouse button. When the button is pressed, the current mouse pointer coordinates are assigned to variables X/Y and X1,Y1 and these form the corners of the box.

Line 180 checks that the user has released the mouse button.

Now we come to the interesting part. Lines 210-260 form a *while..wend* loop which perform the clever operation of drawing the expanding box. Line 220 draws a box using the coordinates X,Y and X1,Y1. Line 240 uses a loop which halts the program until the mouse pointer is moved or the left mouse button is pressed. When the mouse pointer moves or the mouse button is pressed, the program moves on to line 250.

Line 250 copies the BACKground screen to the PHYSICal screen. Remember that the box was only drawn on the PHYSICal screen and is thus not on the BACKground screen. Copying BACK to PHYSIC therefore removes the box from the screen.

The *wend* command at line 260 sends the program back to line 210 where the left mouse button is checked again. If the mouse button has been pressed the program passes on to line 280 where the box is drawn at its final position on the screen (*autoback on* restores normal graphics operation). If the mouse button has not been pressed then the box is redrawn by line 220 and the whole process starts again.

Although a fairly simple concept, it is quite a difficult one to follow. Look carefully at the program listing and run it a few times to see the exact operation. Each time the user moves the mouse pointer, the box is removed from the screen (copy BACK to PHYSIC) and redrawn using the new mouse position.

Finally, we shall look at a program that uses the expanding box routine in a practical application. The following program loads a picture file and lets the user select a rectangular section using an expanding box. The 'cut' section is then zoomed to fill the entire screen.

Load the following:

```
10 rem EXPAND AREA OF SCREEN
20 rem PROGRAM = A:\SCREEN\EXPAND2
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off
60 :
70 rem LOAD PICTURE
80 F$ = file select$("a:\pictures\*.pi1")
90 load F$
100 :
110 rem LET USER SELECT AREA OF SCREEN
120 auto back off
130 repeat : until mouse key = 1
140 X = x mouse : Y = y mouse
150 X1 = X : Y1 = Y
160 repeat : until mouse key = 0
170 while mouse key < > 1
180 box X,Y to X1,Y1
190 X1 = x mouse : Y1 = y mouse
200 while X1 = x mouse and Y1 = y mouse and mouse
key = 0 : wend
210 screen copy back to physic
220 wend
230 :
240 rem ZOOM THE SCREEN
250 autoback on
260 zoom X,Y,X1,Y1 to 0,0,319,199
```

The program allows the user to choose a picture file which is loaded in to memory bank 14 and displayed on the screen. The user can then cut a small section from the picture by moving the mouse pointer to the top corner, clicking the left mouse button, and stretching the box over the required image. When the mouse button is pressed again, the

segment is 'cut' and expanded to fill the entire screen. So let us see how the program works.

Line 80 displays a file selector box and line 90 loads the chosen file so that it is displayed on both the PHYSICal and BACKground screens.

Lines 110-220 form the 'expanding' box routine as previously described.

Line 250 finally expands the selected area to fill the entire screen.

HARDCOPY

You may be aware that the Atari ST has an in-built screen dump utility that is activated by pressing the ALTERNATE and HELP keys together. This facility prints a copy of the current screen to the printer and can also be activated from within a STOS program using the *hardcopy* command.

Enter the following:

```
10 key off: mode 0: flash off
20 load "a:\pictures\pictures.pi1"
30 hardcopy
```

Run the program and it will print the screen to your printer. Note that the screen dump routine is designed for a nine pin, dot matrix printer, and although it may work on other types of printer, you may not get an exact reproduction of the screen. For example, on most 24 pin printers you lose the right hand edge of the picture.

So there we have it, screen copying, screen zooming and reducing, animated graphics, etc. To explore the concepts further we shall now

write a complete game using the animation techniques plus many of the other concepts discussed so far. The listing of a complete game is going to be somewhat longer than any programs previously encountered but do not worry because, as usual, we shall break it down in to small segments and look at each of these in turn. Programs always look a lot more complicated than they are. This is unfortunate and probably deters a lot of would be programmers from taking the plunge when they see such program listings in computer magazines, etc.

Chapter 15

Bonk the Gonk Game

In the last chapter we saw how the *screen copy* command could be used to produce animation. We shall continue on the theme of animation and incorporate the techniques previously discussed in to a complete game.

We shall produce a game called BONK THE GONK. This is based on the fair ground game where you have to hit or 'bonk' the small creatures (Gonks) on the head with a mallet as they pop up out of the holes. Obviously we cannot hit the computer screen so we shall use the mouse to control a mallet which is displayed on the screen.

Before delving in to the workings of the program, load it from disk and have a few goes to see exactly what it does.

Load the following:

```
10 rem BONK THE GONK GAME
20 rem PROGRAM = A:\BONKGONK\BONKGONK
```



```
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off : click off : curs off :
hide on
60 :
70 rem LOAD PICTURES
80 reserve as screen 13
90 reserve as screen 14
100 reserve as screen 15
110 load "a:\bonkgonk\tile.pi1",13
120 load "a:\bonkgonk\gonks.pi1",14
130 load "a:\bonkgonk\holes.pi1",15
140 :
150 rem CHANGE PALETTE TO SUIT PICTURES
160 get palette (13)
170 :
180 rem DISPLAY TITLE SCREEN
190 screen copy 13 to physic : screen copy 13 to back
200 :
210 rem SET ZONES FOR CHOOSING LEVEL
220 set zone 1,3,91 to 105,109
230 set zone 2,109,91 to 210,109
240 set zone 3,215,91 to 317,109
250 :
260 rem WAIT FOR USER TO CHOOSE LEV
270 change mouse 2 : show on
280 while Z=0 or mouse key < > 1
290 Z=zone(0)
300 wend
310 if Z=1 then TIME_ALLOWED=50
320 if Z=2 then TIME_ALLOWED=27
330 if Z=3 then TIME_ALLOWED=17
340 :
350 rem FADE THE TITLE SCREEN
360 fade 10 : wait 10*7
```

```
370 :
380 rem PLACE MAIN BACKGROUND(HOLES) ON
SCREEN
390 screen copy 15 to physic : screen copy 15 to back
400 fade 10 to 15 : wait 10*7
410 change mouse 4
420 :
430 rem INITIALISE VARIABLES
440 MALLETS = 3 : GONK = 0 : S = 0
450 :
460 repeat
470 TIME_TAKEN = 0
480 :
490 rem CHOOSE A HOLE AT RANDOM
500 H = rnd(11)
510 if H = 0 then X = 0 : Y = 50
520 if H = 1 then X = 80 : Y = 50
530 if H = 2 then X = 160 : Y = 50
540 if H = 3 then X = 240 : Y = 50
550 if H = 4 then X = 0 : Y = 100
560 if H = 5 then X = 80 : Y = 100
570 if H = 6 then X = 160 : Y = 100
580 if H = 7 then X = 240 : Y = 100
590 if H = 8 then X = 0 : Y = 150
600 if H = 9 then X = 80 : Y = 150
610 if H = 10 then X = 160 : Y = 150
620 if H = 11 then X = 240 : Y = 150
630 :
640 rem DISPLAY GONK OR DYNAMITE ?
650 R = rnd(9)
660 if R = 0 or R = 5 or R = 9 then R$ = "DYNAMITE" :
gosub 1040 else R$ = "GONK" : gosub 930
670 :
680 rem MONITOR MOUSE
690 show on
```



```
700 while TIME_TAKEN < TIME_ALLOWED
710 if x mouse > X and x mouse < X + 80 and
y mouse > Y and y mouse < Y + 50 and mouse key = 1 and
R$ = "GONK" then gosub 1140 : goto 810
720 if x mouse > X and x mouse < X + 80 and
y mouse > Y and y mouse < Y + 50 and mouse key = 1 and
R$ = "DYNAMITE" then gosub 1290 : goto 810
730 wait 1
740 inc TIME_TAKEN
750 wend
760 :
770 rem NOT HIT SO RETURN TO HOLE
780 if R$ = "GONK" then gosub 1390 else gosub 1500
790 :
800 rem UPDATE SCORE AND GO BACK FOR
ANOTHER GO
810 locate 3,3 : print S
820 locate 34,3 : print MALLETS
830 wait 50
840 until MALLETS = 0 or GONKS = 50
850 :
860 rem GAME OVER
870 cls
880 print "GAME OVER"
890 print : print "YOU SCORED ";S
900 end
910 :
920 rem SUBROUTINES
930 rem GONK APPEARS FROM HOLE
940 inc GONKS
950 hide on
960 screen copy 14,0,100,80,150 to physic,X,Y
970 wait 3
980 screen copy 14,0,50,80,100 to physic,X,Y
990 wait 3
```

```
1000 screen copy 14,0,0,80,50 to physic,X,Y
1010 screen copy 14,0,0,80,50 to back,X,Y
1020 return
1030 :
1040 rem DYNAMITE APPEARS FROM HOLE
1050 hide on
1060 screen copy 14,160,100,240,150 to physic,X,Y
1070 wait 3
1080 screen copy 14,160,50,240,100 to physic,X,Y
1090 wait 3
1100 screen copy 14,160,0,240,50 to physic,X,Y
1110 screen copy 14,160,0,240,50 to back,X,Y
1120 return
1130 :
1140 rem RETURN HIT-UNHAPPY GONK
1150 bell
1160 hide on
1170 screen copy 14,80,0,160,50 to physic,X,Y
1180 wait 30
1190 screen copy 14,80,50,160,100 to physic,X,Y
1200 wait 2
1210 screen copy 14,80,100,160,150 to physic,X,Y
1220 wait 2
1230 screen copy 14,0,150,80,200 to physic,X,Y
1240 screen copy 14,0,150,80,200 to back,X,Y
1250 wait 2
1260 inc S
1270 return
1280 :
1290 rem DYNAMITE HAS BEEN HIT -EXPLODE
1300 hide on
1310 screen copy 14,160,150,240,200 to physic,X,Y
1320 boom : boom
1330 dec MALLETS
1340 wait 50
```



```
1350 screen copy 14,0,150,80,200 to physic,X,Y
1360 screen copy 14,0,150,80,200 to back,X,Y
1370 return
1380 :
1390 rem RETURN GONK UNHARMED
1400 hide on
1410 screen copy 14,0,50,80,100 to physic,X,Y
1420 wait 2
1430 screen copy 14,0,100,80,150 to physic,X,Y
1440 wait 2
1450 screen copy 14,0,150,80,200 to physic,X,Y
1460 screen copy 14,0,150,80,200 to back,X,Y
1470 wait 30
1480 return
1490 :
1500 rem RETURN DYNAMITE TO HOLE
1510 hide on
1520 screen copy 14,160,50,240,100 to physic,X,Y
1530 wait 2
1540 screen copy 14,160,100,240,150 to physic,X,Y
1550 wait 2
1560 screen copy 14,0,150,80,200 to physic,X,Y
1570 screen copy 14,0,150,80,200 to back,X,Y
1580 return
```

The aim of the game is as follows:

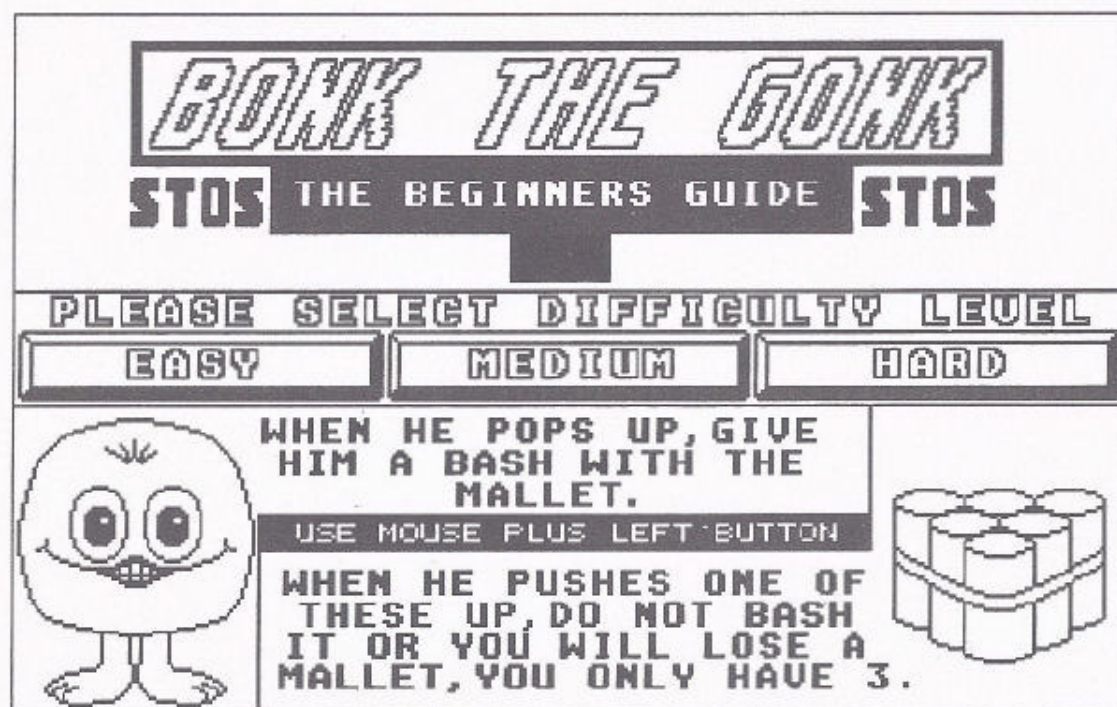
Fifty Gonks will appear from the holes at random. The length of time that they are displayed will depend on the level that is selected at the start of the game. You must hit as many of the Gonks as possible by moving the mallet over the top of them and pressing the left mouse button to bonk them on the head. At the end of the game your score will be displayed, BUT beware of the dynamite which also pops up from the holes occasionally. If you bonk the dynamite you will lose a mallet and you only have three!

Ok, so that is enough playing. I am sure you will agree that although the game is based on a very simple concept it does become rather addictive. The game is aimed at younger ST users but if you want a little more of a challenge we shall see how to speed the game up later on. Let us see how it works then.

The program uses four memory banks. One memory bank is used to store sprite data and the other three to store screen data. The sprite data is stored on disk along with the program (remember that memory bank 1 is used for sprite data and this is a permanent bank) and the three screens are loaded within the program. The three screens are shown below.

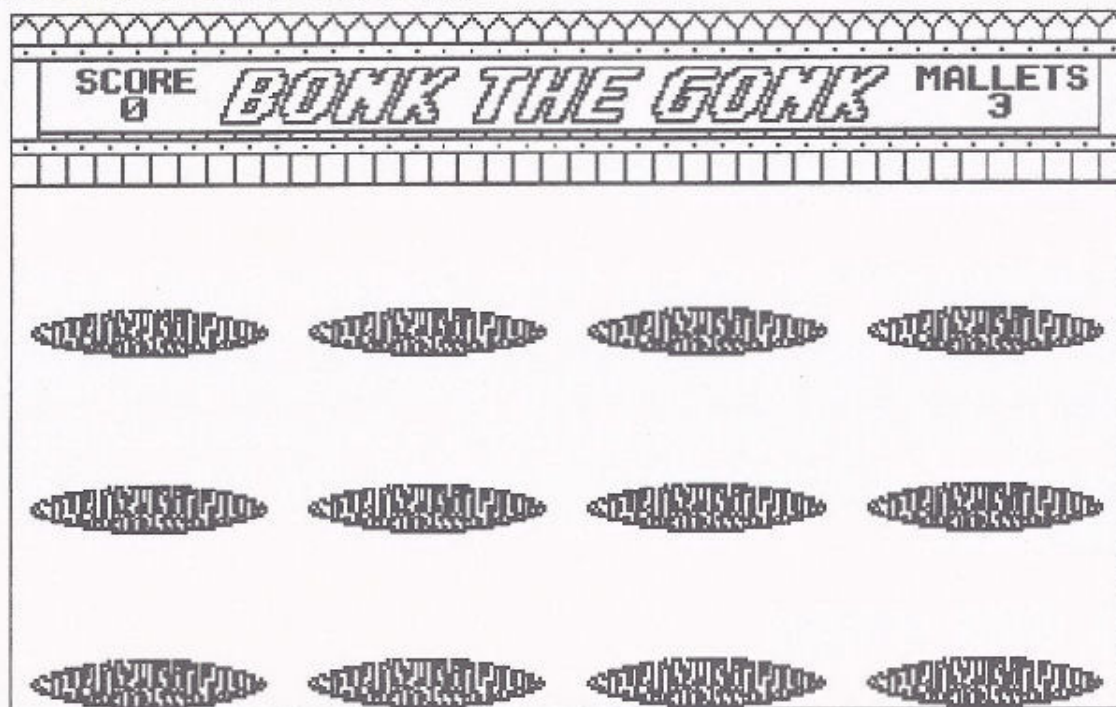
THE TITLE SCREEN

The title screen is displayed at the start of the game and allows the user to choose the required level. The level is selected by clicking the mouse pointer on one of the boxes EASY, MEDIUM or HARD.

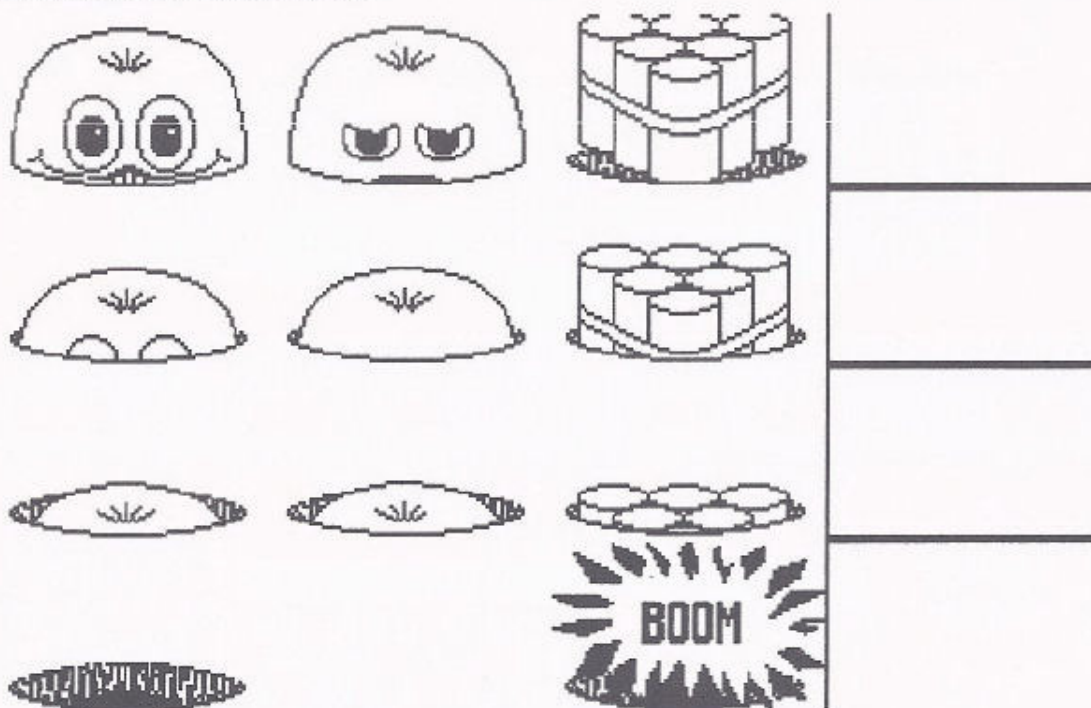


The other two screens are used for the main game play.

THE BACKDROP SCREEN



THE GONKS SCREEN



As you can see from the program listing, the game relies heavily on the *screen copy* command and employs the same methods as that previously discussed for the walking man. The Gonks are moved in and out of the holes by copying a number of frames in the same position on the screen. The frames are held in the screen named "GONKS.PI1" and these are copied to the main game screen "HOLES.PI1" to produce the animation.

The Gonk screen contains the various positions of the Gonks and dynamite, and by copying these individual images or frames to the backdrop screen, we can make the Gonks and dynamite move in and out of the holes. The images of the Gonks on the left hand side of the screen are used for making the Gonk appear from the hole and return back to the hole unharmed. The images on the right hand side make the Gonk look sad and are used to return the Gonk to the hole after it has been hit.

The program is constructed from the following modules:

- (1) Set environment.
- (2) Display title screen and ask user to select level.
- (3) Display background screen.
- (4) Choose random hole , Gonk or dynamite and display.
- (5) Monitor mouse to see if Gonk or dynamite is hit.
- (6) Display final score.

SUBROUTINES

- (7) Make Gonk appear from hole.
- (8) Make dynamite appear from hole.
- (9) Make unhappy gonk return in to hole.
- (10) Dynamite hit - explosion.
- (11) Return Gonk to hole unharmed.
- (12) Return dynamite to hole.

Let us now look at the actual code.

(1) Set environment

```
40 rem SET SCREEN
50 key off : mode 0 : flash off : click off : curs off : hide on
60 :
70 rem LOAD PICTURES
80 reserve as screen 13
90 reserve as screen 14
100 reserve as screen 15
110 load "a:\bonkgonk\title.pi1",13
120 load "a:\bonkgonk\gonks.pi1",14
130 load "a:\bonkgonk\holes.pi1",15
140 :
150 rem CHANGE PALETTE TO SUIT PICTURES
160 get palette (13)
```

Line 50 switches off the function key window, sets the screen resolution low, switches of colour flashing, switches of key click, switches off the cursor and hides the mouse pointer.

Lines 80-130 load the various picture files. The title screen is loaded in to memory bank 13, the screen contain the gonks is loaded in to memory bank 14 and the main backdrop screen is loaded in to bank 15.

Line 160 sets the current colour palette to suit the picture held in memory bank 13. The same colour palette has been used for all three screens and hence we could have grabbed the palette from any of the three memory banks.

(2) Display title screen and ask user to select level

```
180 rem DISPLAY TITLE SCREEN
190 screen copy 13 to physic : screen copy 13 to back
200 :
210 rem SET ZONES FOR CHOOSING LEVEL
220 set zone 1,3,91 to 105,109
230 set zone 2,109,91 to 210,109
240 set zone 3,215,91 to 317,109
250 :
260 rem WAIT FOR USER TO CHOOSE LEV
270 change mouse 2 : show on
280 while Z=0 or mouse key < > 1
290 Z=zone(0)
300 wend
310 if Z=1 then TIME_ALLOWED=50
320 if Z=2 then TIME_ALLOWED=27
330 if Z=3 then TIME_ALLOWED=17
```

Line 190 copies the title screen from memory bank 13 to both the PHYSICAL and BACKground screens.

The screen contains three buttons labelled EASY, MEDIUM and HARD. These are used to select the level by clicking the mouse pointer on them. Lines 220-240 therefore set up three zones which can be used to test for the presence of the mouse pointer.

Line 270 changes the mouse pointer to a pointing hand and shows it on the screen.

Lines 280-300 form a *while..wend* loop which monitors the zones for the presence of the mouse pointer. If the mouse pointer enters one of the zones and the left mouse button is pressed (user has made selection), the selected zone is assigned to variable Z and the loop is terminated.

Lines 310-330 set the `TIME_ALLOWED` variable. Depending on the level button selected, the variable `TIME_ALLOWED` is assigned a value which will later control the amount of time that each Gonk is displayed before returning back in to the hole. Notice that the easy level (zone 1) assigns a higher value (longer time) than the harder levels. If you want to make the game harder you can reduce these values to limit even further the amount of time that each Gonk is displayed.

(3) Display background screen

```
350 rem FADE THE TITLE SCREEN
360 fade 10 : wait 10*7
370 :
380 rem PLACE MAIN BACKGROUND(HOLES) ON
SCREEN
390 screen copy 15 to physic : screen copy 15 to back
400 fade 10 to 15 : wait 10*7
410 change mouse 4
420 :
430 rem INITIALISE VARIABLES
440 MALLETS=3 : GONK=0 : S=0
```

Line 360 fades the entire colour palette to black and the title screen fades from the screen. Remember the *wait* command that is always used with the *fade* command.

Line 390 copies the background screen (held in memory bank 15) to the `PHYSICAL` and `BACKGROUND` screens. Line 400 fades the colour palette back up and the background becomes visible.

Line 410 changes the mouse pointer to a mallet. The image for the mallet is held as a sprite and this is assigned to the mouse pointer using the *change mouse* command. Note that there is only one sprite

(the mallet) in the sprite bank.

Line 440 initialises three variables that are used throughout the game. Variable MALLETs maintains the number of mallets remaining. The game starts with three mallets and the user loses one each time that they hit the dynamite. Variable GONK maintains the number of Gonks displayed so far - the game displays fifty Gonks before ending. The variable S maintains the score and is increased by one each time that a Gonk is successfully hit with the mallet.

(4) Choose random hole, Gonk or dynamite and display

```
460 repeat
470 TIME_TAKEN=0
480 :
490 rem CHOOSE A HOLE AT RANDOM
500 H=rnd(11)
510 if H=0 then X=0 : Y=50
520 if H=1 then X=80 : Y=50
530 if H=2 then X=160 : Y=50
540 if H=3 then X=240 : Y=50
550 if H=4 then X=0 : Y=100
560 if H=5 then X=80 : Y=100
570 if H=6 then X=160 : Y=100
580 if H=7 then X=240 : Y=100
590 if H=8 then X=0 : Y=150
600 if H=9 then X=80 : Y=150
610 if H=10 then X=160 : Y=150
620 if H=11 then X=240 : Y=150
630 :
640 rem DISPLAY GONK OR DYNAMITE ?
650 R=rnd(9)
660 if R=0 or R=5 or R=9 then R$="DYNAMITE" :
gosub 1040 else R$="GONK" : gosub 930
```


Line 470 clears the `TIME_TAKEN` variable that is later used to maintain the amount of time that the Gonk or dynamite has been out of the hole.

There are twelve holes on the screen and we need to select one of these at random for the Gonk or dynamite to appear from. Line 500 therefore chooses a random number in the range 0-11 and assigns this to variable `H`.

Lines 510-620 assign appropriate screen coordinates to the variables `X` and `Y` depending on the hole that is chosen. These coordinates indicate the top left corner of the area above the holes where we shall copy various frames to produce animation - this will become clearer in a moment.

We need to decide whether to display a Gonk or dynamite. Line 650 chooses a random number in the range 0-9 and assigns this to variable `R`.

Line 660 uses the random number to decide whether to display a Gonk or dynamite. We want to display mainly Gonks so we shall only display dynamite when variable `R` is equal to zero, five or nine. All other values will result in a Gonk being displayed. The dynamite is displayed by a subroutine at line 1040 and the Gonk is displayed by a subroutine at line 930. These display the dynamite or Gonk moving up out of the hole and we shall look at this in a moment.

(5) Monitor mouse to see if Gonk or dynamite is hit

```
680 rem MONITOR MOUSE
690 show on
700 while TIME_TAKEN < TIME_ALLOWED
710 if x mouse > X and x mouse < X+80 and y mouse > Y
and y mouse < Y+50 and mouse key=1 and R$="GONK" then
```

```
gosub 1140 : goto 810
    720 if x mouse > X and x mouse < X+80 and y mouse > Y
and y mouse < Y+50 and mouse key = 1 and R$ = "DYNAMITE" then
gosub 1290 : goto 810
    730 wait 1
    740 inc TIME_TAKEN
    750 wend
    760 :
    770 rem NOT HIT SO RETURN TO HOLE
    780 if R$ = "GONK" then gosub 1390 else gosub 1500
```

Now that the Gonk or dynamite is displayed, we need to let the user try and hit it. Line 690 therefore displays the mouse pointer which is now in the shape of a mallet.

Lines 700-750 form a *while..wend* loop which continues until the time allowed (specified by variable TIME_ALLOWED which is set at the start of the program when the user selects the level) has elapsed.

We now need to monitor the mouse to see if it is clicked within the area above the hole and hence on the Gonk or dynamite. Two checks are required, one to check the mouse when a Gonk is displayed and one to check the mouse when dynamite is displayed.

Line 710 checks to see if the Gonk has been hit and if so calls the subroutine at line 1140 which displays the hit Gonk returning to the hole and increments the score.

Line 720 checks to see if the dynamite has been hit and if so calls the subroutine at line 1290 which displays an explosion and decrements the number of mallets remaining.

Line 740 increments the amount of TIME_TAKEN and line 750 sends execution back to the beginning of the loop.

If the Gonk or dynamite is not hit within the specified time allowed, line 780 calls the appropriate subroutine to return the Gonk or dynamite back to the hole unharmed.

```
800 rem UPDATE SCORE + BACK FOR ANOTHER GO
810 locate 3,3 : print S
820 locate 34,3 : print MALLETS
830 wait 50
840 until MALLETS=0 or GONKS=50
```

Lines 810-820 update the score and number of mallets remaining on the screen. Line 830 causes the program to wait for a second.

Line 840 checks to see if all the mallets have been used and whether all the Gonks have been displayed. The program displays 50 Gonks, and if they have not all been displayed and provided there are still mallets available, program execution passes back to line 460 (*repeat* command).

If all the Gonks have been displayed or if all the mallets have been lost, the program moves on to display the final score.

(6) Display final score

```
860 rem GAME OVER
870 cls
880 print "GAME OVER"
890 print : print "YOU SCORED ";S
900 end
```

Line 870 clears the screen and lines 880-900 display the final score and end the program.

SUBROUTINES

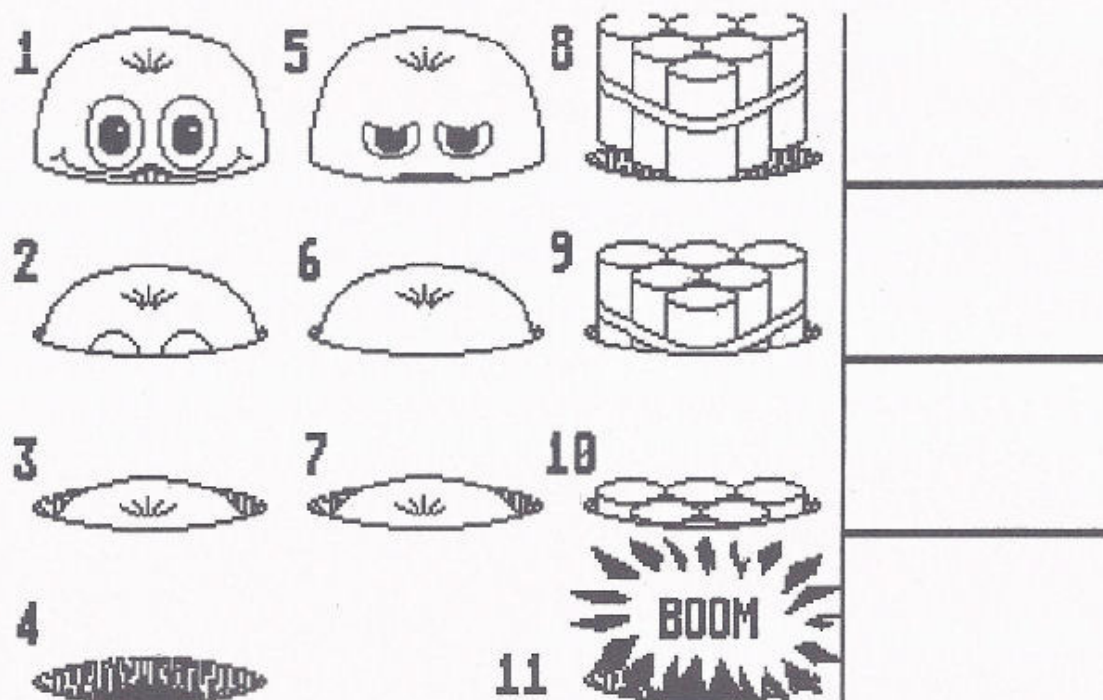
We shall now look at the various subroutines that are called from the main program and perform the animation of the Gonks and dynamite in and out of the holes. To make these easier to follow refer to the screen shown over the page which lists the coordinates of the various frames. You will see that each frame is 80 pixels wide by 50 pixels high and by placing each frame on top of each other we can produce animation to make the Gonks and dynamite move in and out of the hole.

(7) Make Gonk appear from hole

```
930 rem GONK APPEARS FROM HOLE
940 inc GONKS
950 hide on
960 screen copy 14,0,100,80,150 to physic,X,Y
970 wait 3
980 screen copy 14,0,50,80,100 to physic,X,Y
990 wait 3
1000 screen copy 14,0,0,80,50 to physic,X,Y
1010 screen copy 14,0,0,80,50 to back,X,Y
1020 return
```

Line 940 increments the variable GONKS which maintains the total number of Gonks displayed so far and line 950 hides the mouse pointer (mallet).

Lines 960-1010 perform the animation and make the gonk move up out of the hole. Line 960 displays image 3, line 980 displays image 2 and lines 1000-1010 display image 1. The *wait* commands are included in between in each of the *screen copy* commands to slow the animation process down. If these were removed, the Gonk would move out of the hole too fast and the animation effect would be lost.



1 = 0,0 - 80,50
 2 = 0,50 - 80,100
 3 = 0,100 - 80,150
 4 = 0,150 - 80,200
 5 = 80,0 - 160,50
 6 = 80,50 - 160,100

7 = 80,100 - 160,150
 8 = 160,0 - 240,50
 9 = 160,50 - 240,100
 10 = 160,100 - 240,150
 11 = 160,150 - 240,200

(8) Make dynamite appear from hole

```

1040 rem DYNAMITE APPEARS FROM HOLE
1050 hide on
1060 screen copy 14,160,100,240,150 to physic,X,Y
1070 wait 3
1080 screen copy 14,160,50,240,100 to physic,X,Y
1090 wait 3
1100 screen copy 14,160,0,240,50 to physic,X,Y
1110 screen copy 14,160,0,240,50 to back,X,Y
1120 return
  
```

Line 1050 hides the mouse pointer and lines 1060-1110 perform the animation and make the dynamite appear from the hole. Line 1060 displays image 10, line 1080 displays image 9 and lines 1100/1110 display image 8.

(9) Make unhappy gonk return in to hole

```
1140 rem RETURN HIT-UNHAPPY GONK
1150 bell
1160 hide on
1170 screen copy 14,80,0,160,50 to physic,X,Y
1180 wait 30
1190 screen copy 14,80,50,160,100 to physic,X,Y
1200 wait 2
1210 screen copy 14,80,100,160,150 to physic,X,Y
1220 wait 2
1230 screen copy 14,0,150,80,200 to physic,X,Y
1240 screen copy 14,0,150,80,200 to back,X,Y
1250 wait 2
1260 inc S
1270 return
```

When a Gonk is hit, it changes colour and returns to the hole looking rather unhappy.

Line 1150 generates a bell type sound, line 1160 hides the mouse pointer and lines 1170-1240 animate the Gonk returning to the hole. Line 1170 displays image 5, line 1190 displays image 6, line 1210 displays image 7 and lines 1230/1240 display image 4. Line 1260 increments the score.

(10) Dynamite hit - explosion

```
1290 rem DYNAMITE HAS BEEN HIT -EXPLODE
1300 hide on
1310 screen copy 14,160,150,240,200 to physic,X,Y
1320 boom : boom
1330 dec MALLETS
1340 wait 50
1350 screen copy 14,0,150,80,200 to physic,X,Y
1360 screen copy 14,0,150,80,200 to back,X,Y
1370 return
```

When the user hits the dynamite we have to display an explosion.

Line 1300 hides the mouse pointer, line 1310 displays the explosion (image 11), line 1320 generates two boom type sounds and line 1330 decrements the number of mallets remaining. Remember that the user starts the game with three mallets and one is lost each time that dynamite is hit.

Line 1340 halts program execution of one second to give the user time to see the explosion on the screen.

Lines 1350/1360 copy image 4 to the screen which removes the explosion and restores the hole to normal.

(11) Return Gonk to hole unharmed

```
1390 rem RETURN GONK UNHARMED
1400 hide on
1410 screen copy 14,0,50,80,100 to physic,X,Y
1420 wait 2
1430 screen copy 14,0,100,80,150 to physic,X,Y
1440 wait 2
```

```
1450 screen copy 14,0,150,80,200 to physic,X,Y
1460 screen copy 14,0,150,80,200 to back,X,Y
1470 wait 30
1480 return
```

If the user does not bonk the gonk within the specified time scale then it must be returned to the hole.

Line 1400 hides the mouse pointer and lines 1410-1460 perform the animation. Line 1410 displays image 2, line 1430 displays image 3 and lines 1450/1460 display image 4.

Line 1470 halts program execution for 30/50ths of a second.

(12) Return dynamite to hole

```
1500 rem RETURN DYNAMITE TO HOLE
1510 hide on
1520 screen copy 14,160,50,240,100 to physic,X,Y
1530 wait 2
1540 screen copy 14,160,100,240,150 to physic,X,Y
1550 wait 2
1560 screen copy 14,0,150,80,200 to physic,X,Y
1570 screen copy 14,0,150,80,200 to back,X,Y
1580 return
```

If the user does not bonk the dynamite it must be returned to the hole.

Line 1510 hides the mouse pointer and lines 1520-1570 perform the animation. Line 1520 displays image 9, line 1540 displays image 10 and lines 1560/1570 display image 4.

MAKING THE GAME HARDER

(1) The game play can be made harder by reducing the amount of time that each Gonk remains out of the hole. This is determined by variable `TIME_ALLOWED` which is set in lines 310-330. The lower the value assigned to the variable `TIME_ALLOWED`, the harder the game will be.

(2) The game play can also be made harder by increasing the number of times that dynamite is displayed. This is controlled by lines 650-660 and by changing line 660 you can control the ratio of Gonks to dynamite.

That concludes the explanation of Bonk the Gonk.

Chapter 16

Creating an Art Package

Whilst different people have different interests there cannot be many ST owners who have not used an art program of some sort. The ability to draw coloured pictures directly on the screen seems to offer a strange attraction to both computer and non-computer enthusiasts alike, and there are many programs available that perform this task.

In this chapter we shall develop our own art package using the graphic and screen techniques covered in previous chapters. The final program will be some what longer than anything encountered so far, but do not worry, as it shall be designed as a series of small modules in the normal manner. Let us start by defining the programs operation.

PROGRAM DEFINITION

The program shall allow the user to produce pictures in low resolution and shall allow these pictures to be loaded from and saved to disk in

degas (.PI1) format. The program shall offer the following drawing facilities:

- * Switch drawing colour to any colour from a palette of sixteen.
- * Allow the colour palette to be altered.
- * Draw freehand.
- * Draw lines using one of three different line styles. The lines may also have normal or arrowed endings.
- * Draw outlined and filled boxes with square corners.
- * Draw outlined and filled boxes with rounded corners.
- * Draw outlined and filled circles and ellipses.
- * Paint areas of the screen using one of twenty four styles.
- * Cut blocks and copy them to different parts of the screen.
- * Use text with normal, inverted, underlined and shaded effects.
- * Load and save pictures to and from disk.
- * Clear the current picture and quit the program.

All options shall be selected from a main options screen using the mouse, and when an option is selected, the program will 'switch' to the drawing screen. Zones shall be used to determine which option has been selected.

The right mouse button shall be used to switch back from the drawing screen to the options screen.

PROGRAM DESIGN

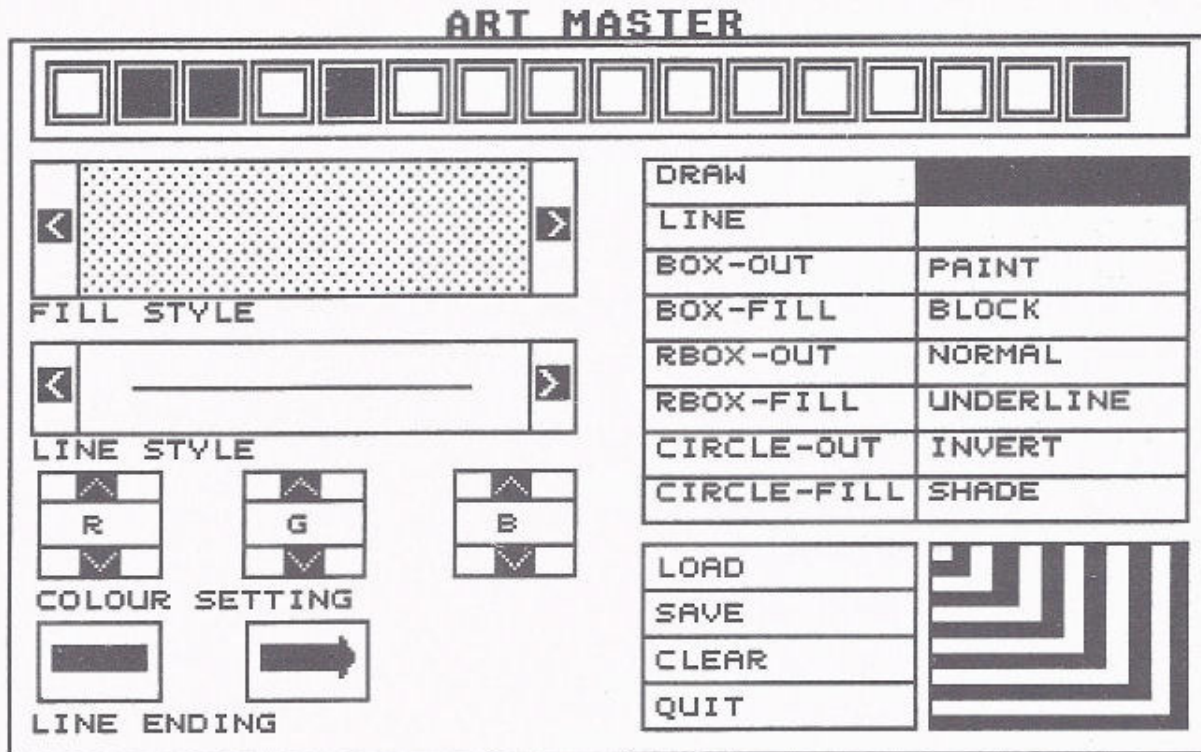
Now that the programs operation has been defined we can separate the various operations in to a number of modules.

- (1) Set screen resolution and initialise variables.
- (2) Reserve memory banks and load option screen.
- (3) Define zones.
- (4) Display main screen and monitor the zones.
- (5) Change the ink colour.
- (6) Change the paint style.
- (7) Change the line style.
- (8) Set the line ending.
- (9) Draw freehand.
- (10) Draw outlined and filled shapes.
- (11) Load picture from disk.
- (12) Save picture to disk.
- (13) Clear the current picture.
- (14) Change the red colour element.
- (15) Change the green colour element.
- (16) Change the blue colour element.
- (17) Paint an area of the screen.
- (18) Text.
- (19) Cut a block.
- (20) Quit the program.

Each module performs a specific task, and when considered individually, is quite easy to program and understand.

Our first consideration is that of the main, option screen. There are many methods that could be used but with so many options available, and bearing in mind that the user will be using the mouse to draw, the most appropriate method is to display all of the options on the screen and let the user select them using the mouse.

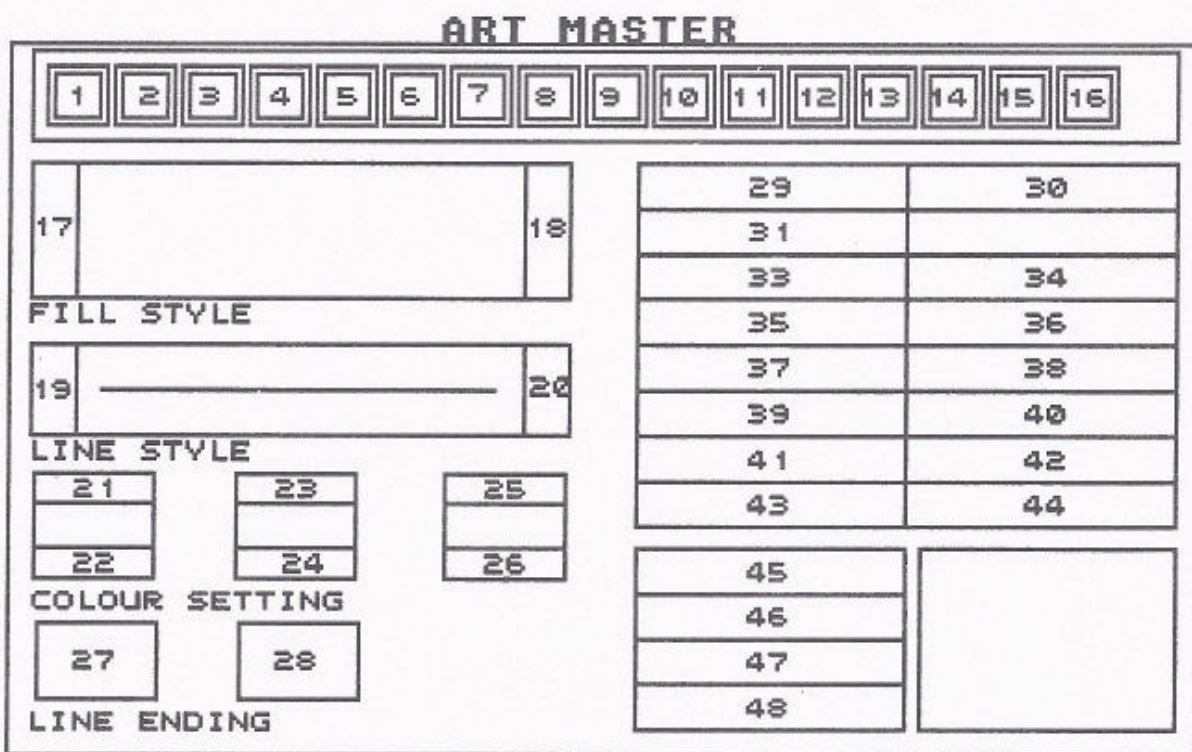
A suitable screen layout is shown below. This was produced using an art package and saved as a low resolution picture (ART.PI1). At the start of the program this can be loaded in to a memory bank and can be recalled from here whenever required.



Look closely at the screen layout. The main drawing options, lines, circles, etc. are listed down the right hand side and can be selected by moving the mouse pointer over the appropriate box and clicking the left mouse button. The desired drawing or ink colour can be selected by clicking the mouse pointer on the colour boxes across the top of the screen. The fill style, line style and colour settings can be changed by clicking the mouse pointer on the arrows which cycle through the various styles or settings.

We now need to consider the way in which we are going to monitor the option screen and determine which options have been selected. We saw in a previous chapter how zones can be used to monitor areas of the screen for the presence of the mouse pointer and this is the method

we shall use here. Forty eight separate zones are defined to cover all of the selectable options or areas as the following diagram illustrates:



Well that covers the theory so let us now look at the actual program.

THE PROGRAM

Place disk 2 in to the disk drive and load the following:

```

10 rem ART MASTER DRAWING PROGRAM
20 rem PROGRAM = A:\ART\ART.BAS
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off : curs off
60 :
70 rem SET VARIABLES
80 I = 1 : ink I : pen I
90 PSTYLE = 1 : set paint 2,PSTYLE,1

```



```
100 LSTYLE = 1 : LE = 0 : dim L(3)
110 L(1) = %1111111111111111
120 L(2) = %1111000011110000
130 L(3) = %1010101010101010
140 set line L(1),1,LE,LE
150 :
160 rem RESERVE BANKS AND LOAD MAIN SCREEN
170 reserve as screen 14
180 reserve as screen 15
190 load "a:\art\art.pi1",15
200 get palette (15)
210 cls 14
220 :
230 rem DEFINE ZONES FOR COLOUR SELECTION
240 Z = 0
250 for A = 12 to 282 step 18
260 inc Z
270 set zone Z,A,18 to A + 12,30
280 next A
290 :
300 rem DEFINE ZONES FOR FILL STYLE SELECTION
310 set zone 17,7,43 to 17,77
320 set zone 18,139,43 to 149,77
330 :
340 rem DEFINE ZONES FOR LINE STYLE SELECTION
350 set zone 19,7,92 to 17,113
360 set zone 20,139,91 to 149,113
370 :
380 rem DEFINE ZONES FOR CHANGING COLOURS
390 set zone 21,8,124 to 40,132 : set zone 22,8,144
to 40,152
400 set zone 23,62,124 to 94,132 : set zone
24,62,144 to 94,152
410 set zone 25,118,124 to 150,132 : set zone
26,118,144 to 150,152
```

```
420 :
430 rem DEFINE ZONES FOR LINE ENDING SELECTION
440 set zone 27,9,165 to 39,184
450 set zone 28,63,165 to 93,184
460 :
470 rem DEFINE ZONES FOR OPTION SELECTION
480 Z = 28
490 for A = 43 to 127 step 12
500 inc Z
510 set zone Z,169,A to 239,A + 10
520 inc Z
530 set zone Z,241,A to 311,A + 10
540 next A
550 :
560 rem DEFINE ZONES FOR FILE OPTION SELECTION
570 for A = 145 to 181 step 12
580 inc Z
590 set zone Z,169,A to 239,A + 10
600 next A
610 :
620 rem DISPLAY MAIN SCREEN + MONITOR ZONES
630 screen copy 15 to physic : screen copy 15 to back
640 show on
650 Z = zone(0)
660 if Z = 0 or mouse key < > 1 then 650
670 repeat : until mouse key = 0
680 on Z gosub 710, 710, 710, 710, 710, 710, 710,
710, 710, 710, 710, 710, 710, 710, 710, 800, 800,
900, 890, 1790, 1790, 1890, 1890, 1990, 1990, 1000,
1000, 1050, 690, 1130, 690, 1130, 2090, 1130, 2310,
1130, 2170, 1130, 2170, 1130, 2170, 1130, 2170, 1460,
1580, 1670, 2630
690 goto 630
700 :
710 rem CHANGE THE INK COLOUR - ZONES 1-16
```



```
720 I=Z-1
730 ink I
740 set paint 1,1,1
750 bar 241,43 to 311,53
760 set paint 2,PSTYLE,1
770 hide on : screen copy physic to 15
780 return
790 :
800 rem CHANGE THE PAINT STYLE - ZONES 17 + 18
810 if Z=17 and PSTYLE>1 then dec PSTYLE
820 if Z=18 and PSTYLE<24 then inc PSTYLE
830 set paint 2,PSTYLE,1
840 ink 15
850 bar 18,42 to 138,78
860 ink I
870 hide on : screen copy physic to 15
880 return
890 :
900 rem CHANGE THE LINE STYLE - ZONES 19 + 20
910 if Z=19 and LSTYLE>1 then dec LSTYLE
920 if Z=20 and LSTYLE<3 then inc LSTYLE
930 set line L(LSTYLE),1,0,0
940 ink 15 : polyline 33,102 to 122,102
950 set line L(LSTYLE),1,LE,LE
960 ink I
970 hide on : screen copy physic to 15
980 return
990 :
1000 rem SET LINE ENDINGS - ZONES 27 + 28
1010 if Z=27 then LE=0 else LE=1
1020 set line L(LSTYLE),1,LE,LE
1030 return
1040 :
1050 rem DRAW - ZONE 29
1060 screen copy 14 to physic: screen copy 14 to back
```

```
1070 repeat
1080 if mouse key = 1 then plot x mouse,y mouse
1090 until mouse key = 2
1100 hide on : screen copy physic to 14
1110 return
1120 :
1130 rem DRAW SHAPES - ZONES 31 + 33 + 35 +
37 + 39 + 41 + 43
1140 screen copy 14 to physic:screen copy 14 to back
1150 repeat : until mouse key
1160 if mouse key = 2 then 1430
1170 X = x mouse : Y = y mouse
1180 X1 = X : Y1 = Y
1190 repeat : until mouse key = 0
1200 auto back off
1210 while mouse key < > 1
1220 if Z = 31 then polyline X,Y to X1,Y1
1230 if Z = 33 then box X,Y to X1,Y1
1240 if Z = 35 then bar X,Y to X1,Y1
1250 if Z = 37 then rbox X,Y to X1,Y1
1260 if Z = 39 then rbar X,Y to X1,Y1
1270 if Z = 41 then earc X, Y, abs(X1-X), abs(Y1-Y), 0,
3600
1280 if Z = 43 then ellipse X,Y,abs(X1-X),abs(Y1-Y)
1290 X1 = x mouse : Y1 = y mouse
1300 while X1 = x mouse and Y1 = y mouse and
mouse key = 0 : wend
1310 screen copy back to physic
1320 wend
1330 auto back on
1340 if Z = 31 then polyline X,Y to X1,Y1
1350 if Z = 33 then box X,Y to X1,Y1
1360 if Z = 35 then bar X,Y to X1,Y1
1370 if Z = 37 then rbox X,Y to X1,Y1
1380 if Z = 39 then rbar X,Y to X1,Y1
```



```
1390 if Z = 41 then earc X, Y, abs(X1-X), abs(Y1-Y), 0,
3600
1400 if Z = 43 then ellipse X,Y,abs(X1-X),abs(Y1-Y)
1410 repeat : until mouse key = 0
1420 goto 1150
1430 hide on : screen copy physic to 14
1440 return
1450 :
1460 rem LOAD PICTURE - ZONE 45
1470 F$ = file select$("* .pi1")
1480 if F$ = "" then 1560
1490 hide on
1500 load F$
1510 repeat : until mouse key = 2
1520 screen copy physic to 14
1530 screen copy 15 to physic:screen copy 15 to back
1540 get palette (14)
1550 screen copy physic to 15
1560 return
1570 :
1580 rem SAVE PICTURE TO DISK - ZONE 46
1585 hide on
1590 screen copy 14 to physic
1595 get palette (15)
1610 screen copy physic to 14
1620 show on
1630 F$ = file select$("* .pi1")
1640 if F$ < > "" then save F$,14
1650 return
1660 :
1670 rem CLEAR CURRENT PICTURE - ZONE 47
1680 limit mouse 0,0 to 1,1 : hide on
1690 ink 0 : bar 169,43 to 311,137
1700 pen 15
1710 locate 23,6 : print "CLEAR PICTURE"
```

```
1720 locate 23,8 : print "Are you sure ?"
1730 Q$ = input$(1) : Q$ = upper$(Q$)
1740 if Q$ = "Y" then cls 14 : flash off
1750 ink 1
1760 limit mouse
1770 return
1780 :
1790 rem CHANGE COLOUR CONTENT RED -
ZONES 21 + 22
1800 C = colour(I)
1810 C$ = hex$(C,3)
1820 R$ = mid$(C$,2,1)
1830 if Z = 21 and R$ < > "7" then C = C + $100
1840 if Z = 22 and R$ < > "0" then C = C - $100
1850 colour I,C
1860 hide on : screen copy physic to 15
1870 return
1880 :
1890 rem CHANGE COLOUR CONTENT GREEN -
ZONES 23 + 24
1900 C = colour(I)
1910 C$ = hex$(C,3)
1920 G$ = mid$(C$,3,1)
1930 if Z = 23 and G$ < > "7" then C = C + $10
1940 if Z = 24 and G$ < > "0" then C = C - $10
1950 colour I,C
1960 hide on : screen copy physic to 15
1970 return
1980 :
1990 rem CHANGE COLOUR CONTENT BLUE -
ZONES 25 + 26
2000 C = colour(I)
2010 C$ = hex$(C,3)
2020 B$ = right$(C$,1)
2030 if Z = 25 and B$ < > "7" then C = C + $1
```



```
2040 if Z = 26 and B$ < > "0" then C = C-$1
2050 colour I,C
2060 hide on : screen copy physic to 15
2070 return
2080 :
2090 rem PAINT - ZONE 34
2100 screen copy 14 to physic:screen copy 14 to back
2110 repeat
2120 if mouse key = 1 then paint x mouse,y mouse
2130 until mouse key = 2
2140 hide on : screen copy physic to 14
2150 return
2160 :
2170 rem DISPLAY TEXT - ZONES 38,40,42,44
2180 screen copy 14 to physic:screen copy 14 to back
2190 pen I
2200 repeat : until mouse key
2210 if mouse key = 2 then 2280
2220 locate xtext(x mouse), ytext(y mouse)
2230 if Z = 38 then line input "" ;T$
2240 if Z = 40 then under on : line input "" ;T$ :
under off
2250 if Z = 42 then inverse on : line input "" ;T$ :
inverse off
2260 if Z = 44 then shade on : line input "" ;T$ :
shade off
2270 goto 2200
2280 hide on : screen copy physic to 14
2290 return
2300 :
2310 rem CUT BLOCK - ZONE 36
2320 screen copy 14 to physic:screen copy 14 to back
2330 auto back off
2340 repeat : until mouse key
2350 if mouse key = 2 then 2600
```

```
2360 X=x mouse : Y=y mouse
2370 X1=X : Y1=Y
2380 repeat : until mouse key=0
2390 while mouse key<>1
2400 box X,Y to X1,Y1
2410 X1=x mouse : Y1=y mouse
2420 while X1=x mouse and Y1=y mouse and
mouse key=0 : wend
2430 screen copy back to physic
2440 wend
2450 repeat : until mouse key=0
2460 if X>X1 then swap X,X1 : swap Y,Y1
2470 if Y>Y1 then swap Y,Y1
2480 S$=screen$(physic,X,Y to X1,Y1)
2490 while mouse key<>1
2500 screen$(physic, x mouse, y mouse)=S$
2510 X1=x mouse : Y1=y mouse
2520 while X1=x mouse and Y1=y mouse and
mouse key=0 : wend
2530 screen copy back to physic
2540 wend
2550 auto back on
2560 screen$(physic,X1,Y1)=S$
2570 screen$(back,X1,Y1)=S$
2580 repeat : until mouse key=0
2590 goto 2330
2600 hide on : screen copy physic to 14
2610 return
2620 :
2630 rem QUIT - ZONE 48
2640 limit mouse 0,0 to 1,1 : hide on
2650 ink 0 : bar 169,43 to 311,137
2660 pen 15
2670 locate 23,6 : print "QUIT ARTMASTER"
2680 locate 23,8 : print "Are you sure ?"
```



```
2690 Q$ = input$(1) : Q$ = upper$(Q$)
2700 if Q$ = "Y" then cls : print "BYE BYE" : end
2710 limit mouse
2720 ink 1
2730 return
```

Run the program and get a feel for its operation. This will make the program easier to understand when we come to look at the code in a moment. Try clicking on the different colours at the top of the screen and try selecting lines and circles, etc.

To draw lines - position the mouse pointer at the required starting point and press the left mouse button. Move the mouse pointer around and you will find that the line follows your movements, and when the required position is found, press the left mouse button to fix the line on the screen.

To draw boxes - position the mouse pointer at the required position for one of the corners and press the left mouse button. Move the mouse pointer around you will find that the box expands, and when the required size is found, press the left mouse button to fix the box on the screen.

To draw circles or ellipses - position the mouse pointer at the required position for the centre of the circle or ellipse and press the left mouse button. Move the mouse pointer around and you will find that the ellipse expands, and when the required size is found, press the left mouse button to fix the shape on the screen.

To use text - click the left mouse pointer on NORMAL, UNDERLINE, INVERT or SHADE and type the text on the screen. Press the Return key to end the line of text.

To change the colour palette - click on the arrows either side of the colour boxes to change the relevant strengths of the colours red, green

and blue.

To change the fill and line styles - click on the arrows either side of the relevant boxes and these will cycle through all of the available styles.

We shall now look at each module.

(1) SET SCREEN RESOLUTION AND INITIALISE VARIABLES

```
40 rem SET SCREEN
50 key off : mode 0 : flash off : curs off
60 :
70 rem SET VARIABLES
80 I=1 : ink I : pen I
90 PSTYLE=1 : set paint 2,PSTYLE,0
100 LSTYLE=1 : LE=0 : dim L(3)
110 L(1)= %1111111111111111
120 L(2)= %1111000011110000
130 L(3)= %1010101010101010
140 set line L(1),1,LE,LE
```

Line 50 switches of the function key window, sets the screen resolution to low, switches of colour flashing and switches of the cursor.

Lines 70-140 initialise some variables that are required throughout the program. Line 80 assigns the value one to variable I. Variable I maintains the current ink setting in the range 0-15 and is used with the *ink* and *pen* commands to set the current colour for graphics and text.

Line 90 sets the initial paint style. The variable PSTYLE is used to maintain the current paint style in the range 1-24. This is initially assigned a value of one and a *set paint* command is used to set the

paint style.

Line 100 sets the initial line style. Variable LSTYLE maintains the current line style, and as there are three different styles available, this can range from 1-3. Variable LE maintains the current line ending (0=normal, 1=arrowed) and variable array L() contains the line definitions for the three line styles. Lines 110-130 assign the three line styles to the variables. Notice that the values have been specified in binary using the % prefix. Line 140 sets the initial line style to 1.

(2) RESERVE MEMORY BANKS AND LOAD OPTION SCREEN.

```
160 rem RESERVE BANKS AND LOAD MAIN SCREEN
170 reserve as screen 14
180 reserve as screen 15
190 load "a:\art\art.pi1",15
200 get palette (15)
210 cls 14
```

Lines 170 and 180 reserve two memory banks. Memory bank 15 is used to maintain the option screen and memory bank 14 used to maintain the picture drawn by the user.

Line 190 loads the main option screen in to memory bank 15 and line 200 sets the current colour palette to match the picture.

Line 210 clears the screen held in memory bank 14 ready for the user to start drawing.

(3) DEFINE ZONES.

```
230 rem DEFINE ZONES FOR COLOUR SELECTION
240 Z=0
```

```
250 for A=12 to 282 step 18
260 inc Z
270 set zone Z,A,18 to A+12,30
280 next A
290 :
300 rem DEFINE ZONES FOR FILL STYLE SELECTION
310 set zone 17,7,43 to 17,77
320 set zone 18,139,43 to 149,77
330 :
340 rem DEFINE ZONES FOR LINE STYLE SELECTION
350 set zone 19,7,92 to 17,113
360 set zone 20,139,91 to 149,113
370 :
380 rem DEFINE ZONES FOR CHANGING COLOURS
390 set zone 21,8,124 to 40,132 : set zone 22,8,144 to 40,152
400 set zone 23,62,124 to 94,132 : set zone 24,62,144 to
94,152
410 set zone 25,118,124 to 150,132 : set zone 26,118,144 to
150,152
420 :
430 rem DEFINE ZONES FOR LINE ENDING SELECTION
440 set zone 27,9,165 to 39,184
450 set zone 28,63,165 to 93,184
460 :
470 rem DEFINE ZONES FOR OPTION SELECTION
480 Z=28
490 for A=43 to 127 step 12
500 inc Z
510 set zone Z,169,A to 239,A+10
520 inc Z
530 set zone Z,241,A to 311,A+10
540 next A
550 :
560 rem DEFINE ZONES FOR FILE OPTION SELECTION
570 for A=145 to 181 step 12
580 inc Z
```



```
590 set zone Z,169,A to 239,A+10
600 next A
```

Lines 230 - 600 define the zones. The positions of the zones were determined using the MOUSE.ACB accessory program that is supplied with STOS. This loads a picture and displays the screen coordinates of the mouse pointer as it is moved around the picture. The mouse pointer can thus be positioned at the corners of each zone and a note made of the screen coordinates.

(4) DISPLAY MAIN SCREEN AND MONITOR THE ZONES.

```
620 rem DISPLAY MAIN SCREEN + MONITOR ZONES
630 screen copy 15 to physic : screen copy 15 to back
640 show on
650 Z=zone(0)
660 if Z=0 or mouse key < > 1 then 650
670 repeat : until mouse key=0
680 on Z gosub 710, 710, 710, 710, 710, 710, 710, 710, 710,
710, 710, 710, 710, 710, 710, 800, 800, 900, 890, 1770, 1770,
1860, 1860, 1950, 1950, 1010, 1010, 1060, 690, 1140, 690, 1140,
2040, 1140, 2270, 1140, 2120, 1140, 2120, 1140, 2120, 1140, 2120,
1480, 1600, 1650, 2600
```

Line 630 copies the option screen from memory bank 15 to the PHYSICAL and BACKground screens and line 640 displays the mouse pointer.

The user can now select an option by moving the mouse pointer over the appropriate box and pressing the left mouse button.

Line 650 uses the *zone* command which checks to see if the mouse pointer is within any of the zones. If the mouse pointer is within a zone then the zone number is assigned to variable Z. If the mouse

pointer is not detected within a zone then the value zero is assigned to variable Z.

Line 660 first checks variable Z to see if the mouse pointer has been detected within a zone. If it has not been detected ($Z=0$), program execution is sent back to line 650 to check the zones again. If a zone has been detected then line 660 makes a second check to see if the left mouse button has been pressed. If it has not been pressed, program execution is sent back to check the zones again. If it has been pressed then we know that the user has selected an option and the program moves on to line 670.

Line 670 forms a *repeat..until* loop which ensures that the mouse buttons are released after the user has selected an option. The mouse button must be released to avoid any interaction with subsequent drawing operations.

Line 680 contains the *on..gosub* command and sends the program to the appropriate area depending on which zone (Z) was detected.

Let us now look at the options in detail.

(5) CHANGE THE INK COLOUR.

```
710 rem CHANGE THE INK COLOUR - ZONES 1-16
720 I=Z-1
730 ink I
740 set paint 1,1,1
750 bar 241,43 to 311,53
760 set paint 2,PSTYLE,1
770 hide on : screen copy physic to 15
780 return
```

Lines 710 - 780 change the current ink colour. The ink colour is

selected by clicking the mouse pointer on one of the colour boxes displayed across the top of the screen. These are zones 1-16 and the zone number can be used to set the ink colour direct. The ink colours range from 0-15 so we can find the appropriate ink colour by subtracting one from the zone. Line 720 therefore subtracts one from the zone number and assigns the ink colour to variable I. Line 730 uses this to set the new ink colour.

The current ink colour is displayed in the box next to the 'draw' option box, and when the colour is changed, we need to update this to the new colour. This operation is performed by lines 740 and 750. Line 740 changes the current paint style to solid and line 750 uses the *bar* command to paint the box with the new colour. Line 760 changes the current paint style back to that currently set by the user. The paint style could be set to any pattern defined by the user and hence we have to change it to solid before displaying the new ink colour. Once the bar has been drawn, we restore the original paint style.

Now that the new colour has been displayed we need to save the updated option screen back to memory bank 15. Line 770 therefore hides the mouse pointer and uses a *screen copy* command to save the new screen to memory bank 15.

Line 780 contains the *return* command which returns program execution from the sub-routine back to line 690 which in turn sends the program back to monitor the zones again.

(6) CHANGE THE PAINT STYLE.

The paint style is selected by clicking the mouse pointer on the boxes either side of the painted box. The left hand box (zone 17) decrements the paint style and the right hand box (zone 18) increments the paint style. The program allows 24 paint styles in the range 1-24.


```
800 rem CHANGE THE PAINT STYLE - ZONES 17+18
810 if Z=17 and PSTYLE > 1 then dec PSTYLE
820 if Z=18 and PSTYLE < 24 then inc PSTYLE
830 set paint 2,PSTYLE,0
840 ink 15
850 bar 19,43 to 137,77
860 ink I
870 hide on : screen copy physic to 15
880 return
```

If zone 17 (decrement paint style) has been selected, line 810 checks that the paint style (PSTYLE) is not already set at the minimum. If it is greater than one then it can be reduced by one. If zone 18 (increment paint style) has been selected, line 820 checks that the paint style (PSTYLE) has not already reached the maximum setting. If it is less than 24 then it can be increased.

Line 830 sets the new paint style using the *set paint* command.

Line 840 sets the ink colour to index 15 and line 850 uses the *bar* command to draw a box containing the new paint style.

Line 860 restores the ink colour (variable I) back to that set by the user and line 870 copies the updated option screen to memory bank 15.

(7) CHANGE THE LINE STYLE.

The program offers three line styles which are selected by clicking the mouse on the boxes either side of the line on the main display. When the user selects a new line style we need to replace the existing line with the new style.

```
900 rem CHANGE THE LINE STYLE - ZONES 19+20
```



```
910 if Z=19 and LSTYLE > 1 then dec LSTYLE
920 if Z=20 and LSTYLE < 3 then inc LSTYLE
930 set line L(LSTYLE),1,0,0
940 ink 15 : polyline 33,102 to 122,102
950 set line L(LSTYLE),1,LE,LE
960 ink I
970 hide on : screen copy physic to 15
980 return
```

If zone 19 (decrement line style) is selected, line 910 decrements the current line style (LSTYLE) provided that it is not already at the minimum setting. If zone 20 (increment line style) is selected, line 920 increments the current line style (LSTYLE) provided that it is not already at the maximum setting.

Line 930 uses the *set line* command to change the line style to the new parameters. The line endings are set to zero as we do not want to display the line endings. Line 940 sets the ink colour to 15 and displays the new line on the option screen. Line 950 then uses the *set line* command again to set the line endings back to those specified by the user.

Line 960 restores the ink colour back to that set by the user and line 970 copies the updated option screen back to memory bank 15.

(8) SET THE LINE ENDING.

This module is quite simple because no change is made to the main screen and hence memory bank 15 does not require updating.

```
1000 rem SET LINE ENDINGS - ZONES 27+28
1010 if Z=27 then LE=0 else LE=1
1020 set line L(LSTYLE),1,LE,LE
1030 return
```

Line 1010 sets the appropriate line endings depending on which zone is selected. If zone 27 (normal line endings) is selected then variable LE is set to zero else the selected zone must be zone 28 (arrowed line endings) and variable LE is set to 1.

Line 1020 uses the *set line* command to update the current line style.

(9) DRAW FREEHAND.

This option allows the user to draw on the screen using the mouse. To draw in the current ink colour the user must hold down the left mouse button. To exit back to the main option screen the user presses the right mouse button.

```
1050 rem DRAW - ZONE 29
1060 screen copy 14 to physic : screen copy 14 to back
1070 repeat
1080 if mouse key=1 then plot x mouse, y mouse
1090 until mouse key=2
1100 hide on : screen copy physic to 14
1110 return
```

Line 1060 copies the 'drawing' screen from memory bank 14 to the PHYSICal and BACKground screens.

Lines 1070-1090 form a *repeat..until* loop which repeats until the right mouse button is pressed. Line 1080 checks the condition of the left mouse button, and each time that it is pressed, a point is plotted at the current mouse pointer position in the current ink colour. When the user finishes drawing they press the right mouse button and the program exits the loop.

Line 1100 hides the mouse pointer and copies the updated drawing screen back to memory bank 14.

(10) DRAW OUTLINED AND FILLED SHAPES.

```
1130 rem DRAW SHAPES - ZONES 31 + 33 + 35 + 37 +  
39 + 41 + 43  
1140 screen copy 14 to physic : screen copy 14 to back  
1150 repeat : until mouse key  
1160 if mouse key=2 then 1430  
1170 X=x mouse : Y=y mouse  
1180 X1=X : Y1=Y  
1190 repeat : until mouse key=0  
1200 autoback off  
1210 while mouse key < > 1  
1220 if Z=31 then polyline X,Y to X1,Y1  
1230 if Z=33 then box X,Y to X1,Y1  
1240 if Z=35 then bar X,Y to X1,Y1  
1250 if Z=37 then rbox X,Y to X1,Y1  
1260 if Z=39 then rbar X,Y to X1,Y1  
1270 if Z=41 then earc X,Y,abs(X1-X),abs(Y1-Y),0,3600  
1280 if Z=43 then ellipse X,Y,abs(X1-X),abs(Y1-Y)  
1290 X1=x mouse : Y1=y mouse  
1300 while X1=x mouse and Y1=y mouse and mouse key=0  
: wend  
1310 screen copy back to physic  
1320 wend  
1330 auto back on  
1340 if Z=31 then polyline X,Y to X1,Y1  
1350 if Z=33 then box X,Y to X1,Y1  
1360 if Z=35 then bar X,Y to X1,Y1  
1370 if Z=37 then rbox X,Y to X1,Y1  
1380 if Z=39 then rbar X,Y to X1,Y1  
1390 if Z=41 then earc X,Y,abs(X1-X),abs(Y1-Y),0,3600  
1400 if Z=43 then ellipse X,Y,abs(X1-X),abs(Y1-Y)  
1410 repeat : until mouse key=0  
1420 goto 1150
```

```
1430 hide on : screen copy physic to 14  
1440 return
```

This module is quite large as it performs seven of the options within the one module.

Line 1140 displays the drawing screen.

Line 1150 uses a *repeat..until* loop to monitor the mouse buttons. The program remains in the loop until one of the mouse buttons is pressed.

Line 1160 checks to see if the right mouse button has been pressed. If the right mouse button is pressed, the program must return to the main options screen and hence program execution is sent to line 1430 where the drawing screen is copied back to memory bank 14 and the subroutine ends. If the right mouse button has not been pressed then the program moves on to line 1170.

Lines 1170-1180 assign the current mouse pointer coordinates to variables X/Y and X1/Y1.

Line 1190 forms a *repeat..until* which loops until the mouse buttons are released. This checks that the user has released the mouse button before moving on to the next stage of the program. If the user is still holding down a mouse button this could upset the next operation.

Line 1200 contains the *autoback off* command. You may recall that the graphic commands normally write to both the PHYSICal and BACKground screens, but when autoback is switched off, the commands only write to the PHYSICal screen.

Lines 1210-1320 form a *while..wend* loop which produces an expanding shape on the screen. Line 1210 initiates the loop which is continuously executed until the left mouse button is pressed. Lines 1220-1280 use the zone number to determine which shape should be

drawn. Each shape uses the values of the X/Y and X1/Y1 variables to determine the current size. Look carefully at lines 1270 and 1280. The *abs* command has been used to ensure that the coordinates are positive.

Line 1290 assigns the current mouse pointer coordinates to variables X1/Y1.

Line 1300 contains a *while..wend* loop which monitors the position of the mouse pointer and the status of the mouse buttons. If the current mouse pointer coordinates (*x mouse* and *y mouse*) are equal to the variables X1/Y1 then the mouse has not moved. The program therefore continues in the loop until either the mouse pointer is moved or a mouse button is pressed. When the pointer is moved or a button is pressed, the program moves on to line 1310

Line 1310 copies the BACKground screen to the PHYSICal screen. Remember that the shape was only drawn on the PHYSICal screen and is thus not on the BACKground screen. Copying BACK to PHYSIC therefore removes the shape from the screen.

The *wend* command at line 1320 sends program execution back to line 1210 where the left mouse button is checked again. If the mouse button has been pressed, the program moves on to line 1330 where the relevant shape is permanently written to the screen (*autoback on* restores normal graphics operation). If the left mouse button has not been pressed, the shape is re-drawn and the whole process starts again.

If you are still unsure about this section then refer back to the 'Expanding Box' section in chapter 14.

(11) LOAD PICTURE FROM DISK.

When a file is loaded, it is displayed on the screen for the user to see. The main option screen is restored by pressing the right mouse button.

```
1460 rem LOAD PICTURE - ZONE 45
1470 F$=file select$("*.*.pi1")
1480 if F$="" then 1560
1490 hide on
1500 load F$
1510 repeat : until mouse key=2
1520 screen copy physic to 14
1530 screen copy 15 to physic : screen copy 15 to back
1540 get palette (14)
1550 screen copy physic to 15
1560 return
```

When a picture file is loaded, we display the picture on the screen, save the picture to memory bank 14 and update the main option screen to match the colour palette of the loaded picture.

Line 1470 displays a file selector so that the user may select the required file for loading. The selected filename is assigned to variable F\$.

If a file has not been selected, line 1480 sends the program to line 1560 where the subroutine ends and the operation terminates.

Line 1490 hides the mouse pointer and line 1500 loads the file. If you can remember back to the chapter on screens, you will recall that the picture file is loaded to both the PHYSICAL and BACKground screens and that the current colour palette changes to match the picture.

Line 1510 monitors the right mouse button and continues in a loop until the button is pressed. This allows the user time to see the

picture.

The picture can now be saved to memory bank 14 where it will be maintained for further editing. This operation is performed by line 1520.

Now that the picture is safely stored, we can re-display the main option screen. The current colour palette may now be different to that of the main option screen stored in memory bank 15, and although the screen will be displayed in the new colours, we need to update memory bank 15. Line 1530 therefore displays the main option screen, line 1540 gets the palette of the picture in memory bank 14 and line 1150 copies the updated option screen back to memory bank 15. The main option screen is now stored with the new colour palette.

(12) SAVE PICTURE TO DISK.

```
1580 rem SAVE PICTURE TO DISK - ZONE 46
1585 hide on
1590 screen copy 14 to physic
1595 get palette(15)
1610 screen copy physic to 14
1620 show on
1630 F$=file select$("*.*.pil")
1640 if F$ < > "" then save F$,14
1650 return
```

Before saving the picture file in memory bank 14, we have to ensure that it is stored with the current colour palette. The main option screen offers a number of options for modifying the colour palette and thus the current colour palette may be different to that stored in memory bank 14. Line 1590 therefore displays the picture, line 1595 calls the current colour palette from memory bank 15 and line 1610 saves the picture file back to memory bank 14 in the new colours. The picture

file can now be saved to disk.

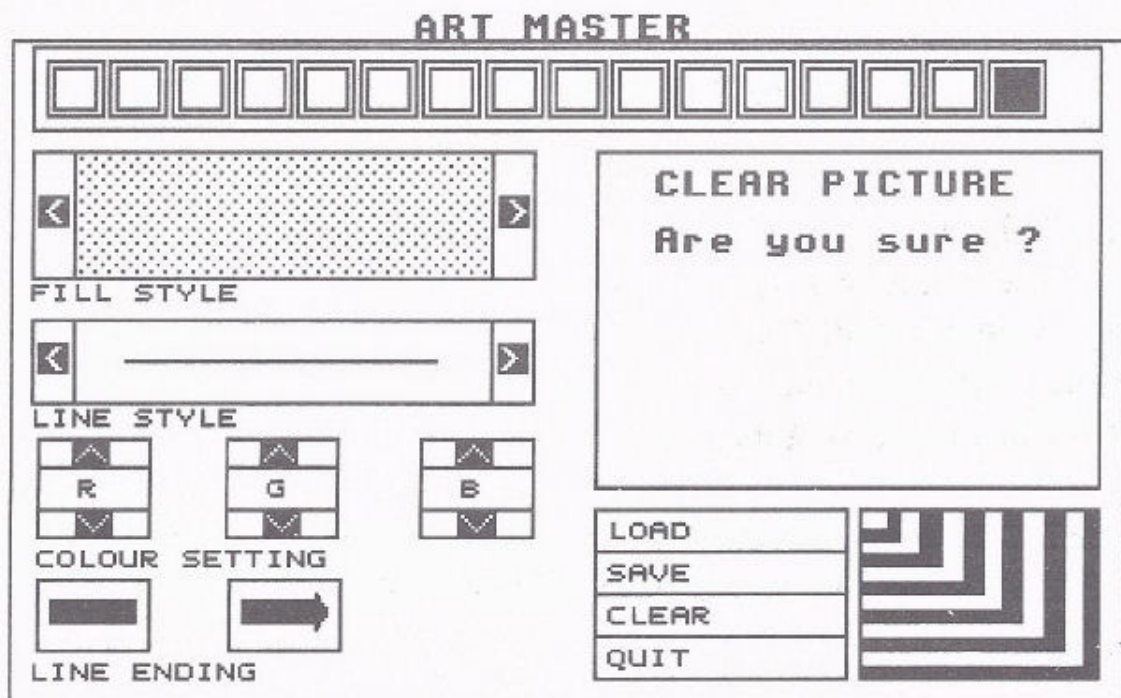
Line 1620 displays the mouse pointer and line 1630 displays a file selector so that the user may enter or select a file name for the file.

Line 1640 checks that a valid filename has been entered, and if so saves the file.

Line 1650 ends the subroutine and passes control back to the calling program.

(13) CLEAR THE CURRENT PICTURE.

When a picture is cleared it is erased from memory and is not recoverable. The user is therefore prompted to confirm their request before erasing the picture. The option box is used to display the warning message as shown below:




```
1670 rem CLEAR CURRENT PICTURE - ZONE 47
1680 limit mouse 0,0 to 1,1 : hide on
1690 ink 0 : bar 169,43 to 311,137
1700 pen 15
1710 locate 23,6 : print "CLEAR PICTURE"
1720 locate 23,8 : print "Are you sure ?"
1730 Q$=input$(1) : Q$=upper$(Q$)
1740 if Q$="Y" then cls 14 : flash off
1750 ink I
1760 limit mouse
1770 return
```

Line 1680 limits mouse movement to the top left hand corner of the screen and hides the mouse pointer. This ensures that the user does not activate any other options.

Line 1690 sets the ink colour to index zero and draws a filled box using the *bar* command. This replaces the list of options with a blank space. Line 1700 changes the pen to colour index 15.

Lines 1710-1720 display a message asking the user to confirm their intentions.

Line 1730 waits for the user to input a character and converts this to upper case. If the user enters 'Y' for YES, line 1740 clears the screen held in memory bank 14 (the drawing screen) and switches off colour flashing. Note that the *cls* command re-instates colour flashing and hence it has to be switched off again.

Line 1750 restores the current ink colour setting, line 1760 restores movement of the mouse and line 1770 returns control to the calling program.

(14) CHANGE THE RED COLOUR ELEMENT.

We have already seen that the easiest way to define colours is to use hexadecimal format with values in the range 0-7 representing the strength of each of the primary colours red, green and blue. Considering the colour as a hexadecimal number, we can easily change the relevant setting.

```
1790 rem CHANGE COLOUR CONTENT RED -  
ZONES 21+22  
1800 C=colour(I)  
1810 C$=hex$(C,3)  
1820 R$=mid$(C$,2,1)  
1830 if Z=21 and R$ < > "7" then C=C+$100  
1840 if Z=22 and R$ < > "0" then C=C-$100  
1850 colour I,C  
1860 hide on : screen copy physic to 15  
1870 return
```

Variable I maintains the current ink colour and line 1800 uses this with the *colour()* command to assign the setting of the current colour to variable C.

Line 1810 converts the colour information to hexadecimal format and assigns this to variable C\$. The variable C\$ now contains the current colour setting in the format \$RGB. Line 1820 then extracts the second character (the red setting) from variable C\$ and assigns this to variable R\$.

We can now increment or decrement the value of the red colour setting depending on which zone was selected. Lines 1830 and 1840 carry out this operation. Line 1830 checks to see if the increment box was selected, and provided the setting is not already at the maximum, increments the colour setting by adding \$100 to it. Line 1840 checks to see if the decrement box was selected, and provided the red colour

setting is not already at the minimum, decrements it by subtracting \$100.

All that remains now is to actually change the colour to the new setting and this is carried out by line 1850. You may recall that colour changes take immediate effect and the colour will thus change in the colour boxes along the top of the screen. Whenever the main option screen is changed in this way, it must be copied back to memory bank 15. Line 1860 performs this function and line 1870 returns control to the calling program.

(15) CHANGE THE GREEN COLOUR ELEMENT.

```
1890 rem CHANGE COLOUR CONTENT GREEN -  
ZONES 23+24  
1900 C=colour(I)  
1910 C$=hex$(C,3)  
1920 G$=mid$(C$,3,1)  
1930 if Z=23 and G$ < > "7" then C=C+$10  
1940 if Z=24 and G$ < > "0" then C=C-$10  
1950 colour I,C  
1960 hide on : screen copy physic to 15  
1970 return
```

This is similar to the previous routine but this time we need to change the green colour element.

Line 1900 assigns the current colour setting to variable C and line 1910 converts this to hexadecimal format. Line 1920 then extracts the third character (the green setting) from variable C\$ and assigns this to variable G\$.

Lines 1930-1940 increment or decrement the green setting by adding or subtracting \$010 to the colour setting depending on which box was

selected.

Line 1950 sets the new colour. As usual, the colour change takes immediate effect and hence the main option screen is changed. This is therefore copied to memory bank 15 before the subroutine ends.

(16) CHANGE THE BLUE COLOUR ELEMENT.

```
1990 rem CHANGE COLOUR CONTENT BLUE -  
ZONES 25+26  
2000 C=colour(I)  
2010 C$=hex$(C,3)  
2020 B$=right$(C$,1)  
2030 if Z=25 and B$ < > "7" then C=C+$1  
2040 if Z=26 and B$ < > "0" then C=C-$1  
2050 colour I,C  
2060 hide on : screen copy physic to 15  
2070 return
```

This is the same as the previous two routines except that the blue colour setting is extracted and changed.

(17) PAINT AN AREA OF THE SCREEN.

```
2090 rem PAINT - ZONE 34  
2100 screen copy 14 to physic : screen copy 14 to back  
2110 repeat  
2120 if mouse key=1 then paint x mouse, y mouse  
2130 until mouse key=2  
2140 hide on : screen copy physic to 14  
2150 return
```

An area of the screen is painted by placing the mouse pointer at a

point within the desired area and pressing the left mouse button.

Line 2100 displays the drawing screen.

Lines 2110-2130 form a *repeat..until* loop which continues until the right mouse button is pressed. Line 2120 checks the status of the mouse buttons, and if the left mouse button is pressed, the screen is painted. Notice how the *xmouse* and *ymouse* functions are used to indicate the area to be painted.

When the right mouse button is pressed, the program moves on to line 2140 where the drawing screen is copied back to memory bank 14 and control is passed back to the calling program.

(18) DISPLAY TEXT.

```
2170 rem DISPLAY TEXT - ZONES 38,40,42,44
2180 screen copy 14 to physic : screen copy 14 to back
2190 pen I
2200 repeat : until mouse key
2210 if mouse key=2 then 2280
2220 locate xtext(x mouse),ytext(y mouse)
2230 if Z=38 then line input "":T$
2240 if Z=40 then under on : line input "":T$ : under off
2250 if Z=42 then inverse on : line input "":T$: inverse off
2260 if Z=44 then shade on : line input "":T$ : shade off
2270 goto 2200
2280 hide on : screen copy physic to 14
2290 return
```

The program can produce four types of text - normal, underlined, inverted and shaded.

Line 2180 displays the drawing screen.

Line 2200 contains a *repeat..until* loop which continues until a mouse key is pressed. When a mouse key is pressed, the program passes on to line 2210

Line 2210 checks to see if the right mouse button has been pressed. If so, the program jumps to line 2280 where the screen is copied to memory bank 14 and control is returned to the calling program.

If the left mouse button is pressed, the program moves on to line 2220 where the text cursor is located at the current mouse pointer position. Notice the *xtext* and *ytext* functions that are used to convert the graphics coordinates of the mouse pointer to text coordinates.

Lines 2230 to 2260 switch on the appropriate text effect and wait for the user to input the text. The zone number is used to determine which effect the user selected from the main menu.

(19) CUT A BLOCK

```
2310 rem CUT BLOCK - ZONE 36
2320 screen copy 14 to physic : screen copy 14 to back
2330 auto back off
2340 repeat : until mouse key
2350 if mouse key=2 then 2600
2360 X=x mouse : Y=y mouse
2370 X1=X : Y1=Y
2380 repeat : until mouse key=0
2390 while mouse key < > 1
2400 box X,Y to X1,Y1
2410 X1=x mouse : Y1=y mouse
2420 while X1=x mouse and Y1=y mouse and
mouse key=0: wend
2430 screen copy back to physic
2440 wend
```



```
2450 repeat : until mouse key=0
2460 if X>X1 then swap X,X1 : swap Y,Y1
2470 if Y>Y1 then swap Y,Y1
2480 S$=screen$(physic,X,Y to X1,Y1)
2490 while mouse key < > 1
2500 screen$(physic,x mouse,y mouse)=S$
2510 X1=x mouse : Y1=y mouse
2520 while X1=x mouse and Y1=y mouse and
mouse key=0: wend
2530 screen copy back to physic
2540 wend
2550 auto back on
2560 screen$(physic,X1,Y1)=S$
2570 screen$(back,X1,Y1)=S$
2580 repeat : until mouse key=0
2590 goto 2330
2600 hide on : screen copy physic to 14
2610 return
```

This is the most involved of all the routines so study it carefully. There was some debate during the design of this course whether such a routine should be included, but it was decided that the routine should be included to illustrate further the powerful screen commands offered by STOS. Let us take a look.

The routine allows the user to draw a box around an area of the screen and then copy this block to another area or areas of the screen. The routine is operated as shown below:

(a) Position the mouse pointer at the top left corner of the block you wish to cut and press the left mouse button.

(b) An expanding rectangle is displayed and this is stretched over the required area before pressing the left mouse button again.

(c) The block is now cut and can be moved around using the mouse. To place the block on the screen you just press the left mouse button.

(d) To exit the routine you press the right mouse button.

So let us take a look at how the module works.

Line 2320 displays the drawing screen.

Line 2330 switches autoback off. This means that any further graphics commands will only write to the PHYSICAL screen and not the BACKGROUND screen.

Line 2340 waits for the user to press a mouse button.

Line 2350 checks to see if the right mouse button has been pressed. If it has been pressed, the program jumps to line 2600 where the drawing screen is copied back to memory bank 14 and the block routine ends. If the left mouse button is pressed, the program moves on to line 2360.

Line 2360-2370 assign the current mouse pointer coordinates to variables X/Y and variables X1/Y2.

Line 2380 ensures that the mouse button has been released. If this is not included, the next part of the program may assume that the mouse button has been pressed again.

Lines 2390-2440 form a *while..wend* loop which produces an expanding box until such time that the left mouse button is pressed. Line 2400 draws the box, which due to autoback being off, is only drawn on the PHYSICAL screen. Line 2410 assigns the current mouse pointer coordinates to variables X1/Y1 and line 2420 checks these against the current mouse pointer position to see if the mouse has moved. If the mouse has moved, the program moves on to line 2430

else the loop continues until the mouse pointer is moved.

When the mouse pointer is moved we need to expand the rectangle. Line 2430 therefore copies the BACKground screen to the PHYSICal screen which removes the current rectangle (remember that it was originally only displayed on the PHYSICal screen and not the BACKground screen). The program then loops back to line 2390, and provided the left mouse button is not pressed, line 2400 re-draws the rectangle at the new size.

This expanding rectangle loop continues until the left mouse button is pressed. When the left mouse button is pressed, we have to 'cut' the block from the screen and this is performed by lines 2450 onwards.

Line 2450 ensures that the mouse button has been released.

The block is cut using the *screen\$* command which allows a section of screen to be assigned to a variable. The *screen\$* command requires the coordinates in a specific form, but we are not sure how the user stretched the rectangle over the block, and thus the coordinates could be in the wrong order. Lines 2460-2470 check the coordinates, and if necessary, swap them around. This ensures that X/Y represent the top left corner of the block and X1/Y1 represent the bottom right corner of the block. Line 2480 then cuts the block and assigns it to variable S\$.

Lines 2490-2540 allow the block to be moved around the screen using the mouse. This uses the same technique as that previously covered for the expanding rectangle and continues until the left mouse button is pressed. When the left mouse button is pressed, the block is permanently placed on both the PHYSICal and BACKground screens.

Line 2550 switches autoback on, lines 2560-2570 place the block on both the screens, line 2580 waits for the mouse button to be released and line 2590 sends the program back to line 2330 so that the user can

cut another block if required.

Well that was very heavy going so do not worry if you do not fully understand it. Try cutting a few blocks and think about what the program is actually doing and it will soon click.

(20) QUIT THE PROGRAM.

```
2630 rem QUIT - ZONE 48
2640 limit mouse 0,0 to 1,1 : hide on
2650 ink 0 : bar 169,43 to 311,137
2660 pen 15
2670 locate 23,6 : print "QUIT ARTMASTER"
2680 locate 23,8 : print "Are you sure ?"
2690 Q$=input$(1) : Q$=upper$(Q$)
2700 if Q$="Y" then cls : print "BYE BYE" : end
2710 limit mouse
2720 ink I
2730 return
```

This uses the same technique as that of the CLEAR PICTURE module where the user is prompted to confirm their request before the action is carried out.

Line 2640 hides the mouse pointer and limits its movement to the top left hand edge of the screen. This ensures that the user does not try and select another option.

Line 2650 clears the area of the screen containing the option boxes and lines 2670-2690 ask the user to confirm their request by pressing the letter "Y". Notice the use of the *input\$* command to input one character from the user. If the user enters "Y" then line 2700 ends the program. If the user enters anything other than "Y", the program moves on to line 2710 where mouse pointer movement is restored, the

current ink colour is restored and program execution passes back to the calling program.

That brings us to the end of Art Master. The majority of the program is quite straightforward but the 'expanding shape' routine and the 'block cutting' routine are quite tricky so spend some time studying them. When studying a programs operation it can often help to run the program and compare its operation with the printed listing. When looking through a program listing you may find a line of, or a section of code which cannot be related to the programs operation. In this situation remove the code and run the program again to see what effect it has. Remember that programming involves a lot of experimentation. You cannot not harm the computer by removing or adding sections of code to existing programs; in fact this is the best way to learn.

Chapter 17

Sprites & Animation

We have already established that a sprite is a small graphical object that can move around the screen without effecting any other information that may be displayed. To understand the effect of this consider the mouse pointer which is itself a sprite. STOS allows 15 sprites to be displayed, moved and animated simultaneously and the real power lies in the fact that all of these operations are carried out under interrupt. This means, that once defined, the sprite operations are carried out automatically in the background while our program continues with other things.

THE SPRITE BANK

Sprite data is always held in memory bank 1 and thus we do not need to specify the memory bank when loading such information. Memory bank 1 is classified as a permanent bank which means that the information within it will be saved along with the program. Subsequent loading of the program will result in the sprite information being loaded as well. Once a program has been loaded we can check to see if any sprite information is loaded by listing the program. When

a program is listed, all of the assigned or used banks are shown at the end.

Sprites are stored in the memory bank one after the other and each sprite is identified by a number which represents its position within the bank. The bank may contain as many sprites as memory will allow. Supplied with this course are three sets of sprites named INVADERS.MBK, ALIEN.MBK and SPOOK.MBK. Let's have a look at one of these sets. The INVADERS sprite set is in a folder named SPRITES so load it by typing the following:

```
load "a:\sprites\invaders.mbk"
```

The file extension .MBK indicates that the information is memory bank information. The data is identified as being sprite information and is loaded straight in to memory bank 1. We can check that the information has been loaded by listing the program:

```
list
```

```
Reserved memory banks:
```

```
1  sprites S:$0D6C00 E:$0D8200 L:001600
```

DISPLAYING SPRITES

Now that the sprite information is in memory we can display the sprites on screen using the *sprite* command. The INVADERS sprite set contains a set of space ship type sprites that have been designed for low resolution, so let's change to low resolution and display one of the sprites.

Enter the following:

```
mode 0
```

```
flash off  
sprite 1, 100, 100, 11
```

The *sprite* command instructs the computer to display a sprite on the screen and the format of the command is as follows:

```
sprite N, X, Y, SPR
```

N is an index number in the range 1 to 15 and this is used to identify the sprite in subsequent operations.

X and Y indicate the coordinates for the position of the sprite on the screen. These values, unlike other commands, can be negative and within the following ranges:

X: -640 to +1280

Y: -400 to +800

This allows the sprites to be moved off the screen and effectively move around the back of the screen. Do not worry about this yet as all will be revealed later.

SPR indicates the number of, or the position of the sprite within the sprite bank.

Looking back to the last example:

```
SPRITE 1,100,100,11
```

This assigns the sprite in the memory bank at position 11 to sprite index number 1 and displays this at coordinates 100,100

Take a look at another couple of the sprites by typing the following:

```
sprite 2,100,140,13  
sprite 3,100,180,16
```


The sprites can be switched off or removed from the screen using the *sprite off* command.

Enter the following:

sprite off 3

This removes sprite number 3 from the screen. Now enter the following:

sprite off

This removes the two remaining sprites from the screen. When the sprite number is omitted, the *sprite off* command removes all of the sprites from the screen.

CREATING SPRITES

You may be wondering how we create sprites in the first place and what determines their position within the sprite bank. STOS comes with a sprite designer which is loaded as an accessory and allows sprites to be either designed from scratch or grabbed from Degas and Neochrome format picture files. The STOS User Guide covers the drawing and design facilities in detail and thus there is no point reproducing them here. We shall though cover a couple of points that are especially important to the programmer.

The sprite designer is loaded as follows:

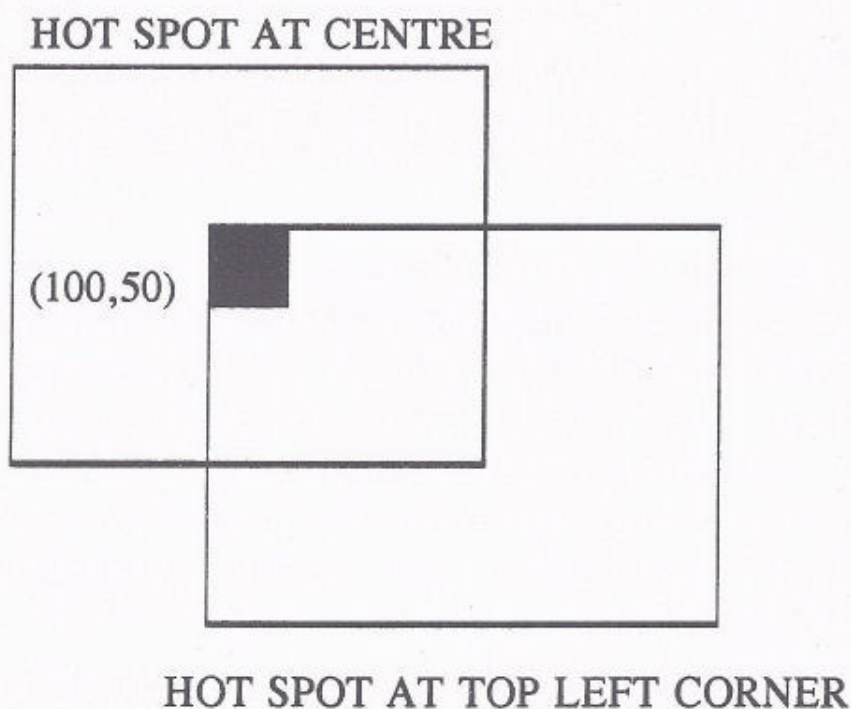
accload "sprite.acb"

so load it, and with the aid of the STOS User Guide, have a look around. Load the INVADER.MBK sprite set and you will be able to look at, and if you wish, modify the sprites.

SPRITE HOT SPOTS

You will soon discover that the sprite designer is a very powerful tool. In addition to the general drawing commands there are a host of other features for placing sprites at specific positions within the sprite bank, checking animation, etc. One area of particular importance is the definition of the *hot spot*. Whenever we display a sprite on the screen (using the *sprite* command) we specify its coordinates in terms of the position of the *hot spot* on the screen. The hot spot menu allows the hot spot to be placed at various positions within the sprite. As default the hot spot is positioned at the top left hand corner of the sprite, but this can be changed to the centre or any other position as required.

Consider the command *sprite 1,100,50,1*. This would place a sprite at the screen coordinates (100,50). If the hot spot was at the top left corner of the sprite then the top left corner of the sprite would be placed at coordinates (100,50). If the hot spot was at the centre of the sprite then the centre of the sprite would be placed at coordinates (100,50). The following diagram illustrates this:



As can be seen, the hot spot makes a considerable difference to the position at which the sprite will be placed. You can see the various hot spot positions for the INVADER sprite set using the sprite definer.

SPRITE COLOUR PALETTE

The colour palette is an area that requires careful consideration when sprites are involved. Remember that all of the sprites will share the same colour palette as any pictures, backdrops, etc.

The sprites in our last examples may have looked a little strange. This is because they were displayed using the current colour palette rather than the colour palette that they were designed for. There are three methods that can be used to set the colour palette when using sprites:

- (1) Set palette within the program.
- (2) Grab palette from a screen type memory bank.
- (3) Grab palette from the sprite memory bank.

(1) The first method is to use the *palette* command at the start of the program. This will define all of the colours to be used by the sprites and pictures within the program.

(2) Setting the palette within the program is ok but many games rely on sprites moving around on some sort of background picture. We know that the sprites and all picture files must share the same colour palette and hence we could use the *get palette* command to grab the colour information from a picture file stored within a memory bank. Although STOS offers powerful graphic commands and an excellent sprite definer, most programmers use an art package for designing background pictures. An art package offers powerful drawing facilities and also allows the colour palette to be set as required. Once complete, the picture can be loaded in to a memory bank and the

current colour palette changed using the *get palette* command.

(3) Unfortunately the *get palette* command only works with screen type memory banks and cannot be used to grab colour information from the sprite bank. Therefore, when no external picture files are used and if you do not want to set the palette at the start of the program, you can grab the colour information from the sprite bank using the small program shown below:

```
10 rem GRAB SPRITE BANK PALETTE
20 A=hunt(start(1) to start(1)+length(1),"PALT")
30 A=A+4
40 for B=0 to 15
50 colour B, deek(B*2+A)
60 next B
```

The program searches the sprite bank for the colour palette information and changes the current palette to suit this. Do not worry about the operation of the program but simply 'bolt' it in to your own programs whenever you wish to grab the colour palette from the sprite bank.

The method used to set the colour palette will depend on the type of program and the style of sprites and screens required, but when a program uses a background screen, the easiest method is to use the *get palette* command.

SPRITE ANIMATION

We saw in a previous chapter how the *screen copy* command can be used to produce animation. This technique was also used for the Bonk the Gonk game which relied on copying various frames to the same section of the screen to make the Gonks move in and out of the holes. We can also produce animation using sprites and these offer a lot

more scope than the screen copy method. Sprite animation uses much the same technique of copying a number of frames or sprites to a common position, but once defined, the animation takes place automatically in the background leaving our program to do other things.

Load the following:

```
10 rem SPRITE ANIMATION
20 rem PROGRAM = A:\SPRITES\ANIM1
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off : hide on
60 palette $0, $666, $444, $0, $322, $334, $112,
$700, $100, $433, $64, $113, $750, $211, $1, $555
70 :
80 rem DISPLAY SPRITES
90 sprite 1,50,100,16
100 sprite 2,100,100,17
110 sprite 3,150,100,17
120 :
130 rem ANIMATE SPRITE
140 anim 3,"(16,25)(17,25)L"
150 anim on
```

The sprites have been saved with the program, but if you are typing the program in direct, do not forget to load the sprites before running the program.

The program displays three space ship type sprites across the screen. The lights on the last space ship appear to be flashing and this effect is achieved by switching or animating the sprite. The first two sprites show the images that are used to produce the effect. The first sprite has the lights off and the second sprite has the lights on. The last sprite is switched between these two sprites every half a second to

produce the effect of flashing lights.

Line 50 switches off the function key window, selects low screen resolution, switches off colour flashing and hides the mouse pointer.

Line 60 sets the colour palette.

Lines 90-110 display three sprites across the screen. Sprite 1 represents the sprite at memory bank position 16, sprite 2 represents the sprite at memory bank position 17 and sprite 3 represents the sprite at memory bank position 17. Notice that sprites 2 and 3 are the same.

Line 140 introduces the *anim* command. The *anim* command allows us to animate or move through a series of sprites one after the other. Following the command we specify the index number of the sprite to be animated followed by the animation data. Each animation step is placed within brackets, the first number indicating the memory bank position of the sprite to be displayed and the second number indicating the period of time for which the sprite should be displayed. The time, as usual, is specified in 50ths of a second. Look closely at line 140:

```
140 anim 3, "(16,25)(17,25)L"
```

This defines the animation sequence for the sprite represented by index number 3. It displays the sprite at memory bank position 16 for 25/50ths of a second (half a second) and then switches to the sprite at memory bank position 17 for half a second. The letter "L" at the end of the line stands for 'loop' and causes the animation sequence to be repeated. Sprite index 3 is therefore changed backwards and forwards between sprites 16 and 17 every half a second. If the letter "L" is omitted, the animation sequence will stop at the end and not repeat.

The *anim* command only defines the animation sequence and does not actually initiate the animation process. To start animation we have to

switch it on using the *anim on* command as shown in line 150. This activates all of the animation sequences previously defined with the *anim* command but can also be directed to a single animation sequence by placing the relevant sprite number at the end of the command. For example:

```
anim on 3
```

would start the animation sequence for sprite index 3 without affecting any other sprite sequences that may have been defined.

When dealing with sprites it is very easy to confuse the sprite index number and the sprite position within the memory bank. Looking back at the *anim* command, the sprite to be animated is specified by the main sprite index number (in the range 1 to 15) whilst the list of sprites for animation are specified as sprite positions within the memory bank.

To further illustrate the principle let us use some other sprites from the sprite bank.

Load the following:

```
10 rem SPRITE ANIMATION
20 rem PROGRAM = A:\SPRITES\ANIM2
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off : hide on
60 palette $0,$666, $444, $0, $322, $334, $112,
$700, $100, $433, $64, $113, $750, $211, $1, $555
70 :
80 rem DISPLAY SPRITES
90 sprite 1,50,100,11
100 sprite 2,100,100,12
110 sprite 3,150,100,12
```

```
120 :  
130 rem ANIMATE SPRITE  
140 anim 3,"(11,10)(12,10)L"  
150 anim on
```

The programs operation is the same as the previous example and all that has changed is the sprites themselves.

This time a different space ship is displayed and the images are animated to give the effect of a rotating centre.

SPRITE MOVEMENT - HORIZONTAL

We shall look at more examples of animation in a moment, but for now, let's see how to make the sprites move around the screen. STOS offers two commands that allow sprites to be moved from left to right or right to left across the screen (X axis), and from top to bottom or bottom to top of the screen (Y axis). We mentioned earlier that the sprites operations are carried out under interrupt and the movement commands are no different. We can therefore move the sprites and animate the sprites whilst our program continues on with something else. Let us take one of the space ships from the last example and make it fly across the top of the screen.

Load the following:

```
10 rem SPRITE MOVEMENT  
20 rem PROGRAM = A:\SPRITES\MOVE1  
30 :  
40 rem SET SCREEN  
50 key off : mode 0 : flash off  
60 :  
70 rem SET COLOUR PALETTE  
80 palette $0, $666, $444, $0, $322, $334, $112,
```


\$700, \$100, \$433, \$64, \$113, \$750, \$211, \$1, \$555

90 :

100 rem PLACE SPRITE ON SCREEN

110 sprite 1,15,50,11

120 :

130 rem DEFINE SPRITE MOVEMENT

140 move x 1,"(1,3,90)"

150 :

160 rem SWITCH ON MOVEMENT

170 move on

The program makes the space ship fly across the screen and stop.

The main area of interest is line 140 which defines the movement. Notice that this command is similar to the *anim* command where the various sequences are placed in brackets. The format of the *move x* command is shown below:

move x SPRITE, "(SPEED, STEP, COUNT)"

where SPRITE indicates the index number of the sprite to be moved in the range 1-15.

The movement itself is expressed as three elements, SPEED, STEP and COUNT. We have to tell STOS how many times we want the sprite to be moved (COUNT), how many pixels we want it to move each time (STEP) and how much time we want to wait between each step of the movement (SPEED). To fully understand this take a look at the movement command in the previous program.

140 move x 1,"(1,3,90)"

The above line will move sprite number 1 as defined in the movement string (1,3,90). This indicates that the movement will consist of 90 steps each of which will move the sprite 3 pixels. The time between

each movement step will be 1/50th of a second.

The speed at which the sprite moves is therefore determined by the SPEED and STEP parameters. Try experimenting with these to see the effect that they have.

Now that the movement has been defined we have to switch it on using the *move on* command. As per the *anim on* command, we can also add a sprite index number to the end of the command to switch on movement for a single sprite rather than all the sprites. For example:

```
move on 8
```

would start sprite 8 moving without affecting any other movement that may have been defined.

Load the following:

```
10 rem SPRITE MOVEMENT
20 rem PROGRAM = A:\SPRITES\MOVE2
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off
60 :
70 rem SET COLOUR PALETTE
80 palette $0, $666, $444, $0, $322, $334, $112,
$700, $100, $433, $64, $113, $750, $211, $1, $555
90 :
100 rem PLACE SPRITE ON SCREEN
110 sprite 1,15,50,11
120 :
130 rem DEFINE SPRITE MOVEMENT
140 move x 1,"(1,3,90)(1,-3,90)"
150 :
```


160 rem SWITCH ON MOVEMENT**170 move on**

Run the program and you will see that the space ship now flies across the screen and then back again. The program is exactly the same as the previous program except for line 140 which has been extended.

```
140 move x 1,"(1,3,90)(1,-3,90)"
```

Notice that the step size on the second movement element is set to -3. A negative value makes the sprite move backwards from right to left across the screen.

We saw, during the section on sprite animation, how the letter "L" could be added to the end of the *anim* command to produce looping. This can also be applied to the movement commands to produce continuous movement.

Change line 140 to the following:

```
140 move x 1,"(1,3,90)(1,-3,90)L"
```

If you prefer, load the modified program from disk. It is named "a:\sprites\move3"

Run the program and you will see the space ship fly backwards and forwards across the screen.

SPRITE MOVEMENT - VERTICAL

The last examples used the *move x* command to move the space ship from side to side across the screen (the X axis), but we can also produce movement up and down the screen (the Y axis) using the matching *move y* command.

Load the following:

```
10 rem SPRITE MOVEMENT - VERTICAL
20 rem PROGRAM = A:\SPRITES\MOVE4
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off
60 :
70 rem SET COLOUR PALETTE
80 palette $0, $666, $444, $0, $322, $334, $112,
$700, $100, $433, $64, $113, $750, $211, $1, $555
90 :
100 rem PLACE SPRITE ON SCREEN
110 sprite 1,100,20,16
120 :
130 rem DEFINE SPRITE MOVEMENT
140 move y 1,"(1,1,150)"
150 :
160 rem SWITCH ON MOVEMENT
170 move on
```

Run the program.

The program makes the sprite move down the screen. Look closely at line 140.

```
140 move y 1,"(1,1,150)"
```

The movement will consist of 150 steps each of which will move the sprite 1 pixel. The time between each movement step will be 1/50th of a second.

Load the following:

```
10 rem SPRITE MOVEMENT - VERTICAL
```



```
20 rem PROGRAM = A:\SPRITES\MOVE5
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off
60 :
70 rem SET COLOUR PALETTE
80 palette $0, $666, $444, $0, $322, $334, $112,
$700, $100, $433, $64, $113, $750, $211, $1, $555
90 :
100 rem PLACE SPRITE ON SCREEN
110 sprite 1,100,20,16
120 :
130 rem DEFINE SPRITE MOVEMENT
140 move y 1,"(1,1,150)(1,-1,150)I"
150 :
160 rem SWITCH ON MOVEMENT
170 move on
```

Run the program and the space ship will keep moving up and down the screen. The program is exactly the same as the previous example except for line 140 which has been extended to include negative movements steps and looping.

SPRITE MOVEMENT - DIAGONAL

The *move x* and *move y* commands are used to move sprites either horizontally or vertically but they can also be combined to produce diagonal movement.

Load the following:

```
10 rem SPRITE MOVEMENT - DIAGONAL
20 rem PROGRAM = A:\SPRITES\MOVE6
30 :
```

```
40 rem SET SCREEN
50 key off : mode 0 : flash off
60 :
70 rem SET COLOUR PALETTE
80 palette $0, $666, $444, $0, $322, $334, $112,
$700, $100, $433, $64, $113, $750, $211, $1, $555
90 :
100 rem PLACE SPRITE ON SCREEN
110 sprite 1,50,100,16
120 :
130 rem MOVE SPRITE
140 move x 1,"(1,1,100)(1,-1,100)I"
150 move y 1,"(1,-1,50)(1,1,50)I"
160 move on
```

Run the program and the sprite will move diagonally on the screen.

Look closely at lines 140 and 150 to work out the movement pattern.

COMBINING SPRITE MOVEMENT AND ANIMATION

The last few sections have illustrated how sprites can be animated and moved around the screen. Remember that these commands operate under interrupt, and thus once defined, are handled by STOS with no further help from the programmer. If the movement and animation techniques are not enough by themselves, we can also combine them to produce simultaneous animation and movement.

Load the following:

```
10 rem MOVEMENT AND ANIMATION COMBINED
20 rem PROGRAM = A:\SPRITES\ANIM3
30 :
40 rem SET SCREEN
```



```
50 key off : mode 0 : flash off
60 :
70 rem SET COLOUR PALETTE
80 palette $0, $666, $444, $0, $322, $334, $112,
$700, $100, $433, $64, $113, $750, $211, $1, $555
90 :
100 rem PLACE SPRITE ON SCREEN
110 sprite 1,300,50,13
120 :
130 rem DEFINE MOVEMENT AND ANIMATION
140 move x 1,"(1,-3,90)"
150 anim 1,"(13,5)(14,5)(15,5)L"
160 :
170 rem SWITCH ON MOVEMENT AND ANIMATION
180 move on
190 anim on
```

Run the program and you will see a space craft fly across the screen with a moving jet at the back.

Line 110 places the sprite at the right hand edge of the screen.

Line 140 defines the movement which will make the space craft fly across from the right hand side of the screen to the left hand side of the screen.

Line 150 defines the animation sequence. Notice that three sprites are used and each is displayed for 5/50ths of a second.

Lines 180 and 190 activate the movement and animation.

Well that just about covers the basics of sprite movement and animation. You would probably agree that the sprite handling facilities offered by STOS really are very impressive. Sprites are an essential part of many games and this is probably why STOS has become

known as 'the language' for writing games.

The most popular type of game, and one which many programmers seem intent on learning about, is the shoot-em-up. For some strange reason a great deal of satisfaction can result from blasting away computer generated nasties as they zap at high speed around the screen. The next chapter thus concentrates on some techniques that might be used to produce such games.

Chapter 18

Guns, Aliens & Collisions

This chapter is devoted to shoot-em-up games. We shall see how to control guns, make them fire missiles, detect collisions with enemy space craft, produce animated explosions, etc.

THE GUN

Many shoot-em-up type games are based on a gun that moves around at the bottom of the screen firing up at enemy craft flying around above. Classics such as Space Invaders and Galaxians all use this method to good effect and we shall now see how to program this facility.

Whilst the keyboard could be used, most programs of this type use either the joystick or the mouse to move the gun backwards and forwards across the screen. Both methods can be accomplished quickly and easily in STOS basic using commands that you have already encountered in previous chapters.

CONTROLLING A GUN USING THE MOUSE

The easiest way to control the gun is using the mouse. Whilst a joystick may be more convenient for many games, it should be remembered that the user may not have a joystick and thus mouse control, etc. should be supplied as an option.

To control a gun using the mouse we simply assign an appropriate sprite to the mouse pointer and then limit its movement to the bottom area of the screen. The following example illustrates this technique.

Load the following:

```
10 rem CONTROL GUN USING MOUSE
20 rem PROGRAM = A:\SPRITES\GUN1
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off
60 :
70 rem SET COLOUR PALETTE
80 palette $0, $666, $444, $0, $322, $334, $112,
$700, $100, $433, $64, $113, $750, $211, $1, $555
90 :
100 rem CHANGE MOUSE POINTER AND LIMIT
110 change mouse 4
120 limit mouse 0,190 to 319,190
```

The main area of interest is line 100 and 110 with previous lines simply carrying out the general housekeeping functions such as setting the colour palette, etc.

Line 110 changes the mouse pointer to sprite number 1 which represents the gun sprite. Line 120 limits the mouse pointer movement so that it can be moved from left to right but cannot be moved up and down. The gun will now automatically follow the movement of the

mouse pointer.

CONTROLLING A GUN USING THE JOYSTICK

The second example allows the gun to be controlled by the joystick.

Load the following:

```
10 rem CONTROL GUN USING JOYSTICK
20 rem PROGRAM = A:\SPRITES\GUN2
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off
60 :
70 rem SET COLOUR PALETTE
80 palette $0, $666, $444, $0, $322, $334, $112,
$700, $100, $433, $64, $113, $750, $211, $1, $555
90 :
100 rem INITIALISE VARIABLES
110 X=160 : Y=190
120 :
130 rem PLACE SPRITE ON SCREEN
140 sprite 1,X,Y,1
150 :
160 rem MONITOR JOYSTICK
170 if jright and X<319 then inc X : sprite 1,x,y,1
180 if jleft and X>0 then dec X : sprite 1,x,y,1
190 goto 170
```

When using the joystick we need to maintain the position of the gun and hence two variables X and Y are used for this. The gun is initially placed at the centre of the screen (X=160,Y=190) by line 140. Line 170 checks to see if the joystick is moved right. If the joystick is moved right we have to look at variable X to check that the gun is not

already at the far right hand side of the screen. If the gun has not yet reached the right hand side of the screen, we can increment variable X and use this within the *sprite* command to reposition the gun. Line 180 checks to see if the joystick has moved left. If it has been moved left, and the gun has not already reached the left hand side of the screen, variable X is decreased by one and the *sprite* command repositions the gun.

The joystick routine is a little more complicated than the mouse routine but a joystick is better suited to many games and also allows the programmer to control the speed at which the gun moves. The speed of movement can be controlled by adding more or less to the variable X when the sprite is moved. For example, to speed the movement up, lines 170 and 180 could be changed to:

```
170 if jright and X<319 then X=X+5 : sprite 1,x,y,1
180 if jleft and X>0 then X=X-5 : sprite 1,x,y,1
```

FIRING A MISSILE

The next element to tackle is that of making the gun fire a missile. Each time that the joysticks fire button is pressed, or if using mouse control the left mouse button is pressed, we need to display and move a missile vertically up the screen. For the next few examples we shall control the gun using the mouse as this requires less code and allows us to concentrate on the missile firing routine.

Load the following:

```
10 rem FIRE BULLET FROM GUN
20 rem PROGRAM = A:\SPRITES\FIRE1
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off
```

```
60 :  
70 rem SET COLOUR PALETTE  
80 palette $0, $666, $444, $0, $322, $334, $112,  
$700, $100, $433, $64, $113, $750, $211, $1, $555  
90 :  
100 rem CHANGE MOUSE POINTER AND LIMIT  
110 change mouse 4  
120 limit mouse 0,190 to 319,190  
130 :  
140 rem MONITOR MOUSE BUTTON  
150 repeat : until mouse key = 1  
160 :  
170 rem FIRE THE MISSILE  
180 sprite 2,x mouse,160,2  
190 move y 2,"(1,-3,62)"  
200 move on  
210 goto 150
```

Run the program and press the left mouse button to fire a few missiles.

Line 150 creates a *repeat..until* loop which continues looping until the left mouse button is pressed.

An appropriate sprite for the missile is located at position 2 within the sprite bank so we need to display this on the screen. Line 180 therefore contains the *sprite* command which assigns the missile sprite to sprite index 2 and displays it on the screen. Notice the Y coordinate of 160 which places the missile just above the gun.

Now that the missile is visible we need to move it up the screen and line 190 therefore defines this movement. There will be 62 movement steps, each of 3 pixels with a 1/50th of a second time delay between each one. Line 200 switches on the movement.

This routine works but it does have a major fault. If you keep pressing the left mouse button you will notice that the missile is removed from its current position and re-launched from the gun each time that the button is pressed. If the routine was used within a full game we would need the missile to continue up the screen until it either hit an enemy craft or flew off the top of the screen. We can overcome the problem by checking to see if the last fired missile has ended its flight before launching a new missile.

Load the following:

```
10 rem FIRE BULLET FROM GUN
20 rem PROGRAM = A:\SPRITES\FIRE2
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off
60 :
70 rem SET COLOUR PALETTE
80 palette $0, $666, $444, $0, $322, $334, $112,
$700, $100, $433, $64, $113, $750, $211, $1, $555
90 :
100 rem CHANGE MOUSE POINTER AND LIMIT
110 change mouse 4
120 limit mouse 0,190 to 319,190
130 :
140 rem MONITOR MOUSE BUTTON
150 repeat : until mouse key = 1 and movon(2) = 0
160 :
170 rem FIRE THE MISSILE
180 sprite 2,x mouse,160,2
190 move y 2,"(1,-3,62)"
200 move on
210 goto 150
```

Run the program and fire a few missiles. You will notice this time

that a new missile cannot be fired until the current missile has left the top of the screen.

The program is identical to the previous example except line 150 which is reproduced below:

```
150 repeat: until mouse key = 1 and movon(2)=0
```

The *movon()* command checks to see if a sprite (whose index number is placed in the brackets) is moving. If the sprite is stationary, the value 0 is returned else another non-specific number is generated. The modified line will now only allow a missile to be fired if the previously fired missile has finished its flight and is thus no longer moving.

The *movon* command could be used as part of an *if..then* expression as shown below:

```
if movon(3) then print "Sprite is moving"  
if movon(3)=0 then print "Sprite is not moving"
```

COLLISION DETECTION

We now know how to control a gun and make it fire missiles vertically up the screen - all we need now is something to shoot at. Enemy space ships can be displayed and moved quite easily but we need a way of checking whether the missiles we fire have hit the target. This section thus addresses the area of collision detection. Each time that a missile is fired we have to check to see if it has collided with the enemy craft. If it does collide we could display an animated explosion.

Collisions are detected by STOS using the *collide* command. This checks for collisions between two sprites (remember that the mouse

pointer is also a sprite) and many people find its operation rather complicated. This is probably due to the fact that it requires the use of binary notation and the mere mention of binary is normally enough to deter even the best of programmers!. Do not let this put you off though as it requires only a very basic understanding of binary and is not really that difficult in practice.

For those who have never encountered binary notation before, it is a method of representing information as a series of 0's and 1's. A computer is not as clever as us humans and cannot recognise the letters of the alphabet and numbers, etc. The computer can only differentiate between on (1) and off (0), and when we type information at the keyboard, this is stored in the computers memory as a series of on and off states. Binary therefore works on a base of 2 rather than 10. For example, consider the number 11. This would normally be represented as:

$$\begin{array}{r} \underline{1000} \quad \underline{100} \quad \underline{10} \quad \underline{1} \\ 1 \quad 1 \end{array}$$

with a '1' in the tens column and a '1' in the single digit column. In binary this would be represented as:

$$\begin{array}{r} \underline{64} \quad \underline{32} \quad \underline{16} \quad \underline{8} \quad \underline{4} \quad \underline{2} \quad \underline{1} \\ 1 \quad 0 \quad 1 \quad 1 \end{array}$$

Therefore the value 11 is represented by 1011 in binary.

Now consider the value 16. This would normally be represented as:

$$\begin{array}{r} \underline{1000} \quad \underline{100} \quad \underline{10} \quad \underline{1} \\ 1 \quad 6 \end{array}$$

with a '1' in the tens column and '6' in the single digit column. In binary this would be represented as:

<u>64</u>	<u>32</u>	<u>16</u>	<u>8</u>	<u>4</u>	<u>2</u>	<u>1</u>
		1	0	0	0	0

Therefore the value 16 is represented by 10000 in binary.

Each '1' and '0' is known as a *bit*. If you find the above explanation confusing do not worry as we are not concerned with the actual numbers - we are only interested in whether a single bit is set to one or zero. Well that it is enough of the theory. Let's delve straight in to a practical example.

DETECTING SINGLE COLLISIONS

Load the following:

```
10 rem SPRITE COLLISION DETECTION
20 rem PROGRAM = A:\SPRITES\COLLIDE1
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off : curs off
60 :
70 rem SET COLOUR PALETTE
80 palette $0, $666, $444, $0, $322, $334, $112,
$700, $100, $433, $64, $113, $750, $211, $1, $555
90 :
100 rem CHANGE MOUSE POINTER AND LIMIT
110 change mouse 4
120 limit mouse 0,190 to 319,190
130 :
140 rem DISPLAY ENEMY CRAFT
150 sprite 1,20,20,11
160 :
170 rem SET ENEMY CRAFT MOVING
180 move x 1,"(1,1,300)(1,-1,300)L"
```



```
190 move on
200 :
210 rem MONITOR LEFT MOUSE BUTTON
220 repeat : until mouse key = 1
230 :
240 rem MOUSE BUTTON PRESSED - FIRE A MISSILE
250 sprite 2,x mouse,160,2
260 move y 2,"(1,-3,61)"
270 move on 2
280 :
290 rem CHECK FOR COLLISION
300 while C=0 and movon(2)
310 C=collide(2,8,8)
320 wend
330 :
340 rem NO COLLISION DETECTED
350 if C=0 then 210
360 :
370 rem COLLISION DETECTED - DISPLAY EXPLOSION
380 sprite off 2
390 move off 1
400 anim 1,"(3,10) (4,10) (5,10) (6,10) (7,10) (8,10)
(9,10) (10,10)"
410 anim on 1
420 wait 80
430 sprite off 1
```

Run the program.

Fire a missile and try and hit the space ship that is flying backwards and forwards across the top of the screen. When you hit the space ship it will explode. Our main interest is the section of the program that detects the collision between the missile and the space ship, but the full program operation is detailed below.

Line 110 changes the mouse pointer to the gun and line 120 limits its movement to the bottom area of the screen.

Line 150 displays the sprite for the space ship and lines 180/190 set this moving.

Line 220 monitors the status of the mouse buttons.

When the left mouse button is pressed, a missile is displayed and set moving up the screen by lines 250-270.

Now we come to the interesting bit. Lines 300-320 form a *while..wend* loop which monitors the missile sprite throughout its flight to see if it collides with the space ship sprite. The routine is reproduced below:

```
300 while C=0 and movon(2)
310 C=collide(2,8,8)
320 wend
```

Line 310 introduces the *collide* command, the format of which is shown below.

C = collide (SPRITE, WIDTH, HEIGHT)

SPRITE indicates the number of the sprite we wish to check for collisions and will be in the range 1-15.

WIDTH and HEIGHT set the sensitivity of the test. They define the width and height of a rectangle around the hot spot of the sprite. When another sprite enters this area, a collision will be detected. If a collision is detected, a binary bit is set within variable C to represent the number of the sprite that collided.

Line 310 of the program thus checks sprite 2 (the missile) for

collisions. If a collision is detected, an appropriate bit is set within variable C else variable C remains equal to zero. In this example we only have one enemy spaceship and thus only need to determine whether or not a collision occurred. The check continues until such time that a collision is detected or movement of the missile ceases due to it reaching the end of the defined movement cycle. When a collision is detected, or when the missile stops moving, the program moves on to line 330.

Line 350 checks the value of variable C to see if a collision occurred. If the value of variable C is still equal to zero (no collision) then the program is sent back to line 210 so that the user can have another go at shooting the space ship.

If the value of C is not equal to zero then we know that the space ship has been hit and can display an explosion. Lines 370-430 display this explosion. Line 380 switches off the missile sprite, line 390 switches off movement of the space ship sprite, lines 400-420 perform the animation and line 430 finally removes the space ship sprite from the screen.

DETECTING MULTIPLE COLLISIONS

The last example only had one enemy space ship and hence we were only interested in detecting a collision. If we had lots of space ships we would need to establish which ship had been hit before displaying an explosion. We would also need some way of detecting when all of the space ships had been hit. We shall now look at a program that addresses these points. The program shall display five space ships to shoot at.

Load the following:

10 rem SPRITE COLLISION DETECTION

```
20 rem PROGRAM = A:\SPRITES\COLLIDE2
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off : curs off
60 :
70 rem SET COLOUR PALETTE
80 palette $0, $666, $444, $0, $322, $334, $112,
$700, $100, $433, $64, $113, $750, $211, $1, $555
90 :
100 rem CHANGE MOUSE POINTER AND LIMIT
110 change mouse 4
120 limit mouse 0,190 to 319,190
130 :
140 rem DISPLAY ENEMY CRAFT
150 sprite 1,20,20,11
160 sprite 2,60,60,16
170 sprite 3,100,20,11
180 sprite 4,140,60,16
190 sprite 5,180,20,11
200 :
210 rem SET NUMBER OF ENEMY CRAFT
220 NO_ALIENS = 5
230 :
240 rem DEFINE ENEMY CRAFT MOVEMENT
250 for A = 1 to 5
260 move x A,"(1,1,120)(1,-1,120)L"
270 move y A,"(2,1,20)(2,-1,20)L"
280 next A
290 :
300 rem ACTIVATE MOVEMENT
310 move on
320 :
330 rem MONITOR LEFT MOUSE BUTTON
340 repeat
350 repeat : until mouse key = 1
```



```
360 :
370 rem MOUSE BUTTON PRESSED SO FIRE MISSILE
380 sprite 6,x mouse,160,2
390 move y 6,"(1,-3,61)"
400 move on 6
410 :
420 rem CHECK FOR COLLISION
430 C = 0
440 while C = 0 and movon(6)
450 C = collide(6,8,8)
460 wend
470 :
480 rem NO COLLISION DETECTED
490 if C = 0 then 330
500 :
510 rem COLLISION DETECTED - WHICH SPRITE HIT
520 for A = 1 to 5
530 if btst(A,C) then SHIP_HIT = A : dec NO_ALIENS
540 next A
550 :
560 rem DISPLAY EXPLOSION
570 sprite off 6
580 move off SHIP_HIT
590 anim SHIP_HIT, "(3,10) (4,10) (5,10) (6,10) (7,10)
(8,10) (9,10) (10,10)"
600 anim on SHIP_HIT
610 wait 80
620 sprite off SHIP_HIT
630 until NO_ALIENS = 0
640 :
650 rem GAME OVER
660 locate 14,10
670 print "GAME OVER"
```

Run the program and try shooting the five enemy space ships. Notice

this time that the ships move in a more interesting zig-zag fashion. The program uses six sprites - sprites 1-5 represent the space ships and sprite 6 represents the missile.

Lines 150-190 display the five space ships on the screen.

Line 220 assigns the number of space ships to variable `NO_ALIENS`. This will be used to maintain the number of space ships that remain to be hit.

Lines 250-280 define the movement for the sprites. Notice how both horizontal and vertical movement is defined and how the loop variable is used to indicate each sprite. Line 310 activates the movement.

Line 350 monitors the left mouse button. If the mouse button is pressed, the program passes on to line 360.

Lines 380-400 display and move the missile up the screen.

Lines 430-460 check the missile for collisions. When a collision is detected or the missile stops moving, the program moves on to line 470.

We now need to check whether or not a collision occurred. Line 490 checks variable `C`, and if equal to zero (no collision), program execution is passed back to line 330 so that the player can have another go. If `C` is not equal to zero then a collision must have occurred and the program passes on to line 500.

We now need to establish which of the five space ship sprites has been hit. This operation is carried out by lines 520-540. Look carefully at line 530. The *btst* (bit test) command is used to check each bit of variable `C`. The loop variable `A` is used to check the binary bits 1-5 within variable `C`. If the bit is set then we have found the sprite with which the collision took place. This is illustrated on the next page.

SPRITE 1 IS HIT	C = %000010
SPRITE 2 IS HIT	C = %000100
SPRITE 3 IS HIT	C = %001000
SPRITE 4 IS HIT	C = %010000
SPRITE 5 is HIT	C = %100000

The number of the hit sprite is assigned to variable SHIP_HIT and the number of space ships remaining (NO_ALIENS) is reduced by one.

Lines 570-620 display the explosion.

Line 630 checks to see if all the space ships have been hit. If NO_ALIENS is not equal to zero then program execution is passed back for the game to continue. If all the space ships have been hit, the program moves on and displays the "GAME OVER" message.

The next two chapters concentrate on the production of two games that will bring together all of the theory and techniques covered in this section. Your games and programs will now only be limited by your imagination and you may feel, now that you are becoming comfortable with programming, that some of the routines in the next two programs could be simplified or performed in a different manner. This being the case, if you think that you can improve on any areas of the programs operation, go ahead and try it.

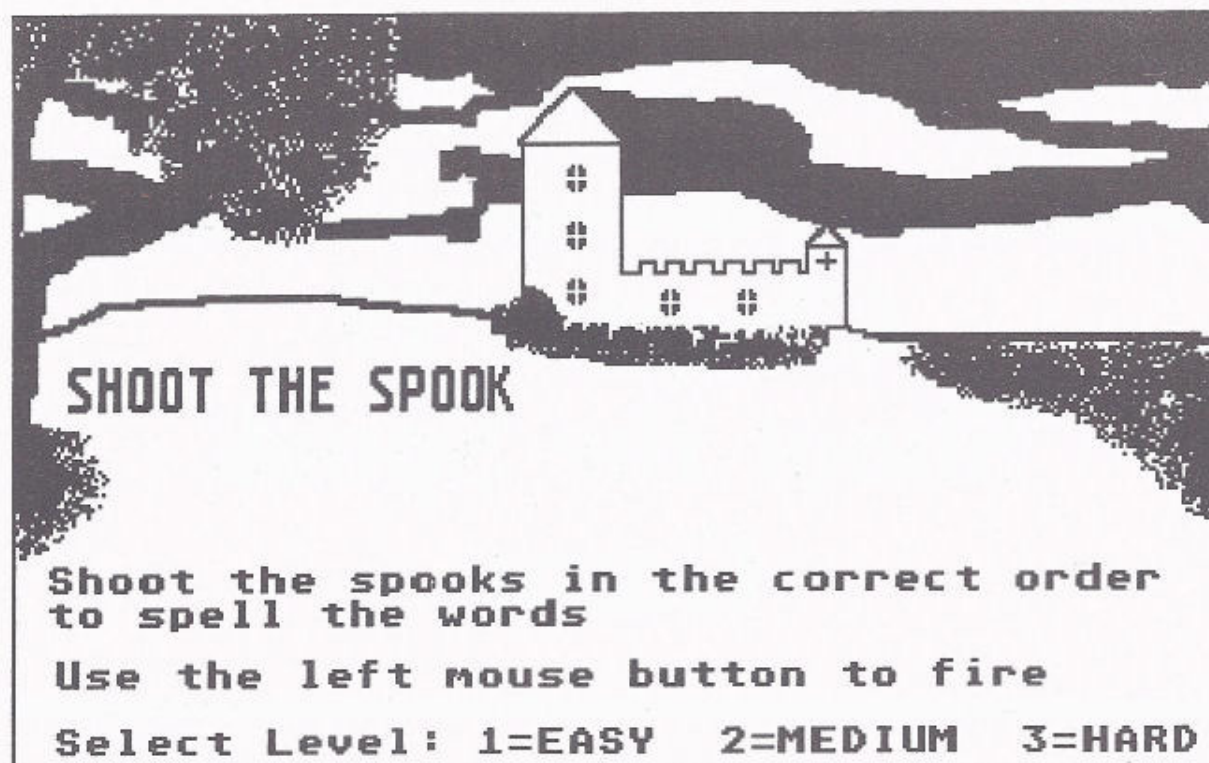
Chapter 19

Shoot the Spook Game

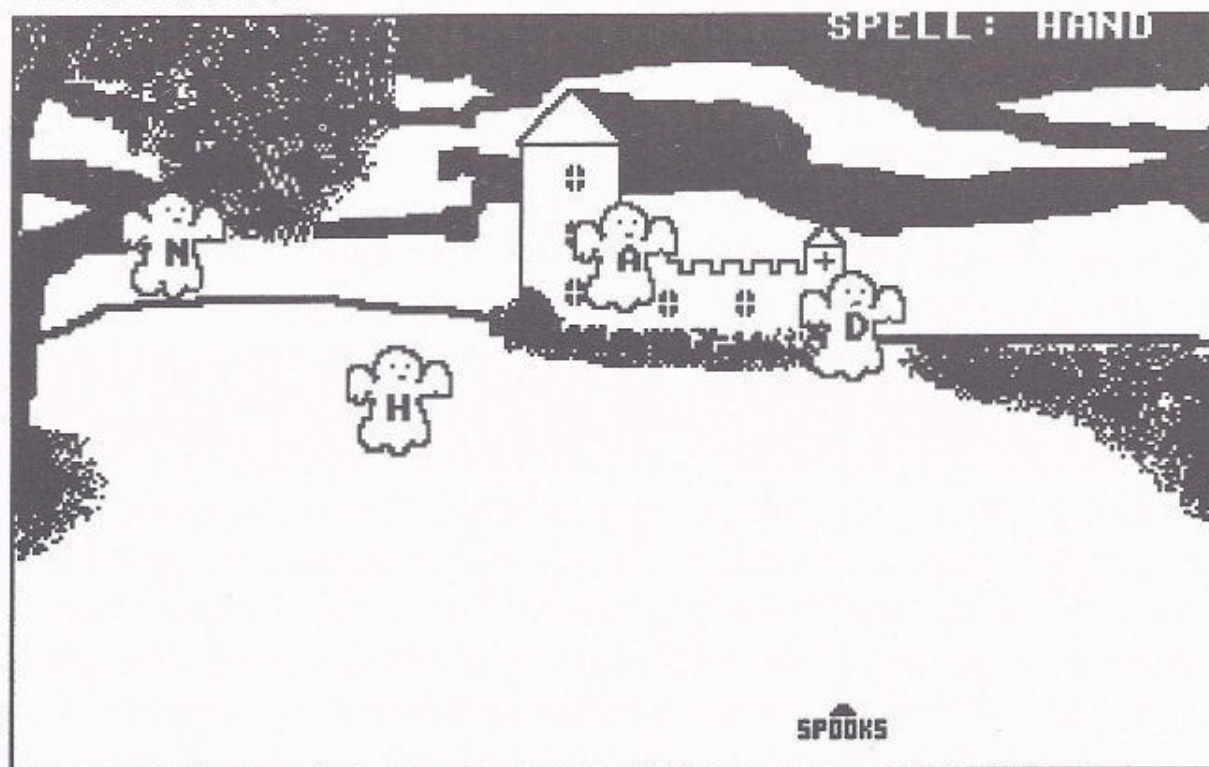
Many shoot-em-up type games rely on a Space theme, and whilst we shall address this scenario in the next chapter, we shall start by producing a game entitled SHOOT THE SPOOK which can be found in the folder named SPOOK on the second disk. This is an educational program designed to help with spelling. A word is displayed at the top of the screen which the user must try and spell by shooting spooks out of the sky. Each spook has a letter on its chest and they must be shot in the correct order to spell the word. The game offers three levels with ten words per level but you can change these parameters as we shall see later.

The program uses two picture files and a set of sprites which are also stored on the disk so that you can load and modify them if you require. The sprites have been saved along with the program (remember that memory bank 1 is permanent) but the two screens are loaded at the start of the program. The two screens are shown over the page. The spook sprites are placed on the background picture is shown.

TITLE



BACKGROUND



The sprite bank consists of the following sprites:

Sprites 1-26	= spooks with the letters A-Z on their chests.
Sprites 27 & 28	= spooks facing left and right with no letters.
Sprite 29	= gun used to shoot at the spooks.
Sprite 30	= missile.
Sprite 31	= 'Game Over' spook.

Load the program:

```
10 rem SHOOT THE SPOOK
20 rem PROGRAM: A:\SPOOK\SPOOK
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off : curs off : hide on
60 :
70 rem DIMENSION VARIABLE ARRAYS
80 dim W$(9),L$(8),POS(8)
90 :
100 rem LOAD BACKDROP
110 reserve as screen 14
120 load "a:\spook\backdrop.pi1",14
130 :
140 rem LOAD AND DISPLAY TITLE SCREEN
150 load "a:\spook\title.pi1"
160 :
170 rem WAIT FOR USER TO ENTER LEVEL
180 while LEV < 1 or LEV > 3
190 LEV$ = input$(1)
200 LEV = val(LEV$)
210 wend
220 :
230 rem DISPLAY GAME BACKGROUND
240 fade 10 : wait 10*7
250 screen copy 14 to physic
```



```
260 screen copy 14 to back
270 fade 10 to 14 : wait 10*7
280 :
290 rem CHANGE MOUSE POINTER + LIMIT
300 show on
310 change mouse 32
320 limit mouse 0,160 to 319,160
330 :
340 rem SET PEN AND PAPER COLOURS
350 pen 15 : paper 2
360 :
370 rem READ WORD DATA IN TO ARRAY
380 if LEV = 1 then restore 1410
390 if LEV = 2 then restore 1440
400 if LEV = 3 then restore 1470
410 for A = 0 to 9
420 read W$(A)
430 W$(A) = upper$(W$(A))
440 next A
450 :
460 rem INITIALISE LOOP FOR 10 WORDS
470 for COUNT = 0 to 9
480 :
490 rem CHOOSE A WORD AT RANDOM
500 R = rnd(9)
510 if W$(R) = "" then 500
520 L = len(W$(R))
530 if L > 8 then cls : print "ERROR-Word too long" :
stop
540 :
550 rem EXTRACT EACH LETTER FROM WORD
560 for A = 1 to L
570 L$(A) = mid$(W$(R), A, 1)
580 next A
590 :
```

```
600 rem DISPLAY WORD AND CLEAR VARIABLE
610 locate 25,0
620 print string$(" ",15);
630 locate 25,0
640 print "SPELL: ";W$(R);
650 :
660 rem CLEAR OLD POSITION INFORMATION
670 for A= 1 to 8
680 POS(A)=0
690 next A
700 :
710 rem POSITION LETTER SPRITES
720 for A= 1 to L
730 Y=rnd(100)
740 if Y<10 then 730
750 P=rnd(8)
760 if P=0 or POS(P)=1 then 750
770 POS(P)=1
780 X=P*30
790 C=asc(L$(A))
800 SPR=C-64
810 sprite A,X,Y,SPR
820 next A
830 NL=1
840 :
850 rem MONITOR GAME
860 repeat
870 :
880 rem WAIT FOR USER TO FIRE
890 repeat : until mouse key = 1
900 :
910 rem DISPLAY AND MOVE BULLET
920 sprite 10,x mouse,y mouse-10,30
930 move y 10,"(1,-5,40)"
940 move on 10
```



```
950 :
960 rem CHECK FOR COLLISION
970 C=0
980 while C=0 and movon(10)
990 C=collide(10,8,8)
1000 wend
1010 :
1020 rem NO COLLISION
1030 if C=0 then 890
1040 :
1050 rem COLLISION - WHICH SPOOK HAS BEEN HIT
1060 for A=1 to L
1070 if btst(A,C) then HIT_SPOOK=A
1080 next A
1090 :
1100 rem IS THIS THE CORRECT LETTER ?
1110 if HIT_SPOOK<>NL then 890
1120 :
1130 rem MAKE SPOOK DISAPPEAR
1140 sprite off 10
1150 inc NL
1160 anim HIT_SPOOK,"(27,10)(28,10)I"
1170 anim on HIT_SPOOK
1180 wait 100
1190 anim off
1200 move y HIT_SPOOK,"(1,-5,45)"
1210 move on HIT_SPOOK
1220 :
1230 rem MORE LETTERS TO HIT
1240 until NL=L+1
1250 wait 50
1260 :
1270 rem CONTINUE ON TO THE NEXT WORD
1280 W$(R)=" "
1290 next COUNT
```

```
1300 :
1310 rem ALL WORDS USED - END GAME
1320 sprite 15,-10,80,31
1330 move x 15,"(1,1,150)"
1340 move on 15
1350 wait key
1360 end
1370 :
1380 :
1390 rem LEVEL 1 WORDS (3 LETTERS)
1400 data "dog", "cat", "hat", "man", "boy", "lip",
"hut", "pin", "lid", "tin"
1410 :
1420 rem LEVEL 2 WORDS (4 LETTERS)
1430 data "bird", "nose", "toes", "bike", "foot",
"hand", "nail", "door", "hold", "loud"
1440 :
1450 rem LEVEL 3 WORDS (HARDER)
1460 data "house", "mouse", "clown", "uncle",
"mother", "father", "sister", "brother", "mouth", "computer"
```

Run the program and play a few games. Notice the way the title screen fades out and the main game screen fades back in. When you hit the correct spook it flaps backwards and forwards before flying off the top of the screen. When ten words have been displayed, the program ends.

As usual the program has been designed in modular form as shown below:

- (1) Display title screen and instructions.
- (2) Ask user to select required level.
- (3) Display background and set mouse pointer.
- (4) Read word data in to array.
- (5) Choose a word at random.

- (6) Extract individual letters and display word.
- (7) Display spooks at random positions.
- (8) Fire a bullet & check for collisions.
- (9) Make spook disappear.
- (10) End game.

(1) DISPLAY TITLE SCREEN AND INSTRUCTIONS

```
40 rem SET SCREEN
50 key off : mode 0 : flash off : curs off : hide on
60 :
70 rem DIMENSION VARIABLE ARRAYS
80 dim W$(9),L$(8),POS(8)
90 :
100 rem LOAD BACKDROP
110 reserve as screen 14
120 load "a:\spook\backdrop.pi1",14
130 :
140 rem LOAD AND DISPLAY TITLE SCREEN
150 load "a:\spook\title.pi1"
```

The game does not require a very high screen resolution but does require a number of different colours. Low screen resolution has thus been chosen and a common colour palette is used for both the picture files and the sprites.

Line 50 switches off the function key window, selects low screen resolution, switches off colour flashing, switches off the text cursor and hides the mouse pointer.

Line 80 dimensions a number of variable arrays that will be required throughout the program. W\$() stores the list of words that are read from the data statements at the end of the program and L\$() stores the list of letters that are extracted from the word. The spook sprites

can be displayed at eight different positions across the screen and variable array POS() keeps track of these.

Lines 110-120 load the background picture in to memory bank 14.

Line 150 loads and displays the title screen. This could have been loaded in to a memory bank but it is only required at the start of the program and there is a big advantage in loading it directly to the screen. As we already know, the sprites and picture files always share the same palette which must be set at the beginning of the program. Loading the picture directly to the screen results in the current colour palette changing to match the picture and thus no further attention need be paid to the definition of colours.

(2) ASK USER TO SELECT REQUIRED LEVEL

```
170 rem WAIT FOR USER TO ENTER LEVEL
180 while LEV < 1 or LEV > 3
190 LEV$=input$(1)
200 LEV=val(LEV$)
210 wend
```

The title screen has a message at the bottom which asks the user to select a level. Lines 180-210 form a *while..wend* loop which waits for the user to make this choice. The *input\$* command is used to input a single character and this is assigned to variable LEV\$. We could have used an *input* command, but as only one character is required, it is easier for the user to simply enter the character without having to also press the Return key. Remember that the *input\$* command can only be used with string variables, and even though we want a numeric value, we have to use the string variable. Line 200 assigns the numeric value of variable LEV\$ to variable LEV. Notice how the loop only exits when a value in the range 1-3 is entered.

(3) DISPLAY BACKGROUND AND SET MOUSE POINTER

```
230 rem DISPLAY GAME BACKGROUND
240 fade 10 : wait 10*7
250 screen copy 14 to physic
260 screen copy 14 to back
270 fade 10 to 14 : wait 10*7
280 :
290 rem CHANGE MOUSE POINTER + LIMIT
300 show on
310 change mouse 32
320 limit mouse 0,160 to 319,160
330 :
340 rem SET PEN AND PAPER COLOURS
350 pen 15 : paper 2
```

We are now ready to play the game. Line 240 fades the entire colour palette to black and thus the title screen fades out. Lines 250-260 copy the main game screen from memory bank 14 to the PHYSICal and BACKground screens, but this is not visible to the user as the colour palette is still set to black. Line 270 fades the picture on to the screen by fading the colour palette to match the contents of memory bank 14.

Line 300 switches on the mouse pointer, line 310 changes this to the gun sprite and line 320 limits its movement to the bottom line of the screen.

Line 350 sets the pen and paper colours.

(4) READ WORD DATA IN TO ARRAY

```
370 rem READ WORD DATA IN TO ARRAY
380 if LEV=1 then restore 1380
390 if LEV=2 then restore 1410
```

```
400 if LEV=3 then restore 1440
410 for A=0 to 9
420 read W$(A)
430 W$(A)=upper$(W$(A))
440 next A
```

Lines 380-400 set the data pointer to the start of the data list. The variable LEV is used to determine whether three, four or more letter words are used as defined by the user at the start of the program (the level).

Lines 410-440 form a *for..next* loop which reads the list of words in to variable array W\$(). Line 430 converts the words to upper case. Remember that variable arrays start at zero and hence the loop is set in the range 0-9 to read the ten words.

(5) CHOOSE A WORD AT RANDOM

```
460 rem INITIALISE LOOP FOR 10 WORDS
470 for COUNT=0 to 9
480 :
490 rem CHOOSE A WORD AT RANDOM
500 R=rnd(9)
510 if W$(R)=" " then 500
520 L=len(W$(R))
530 if L > 8 then cls : print "ERROR-Word too long" : stop
```

The next stage is to choose one of the words at random. Line 470 initiates a *for..next* loop in the range 0-9 which allows 10 words to be displayed. To display more words we could increase the range of the loop and add more words to the list of words.

Line 500 chooses a random number in the range 0-9 and assigns this to variable R. Line 510 checks to see if the word has already been

used. Each time a word is used, we clear the word from the list maintained by W\$(). In this manner we can ensure that the whole list of words are displayed and also that they are displayed in a random order. Therefore, if the word has already been used, the program passes back to line 500 to choose another.

Line 520 assigns the length of the chosen word to variable L and line 530 checks the length of the word. The program allows words with a maximum of eight letters and the check at line 530 ensures that this criteria is met.

(6) EXTRACT INDIVIDUAL LETTERS AND DISPLAY WORD

```
550 rem EXTRACT EACH LETTER FROM WORD
560 for A=1 to L
570 L$(A)=mid$(W$(R),A,1)
580 next A
590 :
600 rem DISPLAY WORD AND CLEAR VARIABLE
610 locate 25,0
620 print string$(" ",15);
630 locate 25,0
640 print "SPELL: ";W$(R);
```

The next stage is to extract the individual letters from the chosen word. Line 560 initiates a *for..next* loop in the range 1 to L. Variable L represents the length of the word and thus the loop will operate for each letter of the word. Line 570 assigns the individual letter to variable array L\$(). Once the loop has completed, the variable array L\$() will contain each letter of the word. For example, if the word is "dog" then variable array L\$() will contain the following:

```
L$(1)  =    "d"
L$(2)  =    "o"
```

L\$(3) = "g"

Lines 610-640 display the word on the screen. Note that lines 610-620 simply display a blank line and are necessary to clear the previous word from the screen before displaying the new word.

(7) DISPLAY SPOOKS AT RANDOM POSITIONS

```
660 rem CLEAR OLD POSITION INFORMATION
670 for A=1 to 8
680 POS(A)=0
690 next A
700 :
710 rem POSITION LETTER SPRITES
720 for A=1 to L
730 Y=rnd(100)
740 if Y<10 then 730
750 P=rnd(8)
760 if P=0 or POS(P)=1 then 770
770 POS(P)=1
780 X=P*30
790 C=asc(L$(A))
800 SPR=C-64
810 sprite A,X,Y,SPR
820 next A
830 NL=1
```

This looks a lot worse than it is!

The next task is to display the spooks on the screen. The spooks can be placed at eight different positions across the screen and these positions are chosen at random so that the letters of the word are muddled.

Lines 670-690 clear the position information used by the previous word.

A random position must now be chosen for each spook, so line 720 initiates a *for..next* loop which will cover each letter (or sprite) to be displayed. Line 730 assigns a random number in the range 0-100 to variable Y. This is the position at which the spook will be displayed down the screen (the Y coordinate). If the chosen value is less than 10, the spook will not fit on the screen and thus line 740 sends the program back to choose another position.

Line 750 assigns a random number in the range 0-8 to variable P and this indicates the position at which the spook will be displayed across the screen. If P is equal to zero, or if the chosen position is already taken, program execution is passed back to line 750 so that another random number can be selected. Variable array POS() maintains a list of used spook positions across the screen, and each time a position is assigned, a value of one is stored in the array. This ensures that two spooks are not displayed at the same position. Line 770 assigns a value of one to variable POS(P). This indicates that the position is now used and should not be used next time around.

Line 780 calculates the X coordinate for the spook across the screen. This is calculated by multiplying the random position by thirty.

Now we have the exact position that the spook will be displayed on the screen, we have to determine which of the spook sprites should be displayed. The first twenty six sprites in the sprite bank (memory bank 1) are spooks with the letters A-Z on their chests and we need to display the correct sprite for each letter. Lines 790-800 calculate which sprite should be displayed. This is quite a clever piece of code so let's take a closer look:

```
790 C=asc(L$(A))  
800 SPR=C-64
```


(8) FIRE A BULLET + CHECK FOR COLLISIONS

```
880 rem WAIT FOR USER TO FIRE
890 repeat : until mouse key=1
900 :
910 rem DISPLAY AND MOVE BULLET
920 sprite 10,x mouse,y mouse-10,30
930 move y 10,"(1,-5,40)"
940 move on 10
950 :
960 rem CHECK FOR COLLISION
970 C=0
980 while C=0 and movon(10)
990 C=collide(10,8,8)
1000 wend
1010 :
1020 rem NO COLLISION
1030 if C=0 then 890
1040 :
1050 rem COLLISION - WHICH SPOOK HAS BEEN HIT
1060 for A=1 to L
1070 if btst(A,C) then HIT_SPOOK=A
1080 next A
1090 :
1100 rem IS THIS THE CORRECT LETTER ?
1110 if HIT_SPOOK < > NL then 890
```

Line 890 monitors the left mouse button. When the left mouse button is pressed, the program continues on to line 900.

Lines 920-940 display and move a bullet up the screen towards the spooks. Line 920 displays the bullet sprite just above the gun, line 930 defines vertical movement and line 940 activates the movement.

Now that the bullet is moving we have to monitor its progress to see

if it collides with any of the spooks. Lines 970-1000 carry out this operation. When a collision is detected or when the bullet stops moving (because it has left the top of the screen without hitting anything), the program moves on to line 1010.

Line 1030 checks to see if a collision has occurred. If the value of C is still equal to zero then there has been no collision and program execution is passed back to line 890 where the user can have another go at shooting the spooks.

If C is not equal to zero then a collision has taken place and we need to determine which spook has been hit. Lines 1060-1080 form a *for..next* loop which tests each binary bit within variable C. The set bit represents the number of the hit spook and this is assigned to variable HIT_SPOOK. Although a spook has been hit, it may not be the correct letter required to spell the word. Line 1110 therefore checks to see if the hit spook is correct.

Variable NL maintains the current letter of the word that the user must shoot. Therefore, if the HIT_SPOOK equals NL, the correct spook has been shot and the program can move on and remove the spook from the screen. If the incorrect spook has been hit, the program is directed back to line 890 where the game continues.

(9) MAKE SPOOK DISAPPEAR

```
1130 rem MAKE SPOOK DISAPPEAR
1140 sprite off 10
1150 inc NL
1160 anim HIT_SPOOK,"(27,10)(28,10)I"
1170 anim on HIT_SPOOK
1180 wait 100
1190 anim off
1200 move y HIT_SPOOK,"(1,-5,45)"
```



```
1210 move on HIT_SPOOK
1220 :
1230 rem MORE LETTERS TO HIT
1240 until NL=L+1
1250 wait 50
1260 :
1270 rem CONTINUE ON TO THE NEXT WORD
1280 W$(R)=" "
1290 next COUNT
```

When the correct spook is hit, it flaps backwards and forwards before flying off the top of the screen.

Line 1140 switches off and removes the bullet sprite from the screen.

Line 1150 increments variable NL. You may recall that this variable maintains the number of the current letter required (1=the first letter, 2=the second letter, etc.).

Lines 1160-1170 define and activate an animation sequence for the hit spook. This makes the spook flap backwards and forwards and uses the sprites at position 27 and 28 in the sprite bank. Notice the letter "L" at the end of the animation sequence in line 1160 which causes the animation to loop.

Line 1180 halts program execution for two seconds so that the user can see the spook flap and then line 1190 terminates the animation.

Lines 1200-1210 define and activate the vertical movement to make the spook fly upwards and off the top of the screen.

Line 1240 checks to see if the word has been completed or whether there are still more spooks to be shot. If there are spooks still remaining, the program loops back so that the user can have another shot. When the word is complete, the program moves on to line 1250.

Line 1250 halts program execution for one second before moving on to the next word. Line 1280 clears the contents of the word array so that the word will not be used again and line 1290 sends the program back to start the next word.

(10) END GAME

```
1310 rem ALL WORDS USED - END GAME
1320 sprite 15,-10,80,31
1330 move x 15,"(1,1,150)"
1340 move on 15
1350 wait key
1360 end
```

When all ten words have been correctly spelt the program ends. To end the game a spook flies on to the screen with a "GAME OVER" message. Line 1320 displays the sprite just off the left hand edge of the screen, line 1330 defines the movement and line 1340 activates the movement. The sprite stops at the centre of the screen and the user presses a key to end the game.

INCREASING THE NUMBER OF WORDS

The game currently offers ten words for each level, but to make the game more interesting, this could be increased. Suppose that you wanted to offer fifty words for each level. The words should be added to the existing data groups using more *data* statements and the following program lines would need changing:

```
80 dim W$(49),L$(8),POS(8)
```

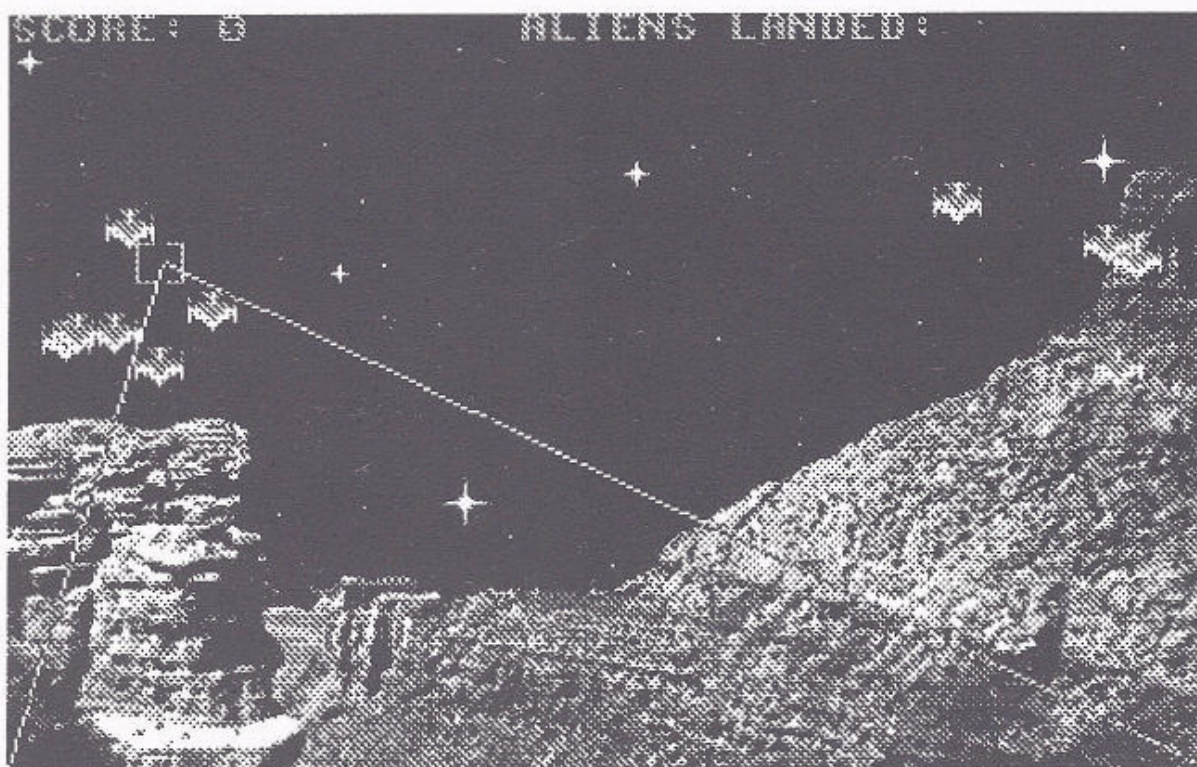
```
410 for A=0 to 49
```


470 for COUNT=0 to 49

500 R=rnd(49)

Chapter 20

Alien Attack Game



In this chapter we shall develop another shoot-em-up game entitled Alien Attack. This shall be slightly different to the last game in that, rather than controlling a gun at the bottom of the screen, we shall give the user complete control and let them move a gun sight over the

whole screen area. The aim of the game is to blast away all of the falling alien ships before they land at the bottom of the screen.

PROGRAM DEFINITION

The aim of the game is to shoot aliens that fall down from the top of the screen. The user starts with ten lives and one is lost each time that an alien reaches the bottom of the screen without being destroyed. The user must destroy fifty aliens to complete a level. Each time that an alien is hit, another alien starts moving down from the top of the screen. The game has ten levels and a bonus is awarded each time that a level is completed without any aliens landing. Each level becomes progressively harder through a combination of speed as illustrated below:

LEVEL 1: the first ten alien ships start slow and new ships are introduced at the same speed.

LEVEL 2: the first ten alien ships start slow and new ships are introduced at the same speed or slightly faster (random selection).

LEVEL 3: the first ten alien ships start slow and new ships are introduced at the same speed, slightly faster, or a lot faster (random selection).

LEVEL 4: the first ten alien ships start at a medium speed and new ships are introduced at the same speed.

LEVEL 5: the first ten alien ships start at a medium speed and new ships are introduced at the same speed or slightly faster (random selection).

LEVEL 6: the first ten alien ships start at a medium speed and new ships are introduced at the same speed, slightly faster or a lot faster

(random selection).

LEVEL 7: the first ten alien ships start at a fast speed and new ships are introduced at the same speed.

LEVEL 8: the first ten alien ships start at a fast speed and new ships are introduced at the same speed or slightly faster (random selection).

LEVEL 9: the first ten alien ships start at a fast speed and new ships are introduced at the same speed, slightly faster or a lot faster (random selection)

LEVEL 10: the first ten alien ships start at a very fast speed and new ships are introduced at even faster speeds (selected at random). It is very unlikely that you will ever reach this level, and if you do, you will not complete it!

The current score and number of aliens landed are displayed at the top of the screen. When the game ends, the final score is displayed at the centre of the screen.

The best way to become accustomed with the practical operation of the game is to play it.

Load the following:

```
10 rem ALIEN ATTACK GAME
20 rem PROGRAM = A:\ALIEN\ALIEN
30 :
40 rem SET SCREEN
50 key off: mode 0: flash off: curs off: hide on: pen 7
60 :
70 rem LOAD MAIN GAME SCREEN
80 reserve as screen 15
90 load "a:\alien\backdrop.pi1",15
```



```
100 :
110 rem LOAD TITLE SCREEN
120 load "a:\alien\title.pi1"
130 while mouse key = 0 : wend
140 :
150 rem SET GRAPHICS WRITING MODE TO XOR
160 gr writing 3
170 :
180 rem CHANGE MOUSE POINTER TO GUN SIGHT
190 change mouse 44
200 show on
210 :
220 rem INITIALISE VARIABLES
230 SC = 0
240 ALIENS_L = 0
250 LEVEL = 0
260 :
270 rem SET THE LEVEL
280 inc LEVEL
290 if LEVEL = 1 then ALIEN_TYPE = 1 : SPEED = 5 :
PTS = 10 : BONUS = 100 : ALIEN_DISPLAY = 1
300 if LEVEL = 2 then ALIEN_TYPE = 11 : SPEED = 5 :
PTS = 20 : BONUS = 200 : ALIEN_DISPLAY = 2
310 if LEVEL = 3 then ALIEN_TYPE = 21 : SPEED = 5 :
PTS = 30 : BONUS = 300 : ALIEN_DISPLAY = 3
320 if LEVEL = 4 then ALIEN_TYPE = 1 : SPEED = 4 :
PTS = 40 : BONUS = 400 : ALIEN_DISPLAY = 1
330 if LEVEL = 5 then ALIEN_TYPE = 11 : SPEED = 4 :
PTS = 50 : BONUS = 500 : ALIEN_DISPLAY = 2
340 if LEVEL = 6 then ALIEN_TYPE = 21 : SPEED = 4 :
PTS = 60 : BONUS = 600 : ALIEN_DISPLAY = 3
350 if LEVEL = 7 then ALIEN_TYPE = 1 : SPEED = 3 :
PTS = 70 : BONUS = 700 : ALIEN_DISPLAY = 1
360 if LEVEL = 8 then ALIEN_TYPE = 11 : SPEED = 3 :
PTS = 80 : BONUS = 800 : ALIEN_DISPLAY = 2
```

```
370 if LEVEL=9 then ALIEN_TYPE=21 : SPEED=3 :  
PTS=90 : BONUS=900 : ALIEN_DISPLAY=3  
380 if LEVEL=10 then ALIEN_TYPE=1 : SPEED=2 :  
PTS=100 : BONUS=1000 : ALIEN_DISPLAY=3  
390 if LEVEL=11 then locate 11,8 : print "ALL LEVELS  
COMPLETED" : goto 2050  
400 ALIENS_HIT=0  
410 OLD_ALIENS_L=ALIENS_L  
420 gosub 510  
430 :  
440 rem IS A BONUS DUE  
450 move freeze  
460 if ALIENS_L=OLD_ALIENS_L then locate 12,10 :  
print "BONUS = ";BONUS : SC=SC+BONUS  
470 if ALIENS_L<>OLD_ALIENS_L then locate 12,10  
: print "NO BONUS"  
480 wait 100  
490 goto 270  
500 :  
510 rem START 10 ALIENS MOVING FROM TOP  
520 rem display main game screen  
530 sprite off  
540 screen copy 15 to physic  
550 screen copy 15 to back  
560 :  
570 rem is user ready to start  
580 home  
590 print " LEVEL: ";LEVEL;" PRESS MOUSE KEY TO  
PLAY"  
600 while mouse key=0 : wend  
610 :  
620 rem clear message and display score  
630 screen copy 15 to physic  
640 screen copy 15 to back  
650 pen 5 : locate 0,0
```



```
660 print "SCORE: ";tab(10);"ALIENS LANDED:"
670 pen 7 : locate 6,0 : print SC : locate 31,0 : print
ALIENS_L
680 :
690 rem start 10 aliens moving
700 for A=1 to 10
710 :
720 rem select random position across screen
730 X=rnd(310)
740 if X<10 then 730
750 :
760 rem select random position down screen
770 Y=rnd(50)
780 Y=-Y
790 :
800 rem display sprite
810 sprite A,X,Y,ALIEN_TYPE
820 :
830 rem create movement string and define movement
840 M$="(" + str$(SPEED) + ",1,0)"
850 move y A,M$
860 :
870 next A
880 :
890 rem start movement
900 move on
910 :
920 rem MONITOR THE GAME
930 repeat
940 :
950 rem has user fired at alien
960 if mouse key=1 then gosub 1040
970 :
980 rem have any aliens landed
990 gosub 1640
```

```
1000 :
1010 until ALIENS_HIT = 50
1020 return
1030 :
1040 rem FIRE GUN AT ALIEN
1050 rem draw lines to gun position
1060 XM=x mouse : YM=y mouse
1070 draw 0,199 to XM,YM
1080 draw 319,199 to XM,YM
1090 :
1100 rem check for collision with alien
1110 gosub 1210
1120 :
1130 rem wait for user to release mouse button
1140 repeat : until mouse key = 0
1150 :
1160 rem remove fire lines from screen
1170 draw 0,199 to XM,YM
1180 draw 319,199 to XM,YM
1190 return
1200 :
1210 rem CHECK FOR COLLISION OF GUNSIGHT AND
ALIEN
1220 C = 0
1230 C = collide(0,8,8)
1240 :
1250 rem if no collision then exit
1260 if C = 0 then 1410
1270 :
1280 rem update total number of aliens hit
1290 inc ALIENS_HIT
1300 :
1310 rem find alien that has been hit
1320 A = 0
1330 repeat : inc A : until btst(A,C)
```



```
1340 ALIEN_HIT = A
1350 :
1360 rem call subroutine to display explosion
1370 gosub 1430
1380 :
1390 rem call subroutine to replace alien
1400 gosub 1860
1410 return
1420 :
1430 rem DISPLAY EXPLOSION
1440 rem sound a boom
1450 boom
1460 :
1470 rem switch off movement of the hit alien
1480 move off ALIEN_HIT
1490 :
1500 rem define and animate explosion sprites
1510 anim ALIEN_HIT, "(31,5) (32,5) (33,5) (34,5)
(35,5) (36,5) (37,5) (38,5)"
1520 anim on ALIEN_HIT
1530 wait 12
1540 :
1550 rem remove sprite from screen
1560 sprite off ALIEN_HIT
1570 :
1580 rem update score
1590 SC = SC + PTS
1600 locate 6,0
1610 print SC
1620 return
1630 :
1640 rem CHECK TO SEE IF ALIENS LANDED
1650 for A = 1 to 10
1660 :
1670 rem has sprite reached bottom of screen
```

```
1680 if y sprite(A) < 210 then 1830
1690 :
1700 rem sprite has reached bottom of screen
1710 bell
1720 sprite off A
1730 inc ALIENS_L
1740 locate 31,0
1750 print ALIENS_L
1760 :
1770 rem have 10 aliens landed
1780 if ALIENS_L = 10 then 2050
1790 rem call subroutine to display new alien
1800 ALIEN_HIT = A
1810 gosub 1860
1820 :
1830 next A
1840 return
1850 :
1860 rem START NEW SHIP
1870 rem select random starting position across screen
1880 X = rnd(310)
1890 if X < 10 then 1880
1900 :
1910 rem select style of sprite and display
1920 SPR = ALIEN_TYPE + rnd(9)
1930 sprite ALIEN_HIT,X,-8,SPR
1940 :
1950 rem create movement string and define
movement
1960 ALIEN = rnd(ALIEN_DISPLAY)
1970 if ALIEN = 0 then ALIEN = 1
1980 if ALIEN = 1 then M$ = "(" + str$(SPEED) + ",1,0)"
1990 if ALIEN = 2 then M$ = "(" + str$(SPEED) + ",2,0)"
2000 if ALIEN = 3 then M$ = "(" + str$(SPEED) + ",3,0)"
2010 move y ALIEN_HIT,M$
```



```
2020 move on ALIEN_HIT
2030 return
2040 :
2050 rem GAME OVER
2060 locate 11,10
2070 print "GAME OVER"
2080 locate 11,12
2090 print "FINAL SCORE: ";SC
2100 wait key
2110 end
```

Run the program and play it for a while. The first three levels are fairly easy but the game soon starts to speed up!

The program listing looks very involved but its size is rather deceiving as a lot of the lines simply contain remarks and colons to separate the program in to modules and thus make it easier to study.

DESIGN

Now that you have a feel for the games operation we can take a look at the program itself.

The program uses two picture files which serve as a title screen and a main game screen. The game also uses a set of sprites which have been saved along with the program and which are detailed below:

Sprites 1-10	Alien Ships - style 1
Sprites 11-20	Alien Ships - style 2
Sprites 21-30	Alien Ships - style 3
Sprites 31-38	Explosion
Sprite 41	Gun Sight

Both the pictures and the sprites share a common colour palette.

The program consists of 10 main modules and these are listed below:

- (1) Initialisation.
- (2) Set the game level.
- (3) Start first 10 aliens moving down the screen.
- (4) Monitor the game.
- (5) Fire the gun at alien.
- (6) Check for collisions.
- (7) Display explosion.
- (8) Check to see if aliens have landed.
- (9) Start a new alien ship.
- (10) Game over.

We shall now look at each of the modules in turn.

(1) INITIALISATION

```
40 rem SET SCREEN
50 key off : mode 0 : flash off : curs off : hide on : pen 7
60 :
70 rem LOAD MAIN GAME SCREEN
80 reserve as screen 15
90 load "a:\alien\backdrop.pi1",15
100 :
110 rem LOAD TITLE SCREEN
120 load "a:\alien\title.pi1"
130 while mouse key=0 : wend
140 :
150 rem SET GRAPHICS WRITING MODE TO XOR
160 gr writing 3
170 :
180 rem CHANGE MOUSE POINTER TO GUN SIGHT
190 change mouse 44
200 show on
210 :
```



```
220 rem INITIALISE VARIABLES
230 SC=0
240 ALIENS_L=0
250 LEVEL=0
```

Line 50 switches off the function key window, sets the screen mode to low resolution, switches off colour flashing, switches off the text cursor, hides the mouse pointer and sets the pen colour to index 7.

Lines 80-90 load the main backdrop picture in to memory bank 15.

Line 120 loads the title screen directly to the screen and the colour palette is automatically changed accordingly. The title screen asks the user to press the mouse button to start the game and the loop in line 130 waits for this action to take place. When the mouse button is pressed, the program continues.

Line 160 sets the graphics writing mode to 3 (XOR). The reason for this will become apparent later so do not worry yet.

Line 190 changes the mouse pointer to a gun sight and line 200 displays this on the screen.

Lines 230-250 clear three variables ready for use during the game. Variable SC is used to maintain the current score, variable ALIENS_L is used to maintain how many aliens have landed (reached the bottom of the screen) and variable LEVEL indicates the current level of the game.

(2) SET THE GAME LEVEL

```
270 rem SET THE LEVEL
280 inc LEVEL
290 if LEVEL=1 then ALIEN_TYPE=1 : SPEED=5 :
```

```
PTS=10 : BONUS=100 : ALIEN_DISPLAY=1
    300 if LEVEL=2 then ALIEN_TYPE=11 : SPEED=5 :
PTS=20 : BONUS=200 : ALIEN_DISPLAY=2
    310 if LEVEL=3 then ALIEN_TYPE=21 : SPEED=5 :
PTS=30 : BONUS=300 : ALIEN_DISPLAY=3
    320 if LEVEL=4 then ALIEN_TYPE=1 : SPEED=4 :
PTS=40 : BONUS=400 : ALIEN_DISPLAY=1
    330 if LEVEL=5 then ALIEN_TYPE=11 : SPEED=4 :
PTS=50 : BONUS=500 : ALIEN_DISPLAY=2
    340 if LEVEL=6 then ALIEN_TYPE=21 : SPEED=4 :
PTS=60 : BONUS=600 : ALIEN_DISPLAY=3
    350 if LEVEL=7 then ALIEN_TYPE=1 : SPEED=3 :
PTS=70 : BONUS=700 : ALIEN_DISPLAY=1
    360 if LEVEL=8 then ALIEN_TYPE=11 : SPEED=3 :
PTS=80 : BONUS=800 : ALIEN_DISPLAY=2
    370 if LEVEL=9 then ALIEN_TYPE=21 : SPEED=3 :
PTS=90 : BONUS=900 : ALIEN_DISPLAY=3
    380 if LEVEL=10 then ALIEN_TYPE=1 : SPEED=2 :
PTS=100 : BONUS=1000 : ALIEN_DISPLAY=3
    390 if LEVEL=11 then locate 11,8 : print "ALL LEVELS
COMPLETED" : goto 2050
    400 ALIENS_HIT=0
    410 OLD_ALIENS_L=ALIENS_L
    420 gosub 510
    430 :
    440 rem IS A BONUS DUE
    450 move freeze
    460 if ALIENS_L=OLD_ALIENS_L then locate 12,10 : print
"BONUS = ";BONUS : SC=SC+BONUS
    470 if ALIENS_L < > OLD_ALIENS_L then locate 12,10 :
print "NO BONUS"
    480 wait 100
    490 goto 270
```

This looks quite complicated due to the number of variables but it

really is quite straightforward.

Line 280 increments the level. The program returns to this point each time that a level is completed and hence the game must move on to the next level.

Lines 290-390 use the value of variable LEVEL to set a number of other variables that control the game play. These variables are detailed below:

ALIEN_TYPE: The game uses three different styles of aliens. There are normal space ships, pod type space ships and flying saucer type space ships. The style of ship is changed for each level and so variable ALIEN_TYPE indicates the starting position of the space ship sprites within the sprite bank. Look back to the original sprite list and this should become clearer - the first set of space ship sprites occupy positions 1-10, the second set occupy positions 11-20 and the last set occupy positions 21-30.

SPEED: this specifies the speed at which the aliens will move down the screen. It is used later within a *move y* command to specify the time delay between each step of the movement.

PTS: this specifies the number of points that will be awarded each time that an alien is hit and destroyed.

BONUS: this specifies the number of points that will be awarded if the user achieves a bonus. A bonus is awarded if the user completes a level without letting any of the aliens land.

ALIEN_DISPLAY: this specifies the speed of movement for new aliens that are added to the screen when an alien is destroyed or when an alien lands. The first ten aliens always start at a constant speed but new aliens can have a different speed to make the game more challenging. The variable can be assigned one of three values:

ALIEN_DISPLAY = 1: all new aliens will remain at a constant speed

ALIEN_DISPLAY = 2: new aliens may be at the original speed or slightly faster.

ALIEN_DISPLAY = 3: new aliens may be at the original speed, slight faster or very fast.

Look at the level data carefully and you will see that the levels are made progressively harder by changing the speed at which the aliens move and the speed at which new aliens are introduced in to the game.

Now that the level has been set, we can start the game.

Line 400 resets the variable ALIENS_HIT. This, as its name implies, maintains a total of how many aliens the user has hit. The level is complete when fifty aliens have been hit.

Line 410 assigns the current number of aliens that have landed (variable ALIENS_L) to variable OLD_ALIENS_L. This is used at the end of the level to see if any further aliens have landed and hence determine whether a bonus is due.

Line 420 calls a subroutine. The subroutine is located at line 510 and starts the first ten aliens moving down the screen.

We shall cover the operations for the main game play in a moment, but for now, let us just look at the bonus points routine which also forms part of the main level setting module.

When a level is completed, program execution is passed back to line 450 to check if a bonus is due. Line 450 freezes all sprite movement and lines 460/470 check to see if the bonus is due.

Line 460 compares the number of aliens landed (ALIENS_L) to the number that had landed at the end of the previous level (OLD_ALIENS_L). If no more aliens have landed then the bonus is awarded. If aliens have landed then no bonus is awarded and line 470 displays this fact on the screen.

Line 480 halts program execution for two seconds allowing the user time to read the bonus information.

Line 490 sends program execution back to line 270 where the next level is set up.

(3) START FIRST 10 ALIENS MOVING DOWN THE SCREEN

```
510 rem START 10 ALIENS MOVING
520 rem display main game screen
530 sprite off
540 screen copy 15 to physic
550 screen copy 15 to back
560 :
570 rem is user ready to start
580 home
590 print " LEVEL: ";LEVEL;" PRESS MOUSE KEY TO
PLAY"
600 while mouse key=0 : wend
610 :
620 rem clear message and display score
630 screen copy 15 to physic
640 screen copy 15 to back
650 pen 5 : locate 0,0
660 print "SCORE: ";tab(10);"ALIENS LANDED:"
670 pen 7 : locate 6,0 : print SC : locate 31,0 : print
ALIENS_L
680 :
```

```
690 rem start 10 aliens moving
700 for A=1 to 10
710 :
720 rem select random position across screen
730 X=rnd(310)
740 if X<10 then 730
750 :
760 rem select random position down screen
770 Y=rnd(50)
780 Y=-Y
790 :
800 rem display sprite
810 sprite A,X,Y,ALIEN_TYPE
820 :
830 rem create movement string and define movement
840 M$="("+str$(SPEED)+",1,0)"
850 move y A,M$
860 :
870 next A
880 :
890 rem start movement
900 move on
```

Line 530 uses the *sprite off* command to switch off any sprites that are currently displayed from the previous level.

Lines 540 and 550 copy the main game screen from memory bank 15 to the PHYSICAL and BACKground screens. This removes the old bonus score information from the screen.

Line 580 uses the *home* command to position the text cursor at the top, left hand corner of the screen and line 590 displays the current level and a message asking the user to press a mouse key to start playing.

Line 600 monitors the state of the mouse buttons using a *repeat..until* loop. When a mouse button is pressed the game starts.

Lines 630 and 640 once again copy the main game screen from memory bank 15 to the PHYSICal and BACKground screens so as to clear the previous message.

Lines 650-670 display the current score information on the top line of the screen.

We can now start the first ten aliens moving from the top of the screen. These are to start at random positions across the screen and random positions down the screen.

Line 700 initiates a *for..next* loop which shall allow the ten aliens to be set up.

Lines 730-740 select a random starting position across the screen. If the selected value is smaller than ten, it is reselected as the alien sprite would not be able to fit on the left hand edge of the screen.

Lines 770-780 select a random starting position down the screen. A random number in the range 0-50 is assigned to variable Y and this is then converted to a negative value. Remember that sprites can be placed at negative coordinates on the screen which effectively wraps them around the back of the screen and thus hides them from view. The alien sprites will initially be placed above the top of the screen and out of view of the user.

Line 810 displays the sprite at the random position on the screen. Remember that this will currently be out of view off the top of the screen.

Line 840 creates the movement string. Notice how the variable SPEED is used to set the speed of movement and how it has to be

converted to a string form using the *str\$* function. Line 850 defines the movement using the movement string.

Lines 870 contains a *next* command which sends the program back to define the next alien position. When all ten aliens have been defined, the program moves on to line 890.

Line 900 starts the movement using the *move on* command. Remember that the *move on* command starts movement of all sprites unless a certain sprite is specified.

Now that the sprites are moving, we can monitor the game.

(4) MONITOR THE GAME

```
920 rem MONITOR THE GAME
930 repeat
940 :
950 rem has user fired at alien
960 if mouse key=1 then gosub 1040
970 :
980 rem have any aliens landed
990 gosub 1640
1000 :
1010 until ALIENS_HIT=50
1020 return
```

There are two main areas that we need to monitor. First, we need to regularly check the status of the mouse buttons to see if the user has fired at an alien. Secondly, we need to regularly check the position of the aliens to see if any have reached the bottom of the screen.

We already know that to complete a level the user must destroy fifty aliens. A loop can therefore be used to continue the monitoring

process until this criteria has been met. Lines 930-1010 therefore form a *repeat...until* loop.

Line 960 checks the mouse buttons, and if the left mouse button is pressed, the program branches to the subroutine at line 1040 which fires the gun, etc. Line 990 calls a subroutine that checks the position of all the alien sprites to see if they have reached the bottom of the screen.

When fifty aliens have been destroyed, the program moves on to line 1020 which sends program execution back to line 440 so that a bonus, if due, can be awarded and the game can move on to the next level.

(5) FIRE GUN AT ALIEN

```
1040 rem FIRE GUN AT ALIEN
1050 rem draw lines to gun position
1060 XM=x mouse : YM=y mouse
1070 draw 0,199 to XM,YM
1080 draw 319,199 to XM,YM
1090 :
1100 rem check for collision with alien
1110 gosub 1210
1120 :
1130 rem wait for user to release mouse button
1140 repeat : until mouse key=0
1150 :
1160 rem remove fire lines from screen
1170 draw 0,199 to XM,YM
1180 draw 319,199 to XM,YM
1190 return
```

When the user presses the left mouse button we need to fire the gun. This consists of displaying two lines from each side of the screen

which meet with the gun sight. Line 1060 assigns the current mouse pointer coordinates to variables XM and YM. Lines 1070 and 1080 use these variables to draw the firing lines from each side of the screen.

We now have to determine whether or not the gun sight is over an alien and hence whether an alien has been hit. Line 1110 calls a subroutine which carries out this operation and we shall look at this in a moment.

Lines 1140-1150 wait for the user to release the mouse button before continuing.

Lines 1170-1180 draw the fire lines again. You may remember that, at the start of the program, we set the graphics writing mode to 3 (XOR). In XOR mode we can write information to the screen and then remove it again by copying the same information over the top. By drawing the fire lines a second time we can completely remove the lines from the screen.

Line 1190 returns program execution back to line 970 where monitoring of the game continues.

(6) CHECK FOR COLLISIONS

```
1210 rem CHECK FOR COLLISION OF GUN SIGHT AND  
ALIENS
```

```
1220 C=0
```

```
1230 C=collide(0,8,8)
```

```
1240 :
```

```
1250 rem if no collision then exit
```

```
1260 if C=0 then 1410
```

```
1270 :
```

```
1280 rem update total number of aliens hit
```



```
1290 inc ALIENS_HIT
1300 :
1310 rem find alien that has been hit
1320 A=0
1330 repeat : inc A : until btst(A,C)
1340 ALIEN_HIT=A
1350 :
1360 rem call subroutine to display explosion
1370 gosub 1430
1380 :
1390 rem call subroutine to replace alien
1400 gosub 1860
1410 return
```

This routine checks to see if the gun sight was over one of the aliens when the mouse button was pressed and hence determines whether an alien has been hit.

Line 1220 clears variable C which shall be used with the *collide* command.

Line 1230 checks for collisions with the mouse pointer (sprite 0) and assigns the result to variable C. Remember that the *collide* command sets a binary bit representing each sprite with which a collision has occurred. In previous examples we placed the *collide* command within a loop so as to monitor the progress of a bullet as it travelled up the screen. In this program we make a quick check and then move on.

Line 1260 checks the value of variable C. If it is equal to zero then no collision took place and program execution is passed to line 1410 where the subroutine exits.

If the value of C is not equal to zero, a collision has been detected and we need to establish which one of the alien sprites is involved.

Line 1290 increments variable ALIENS_HIT which maintains the total number of aliens hit.

Line 1320 clears variable A.

Line 1330 forms a *repeat..until* loop which tests each binary bit of variable C. The loop continues until the set bit is detected. Notice how variable A is increased during each pass of the loop and how it is used within the *bst* test to check the bits. When the loop exits, variable A contains the number of the sprite and this is assigned to variable ALIEN_HIT in line 1340.

Now that we know which alien has been hit, we can display an explosion and line 1370 calls the relevant subroutine.

Finally, line 1400 calls a subroutine to start a new alien at the top of the screen.

(7) DISPLAY EXPLOSION

```
1430 rem DISPLAY EXPLOSION
1440 rem sound a boom
1450 boom
1460 :
1470 rem switch off movement of the hit alien
1480 move off ALIEN_HIT
1490 :
1500 rem define and animate explosion sprites
1510 anim ALIEN_HIT, "(31,5) (32,5) (33,5) (34,5) (35,5)
(36,5) (37,5) (38,5)"
1520 anim on ALIEN_HIT
1530 wait 12
1540 :
1550 rem remove sprite from screen
```



```
1560 sprite off ALIEN_HIT
1570 :
1580 rem update score
1590 SC=SC+PTS
1600 locate 6,0
1610 print SC
1620 return
```

Line 1450 uses the *boom* command to generate an explosion sound.

Line 1480 switches off movement of the alien sprite.

Line 1510 defines the animation sequence for the explosion and line 1520 activates the animation.

Line 1530 halts program execution for 12/50ths of a second which gives the explosion sequence time to operate.

Line 1560 removes the sprite from the screen.

Lines 1590-1610 update the score and display it at the appropriate position at the top of the screen.

Line 1620 returns program execution to the calling program.

(8) CHECK TO SEE IF ALIENS HAVE LANDED

```
1640 rem CHECK TO SEE IF ALIENS LANDED
1650 for A=1 to 10
1660 :
1670 rem has sprite reached bottom of screen
1680 if y sprite(A)<210 then 1830
1690 :
1700 rem sprite has reached bottom of screen
```

```
1710 bell
1720 sprite off A
1730 inc ALIENS_L
1740 locate 31,0
1750 print ALIENS_L
1760 :
1770 rem have 10 aliens landed
1780 if ALIENS_L=10 then 2050
1790 rem call subroutine to display new alien
1800 ALIEN_HIT=A
1810 gosub 1860
1820 :
1830 next A
1840 return
```

When a sprite reaches the bottom of the screen we must subtract one of the users lives and start another alien at the top of the screen. Ten aliens are displayed on the screen simultaneously and we must regularly check the position of them to see if any have landed.

Line 1650 initiates a *for..next* loop in the range 1-10 so that the loop variable A can be used to represent each of the ten sprites.

Line 1680 checks to see if the current Y coordinate of the alien sprite is less than two hundred and ten. Remember that Y screen coordinates are taken from the top of the screen (0) down to the bottom of the screen (199), and if the y coordinate of the alien sprite is less than 210, it is still on the screen and thus the program is sent to line 1830 so that the next alien sprite can be tested.

If the sprite has reached the bottom of the screen, the program moves on to line 1700.

Line 1710 generates a bell sound indicating to the user that an alien has landed.

Line 1720 switches off the sprite.

Lines 1730-1750 update the variable `ALIENS_L` and display this updated information on the screen. Remember that the variable `ALIENS_L` maintains the number of aliens that have landed during the game and that the game ends when ten have landed.

Line 1780 checks to see if ten aliens have landed, and if so, sends program execution to line 2050 where the final score is displayed and the game ends. If ten aliens have not yet landed, the program moves on to line 1800.

When an alien lands we must start another alien from the top of the screen in the same way as when an alien is destroyed. Line 1800 therefore assigns the number of the sprite to be replaced to variable `ALIEN_HIT` and line 1810 calls a subroutine to start a new alien.

Line 1830 sends the program back to the start of the *for..next* loop so that the next alien sprite can be checked and line 1840 returns control to the calling program (line 1000).

(9) START A NEW ALIEN SHIP

```
1860 rem START NEW SHIP
1870 rem select random starting position across screen
1880 X=rnd(310)
1890 if X<10 then 1880
1900 :
1910 rem select style of sprite and display
1920 SPR=ALIEN_TYPE+rnd(9)
1930 sprite ALIEN_HIT,X,-8,SPR
1940 :
1950 rem create movement string and define movement
1960 ALIEN=rnd(ALIEN_DISPLAY)
```

```
1970 if ALIEN=0 then ALIEN=1
1980 if ALIEN=1 then M$="(" + str$(SPEED) + ",1,0)"
1990 if ALIEN=2 then M$="(" + str$(SPEED) + ",2,0)"
2000 if ALIEN=3 then M$="(" + str$(SPEED) + ",3,0)"
2010 move y ALIEN_HIT,M$
2020 move on ALIEN_HIT
2030 return
```

Whenever an alien is destroyed or leaves the bottom of the screen, we must start a new alien at the top of the screen.

Line 1880 selects a random starting position across the screen and assigns this to variable X. If the value of variable X is less than 10, line 1890 sends the program back to choose another number as there would not be enough room on the left hand side of the screen to display the sprite.

Line 1920 selects, at random, the style of alien sprite to display. Remember that the variable ALIEN_TYPE indicates the starting position of the current set of alien sprites within the sprite bank. Each set of alien sprites has 10 separate sprites and one of these is selected at random.

Line 1930 displays the sprite on the screen. It is positioned just off the top of the screen and out of sight of the user.

Lines 1960-2020 define the movement for the sprite. The sprite can be moved at one of three different speeds depending on the value of variable ALIEN_DISPLAY. This is set during the level definitions near the start of the program. The value of variable ALIEN_DISPLAY determines the speed of movement as shown below:

ALIEN_DISPLAY = 1	one speed	standard speed
-------------------	-----------	----------------

ALIEN_DISPLAY = 2	two speeds	standard speed medium speed
-------------------	------------	--------------------------------

ALIEN_DISPLAY = 3	three speeds	standard speed medium speed fast speed
-------------------	--------------	--

So how does this work within the program itself ?

Line 1960 chooses a random number in the range 0-ALIEN_DISPLAY and assigns this to variable ALIEN. The variable ALIEN_DISPLAY therefore controls the range of the random number.

We only actually want random numbers starting at the value 1, so if the random number is zero, line 1970 changes it to one.

Lines 1980-2000 create a movement string depending on which random number is selected. Line 1980 creates the current speed, line 1990 creates a medium speed and line 2000 creates a fast speed.

Line 2010 uses the movement string to define the movement and line 2020 switches on the movement.

Line 2030 sends program execution back to the calling program which could be either line 1410 or line 1820.

(10) GAME OVER

```
2050 rem GAME OVER
2060 locate 10,10
2070 print "GAME OVER"
2080 locate 10,12
2090 print "FINAL SCORE: ";SC
```

```
2100 wait key  
2110 end
```

The final part is the GAME OVER module where the final score is displayed and the game ends. This is quite straightforward and should require no explanation.

Well that brings us to the end of Alien Attack. If you are unsure about any area then run the game whilst looking at the program listing. Try and relate the programs operation to the series of commands.

CHANGING THE GAME

CHANGING THE PICTURES: the title and background pictures are supplied along with the program and can be altered if required.

CHANGING THE SPRITES: the sprite bank contains all of the sprites which can be changed using the sprite designer. There are three sets of space craft and you may like to change or add to these.

CHANGING THE NUMBER OF LIVES: the game currently ends when 10 aliens land at the bottom of the screen but this can be changed to make the game easier or harder. A single line of code can be changed to implement this:

```
1780 if ALIENS_L=10 then 2050
```

CHANGING THE NUMBER OF ALIENS DISPLAYED: to complete a level, the user must destroy fifty aliens. You may like to increase or decrease this number by changing the following line:

```
1010 until ALIENS_HIT=50
```


CHANGING THE SPEED OF ALIEN MOVEMENT: the speed at which the aliens move is determined by the movement commands. You may like to change the speed at which the initial aliens move or the speed at which new aliens are introduced.

FINAL NOTE

Alien Attack made, what some may consider to be, rather excessive use of subroutines. The collision detection and the explosion modules were produced as subroutines when they could have formed part of the gun firing routine. This use of the *gosub* command causes the program to 'jump' around a little but does allow each module to be produced as a single unit and also provides a good structure to the program.

Programming technique grows with experience and you will soon develop your own style. Some people prefer to create programs as a series of subroutines whilst others prefer to complete the task in one hit. Provided the program works, it really does not matter what method you use.

Chapter 21

Music & Sound Effects

Anybody who has seen or played games on the Atari ST could not fail to be impressed by the very high quality music and sound effects that are produced by the software. A few years ago most games could only offer a few simple beeps but now we have complete, record quality, sound tracks that really are quite amazing.

The Atari computer has a sound generator chip which can produce a large array of sounds and effects, and STOS offers a range of commands for making the most of these. There are three main areas available to the budding musician and these are listed below:

- (1) Simple beeps and sound effects.
- (2) Computer generated music.
- (3) Sampled or digitised sound and music.

Most games can gain from the introduction of sound and even the simplest of effects can add new dimension to a program. We shall start by looking at basic sound generation and shall work through to

see how the commercial companies achieve such high quality sound effects within their software.

SIMPLE BEEPS AND SOUND EFFECTS

Single Tones

The Atari's sound chip can play three separate voices simultaneously. These voices may be used individually or combined to form other effects and harmonies. The most basic of the STOS sound commands is *play* which plays a single note on one or all of the voices. You have probably noticed that, when you press a key on the keyboard, the computer makes a clicking sound. Before we use any of the sound commands we need to switch off the key click to ensure no interaction when a key is pressed.

Enter the following:

click off

Now we have to set the volume, so enter the following:

**volume 15
play 0,0**

The volume can be in the range 1 (quiet) to 15 (loud) although a value of zero can be used to switch the sound completely off. The *play 0,0* command stops any previous sounds that may be playing.

We can now use the *play* command to produce sound. Enter the following:

play 1,44,10

and you should hear a single tone.

To stop the sound enter the following:

play 0,0

The format of the *play* command is shown below:

play VOICE, PITCH, DURATION

VOICE indicates the number of the voice to be played. This can be any of the voices 1,2 or 3, and if omitted, causes the command to operate on all three of the voices.

PITCH indicates the pitch of the note.

DURATION: indicates the amount of time in 50ths of a second that shall elapse before the program moves on to the next command.

The description of the *play* command in the STOS User Guide is rather misleading as it implies that the DURATION specifies the length of time that the note will play. You would therefore consider the command **play 1,25,50** to play for one second (50/50ths of a second) and then stop, but this is not the case. What actually happens is that the note is set playing continuously and the command waits for one second before moving on to the next command in the program. This was demonstrated in the previous program where we specified a duration of 10 but the note played continuously.

Morse Code Keyboard

Let us now consider the *play* command used in a practical application. The following program will be of interest to amateur radio operators, scouts or anybody interested in learning morse code. When a letter is pressed on the keyboard, the computer will convert it to morse code and sound this using the *play* command.

Load the following:

```
10 rem MORSE CODE PROGRAM
20 rem PROGRAM : A:\MUSIC\MORSE
30 :
40 rem READ THE LETTER DATA
50 dim M$(26)
60 for A=1 to 26
70 read M$(A)
80 next A
90 :
100 rem SWITCH OFF KEY CLICK
110 click off
120 :
130 rem SET THE VOLUME
140 volume 15
150 play 0,0
160 :
170 rem ASK USER FOR LETTER
180 input "Enter a letter";L$
190 L$ = upper$(L$)
200 :
210 rem FIND THE MORSE LETTER
220 L = asc(L$)-64
230 L$ = M$(L)
240 :
250 rem SOUND THE MORSE CODE
260 for A=1 to len(L$)
270 A$ = mid$(L$,A,1)
280 if A$ = "1" then play 1,40,30 : play 0,10
290 if A$ = "0" then play 1,40,10 : play 0,10
300 next A
310 play 0,0
320 goto 170
330 :
```

```
340 rem DATA FOR MORSE CHARACTERS
350 data "01", "1000", "1010", "100", "0", "0010",
"110"
360 data "0000", "00", "0111", "101", "0100",
"11", "10"
370 data "111", "0110", "1101", "010", "000", "1",
"001"
380 data "0001", "011", "1001", "1011", "1100"
```

Run the program and press the letter 'C'. The morse code for this letter is -.-. (dah di dah dit) and you should hear this played. Try pressing various letters and you should hear the morse code.

The programs operation is detailed below.

Line 50 dimensions the variable array M\$() which will hold the data representing the morse characters. The data is contained in lines 350-380 with 0's representing dots and 1's representing dashes. There are twenty six items of data representing the letters A-Z.

Lines 60-80 read the data in to the variable array M\$().

Line 110 switches of key click. Remember that key click should always be switched off before attempting to use any of the sound and music commands.

Line 140 sets the volume and line 150 resets the voices.

Line 180 asks the user to enter a letter and assigns this to variable L\$. Line 190 then converts this to upper case.

Line 220 identifies which letter the user has entered. The letters A-Z are represented by the ascii codes 65-90, and thus if we subtract 64 from the ascii code of the input letter, we can convert the letter to a number representing its position within the alphabet. The following

should make this clear.

The user enters the letter "A", so variable L\$="A"

$$L = \text{asc}(L\$)-64$$

The ascii code for the letter "A" is 65 so:

$$L = 65-64$$

Therefore L=1 which is correct as the letter "A" is the first letter in the alphabet.

Now consider what happens when the user enters the letter "C" (L\$ = "C").

$$L = \text{asc}(L\$)-64$$

The ascii code for the letter C is 67 so:

$$L = 67-64$$

Therefore L=3 and the letter "C" is the third letter of the alphabet.

Hopefully that is clear. We are simply finding the position of the letter within the alphabet.

Lines 260-300 form a *for..next* loop to play the morse code. Line 270 uses the *mid\$* command to extract the individual dots and dashes making up the morse character. Line 280 sounds the dashes and line 290 sounds the dots. Notice that the dashes have a delay of 30 whilst the dots have a delay of 10 and hence the dashes sound three times longer than the dots.

When the loop is complete and the complete letter has been produced,

the program moves on to line 310 where the volume is switched off.

Musical Notes

The *play* command can produce more than a single tone, in fact it can produce musical notes over a seven octave range. The pitch is specified as part of the *play* command and can be in the range 1-96 as shown in the table below:

		OCTAVE						
		1	2	3	4	5	6	7
Note		PITCH						
C	1	13	25	37	49	61	73	85
C#	2	14	26	38	50	62	74	86
D	3	15	27	39	51	63	75	87
D#	4	16	28	40	52	64	76	88
E	5	17	29	41	53	65	77	89
F	6	18	30	42	54	66	78	90
F#	7	19	31	43	55	67	79	91
G	8	20	32	44	56	68	80	92
G#	9	21	33	45	57	69	81	93
A	10	22	34	46	58	70	82	94
A#	11	23	35	47	59	71	83	95
B	12	24	36	48	60	72	84	96

We can use variables to represent the values for the *play* command and could thus use a *for..next* loop to play a complete octave.

Load the following:

```

10 rem PLAY COMPLETE OCTAVE
20 rem PROGRAM = A:\MUSIC\OCTAVE
30 :
40 click off
50 volume 15

```



```
60 :  
70 for M=49 to 60  
80 play 1,M,10  
90 next M  
100 :  
110 play 0,0
```

Run the program and it will play the notes.

Notice that voice number one has been specified with the *play* command in line 80 and hence the sounds are produced using a single voice. Line 110 contains a *play* command with the parameters 0,0 and this stops the tone.

We could now use the *play* command to produce a simple piano program.

Load the following:

```
10 rem PLAY PIANO USING KEYS  
20 rem PROGRAM = A:\MUSIC\PIANO  
30 :  
40 rem SET SCREEN  
50 key off : mode 0 : hide on : curs off : flash off  
60 :  
70 rem SWITCH OFF KEY CLICK  
80 click off  
90 :  
100 rem LOAD PICTURE  
110 load "a:\music\kboard.pi1"  
120 :  
130 rem SET VOLUME  
140 volume 15  
150 play 0,0  
160 :
```

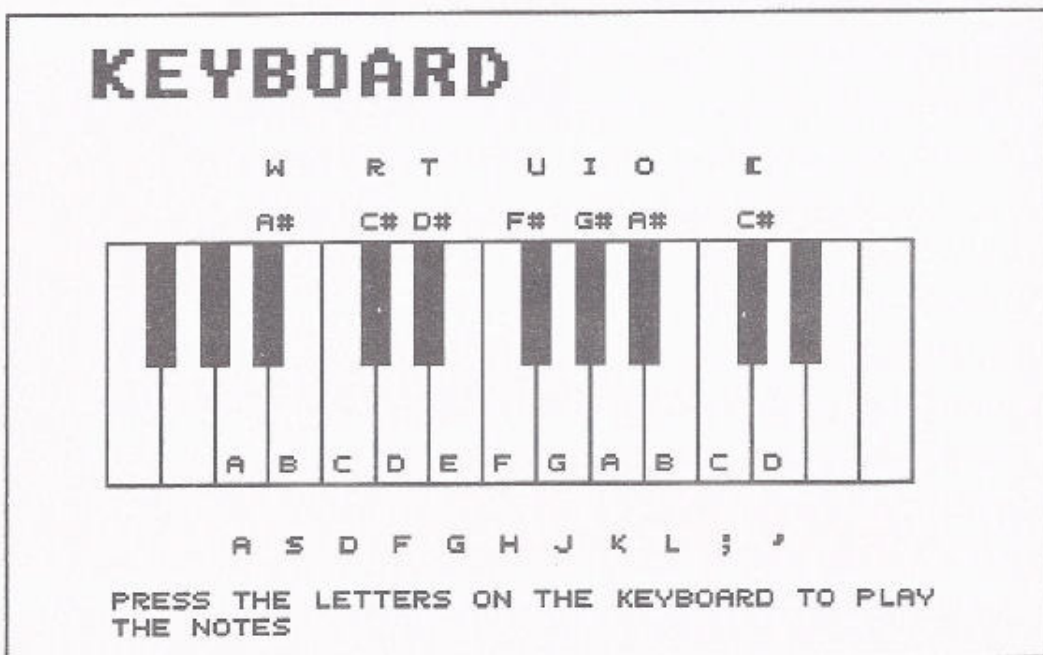
```
170 rem READ NOTE DATA IN TO VARIABLE ARRAY
180 dim K$(18),N(18)
190 for A= 1 to 18 : read K$(A) : next A
200 for A= 1 to 18 : read N(A) : next A
210 :
220 rem PLAY THE PIANO
230 N$=input$(1)
240 for A= 1 to 18
250 if upper$(N$)=K$(A) then play 1,N(A),10
260 next A
270 :
280 rem STOP THE NOTE AND GO BACK FOR NEXT
KEY
290 play 0,0
300 goto 230
310 :
320 :
330 rem NOTE DATA
340 data "A", "W", "S", "D", "R", "F", "T", "G", "H",
"U", "J", "I", "K", "O", "L", ";", "[", "'"
350 data 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,
45, 46, 47, 48, 49, 50, 51
```

Run the program and the main screen will be displayed as shown over the page.

Play a few notes using the keys as shown on the screen.

You may have worked out how the program works but we shall go over it anyway.

Line 50 sets the screen resolution and various other options that you are now familiar with.



Line 80 switches off the key click. Remember that this should always be switched off before using any of the sound and music commands.

Line 110 loads and displays a picture on the screen. The picture shows the layout of the keyboard and was produced using an art package.

Line 140 sets the volume to the maximum level and line 150 contains the *play* command. Quite often, when setting the volume, you will find that there is already some data in the sound chip and hence a note is played. Therefore line 150 stops any rogue notes that may already be playing in the sound chip.

Lines 180-200 read and assign the note data to two variable arrays. Array `K$()` maintains the keys that are used to play the notes and array `N()` holds the associated pitch number required to produce the note.

Line 230 waits for the user to press a key and lines 240-260 form a *for..next* loop to see which note has been pressed. The entered letter

is compared to those stored in the variable array `K$()`, and when a match is found, the associated pitch value in array `N()` is used to play the note. The *play* command waits for 10/50ths of a second and then moves on.

Line 290 stops the note and line 300 uses a *goto* command to send the program back to line 230 where it waits for the next key to be pressed.

Lines 340 and 350 contain the note data.

MULTIPLE VOICES

So far we have only used a single voice to produce the sounds, but complete harmonies and chords can be produced using all three voices.

Load the following:

```
10 rem PROGRAM TO PLAY CHORDS
20 rem PROGRAM = A:\MUSIC\CHORDS
30 :
40 click off : volume 15 : play 0,0
50 :
60 print "C CHORD"
70 play 1,44,1
80 play 2,49,1
90 play 3,53,1
100 wait 50
110 :
120 print "F CHORD"
130 play 1,46,1
140 play 2,49,1
150 play 3,54,1
```



```
160 wait 50
170 :
180 print "G CHORD"
190 play 1,44,1
200 play 2,48,1
210 play 3,51,1
220 :
230 wait 50
240 print "C CHORD"
250 play 1,44,1
260 play 2,49,1
270 play 3,53,1
280 :
290 wait 50
300 play 0,0
```

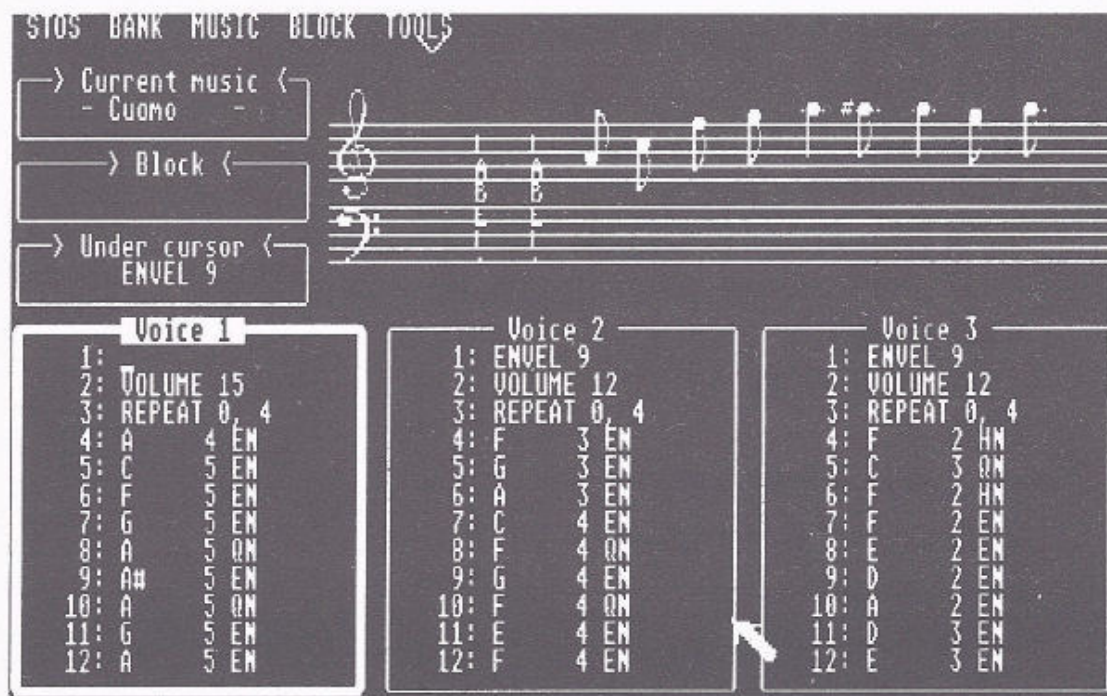
Run the program and it will play a sequence of C, F and G chords.

Notice how the three voices have been set at different pitches to produce the harmony. In addition to harmonies, the *play* command can also be used to produce complete tunes. Obviously a long tune would require a lot of *play* commands and hence STOS offers an easier alternative in the form of a music accessory. The music accessory allows the programmer to create tunes which are stored in a memory bank for easy reproduction within a program. Once the music is in a memory bank, we simply issue a *music* command and all the notes are automatically played back without having to use any *play* commands. The music accessory can be found on the STOS accessory disk and is loaded as shown below:

```
accload "music.acb"
```

The accessory program will load in to one of the accessory slots which can be viewed by pressing the help key. Select the music accessory by pressing the appropriate function key and the program

will run.




```
100 :  
110 rem PLAY CHANNEL 1 ONLY  
120 print "Currently playing voice 1"  
130 music 1  
140 wait 500  
150 :  
160 rem ADD CHANNEL 2  
170 print "Now playing voices 1 and 2"  
180 music 2  
190 wait 500  
200 :  
210 rem ADD VOICE 3  
220 print "Now playing all 3 voices"  
230 music 3  
240 wait 500  
250 :  
260 rem ADD TREMOLO TO VOICE 1  
270 print "Now adding tremolo to voice 1"  
280 music 4  
290 :  
300 print  
310 print "HOW'S ABOUT THAT THEN !!!"
```

Run the program and it will play a popular song.

The music definer accessory program allows us to produce complete compilations using all three voices. In addition to this it allows tremolo and envelope effects to be added to individual or all three voices. We shall look at these in a moment but for now let us look at the example program.

The music data is stored in memory bank 3 and the music definer accessory allows thirty two separate tunes to be defined. These are assigned the numbers 1-32 and can be accessed within our programs using the *music* command. In the above example the memory bank

contains four separate tunes numbered 1-4

Line 90 sets the tempo (the speed at which it is played) using the *tempo* command. You may like to try changing this to see the effect.

Lines 110-140 play the tune using a single voice. The first tune uses voice 1 only and is played using the *music* command as illustrated by line 130. Line 140 halts program execution for eight seconds allowing the music time to play.

Line 180 starts the second tune playing. This is the same as the first except voices 1 and 2 are now used with voice 2 adding a counter melody.

Line 230 then plays the third tune which now includes the third voice.

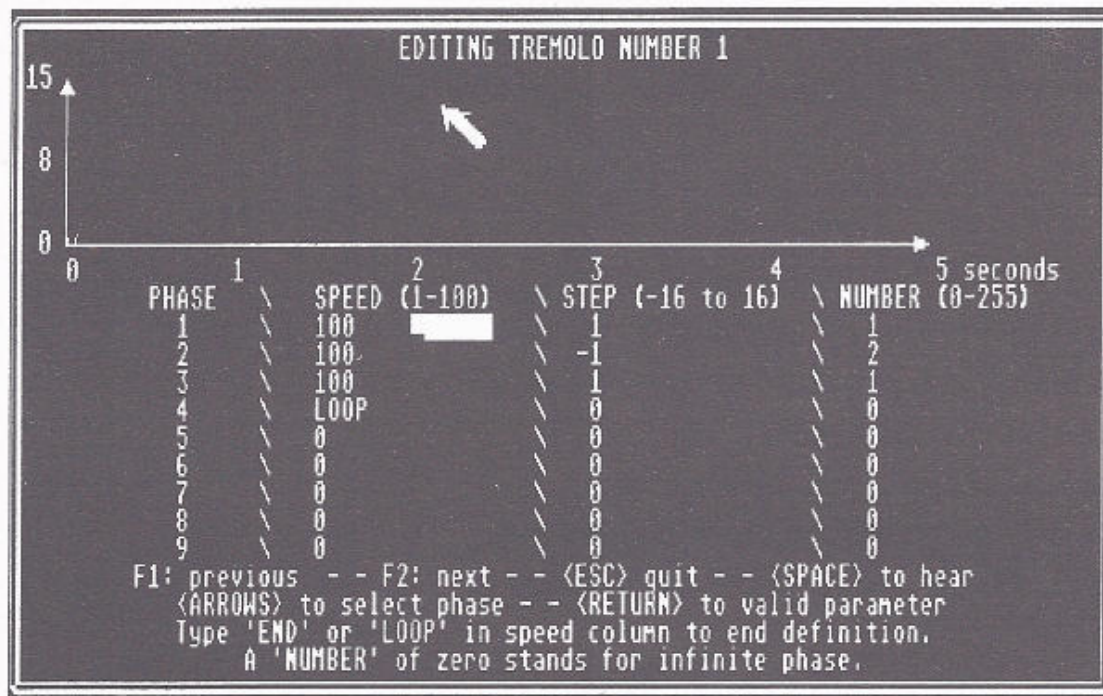
Finally line 280 plays the fourth and last tune. This is the same as tune 3 except tremolo has been added to the first voice.

The program illustrates the type of music that can be achieved using the music definer accessory.

TREMOLO

Tremolo is an effect that alters the pitch of a note as it is played. It is good for producing 'wavering' or vibrato type effects. STOS allows sixteen different tremolo effects to be used and these are defined using the tremolo editor supplied as part of the music definer. This is shown over the page.

The first eight tremolos are set by STOS although they can be re-defined by the programmer. The effect of these is illustrated by the next program.



Load the following:

```

10 rem THE TREMOLO EFFECT
20 rem PROGRAM = A:\MUSIC\TREMOLO
30 :
40 key off : mode 0
50 :
60 rem WHAT EFFECT ?
70 input "Enter tremolo value (1-8)";T
80 if T < 1 or T > 8 then print "Must be in range 1-8" :
goto 70 90 :
100 rem PLAY MUSIC
110 music T
120 :
130 wait 150
140 goto 70

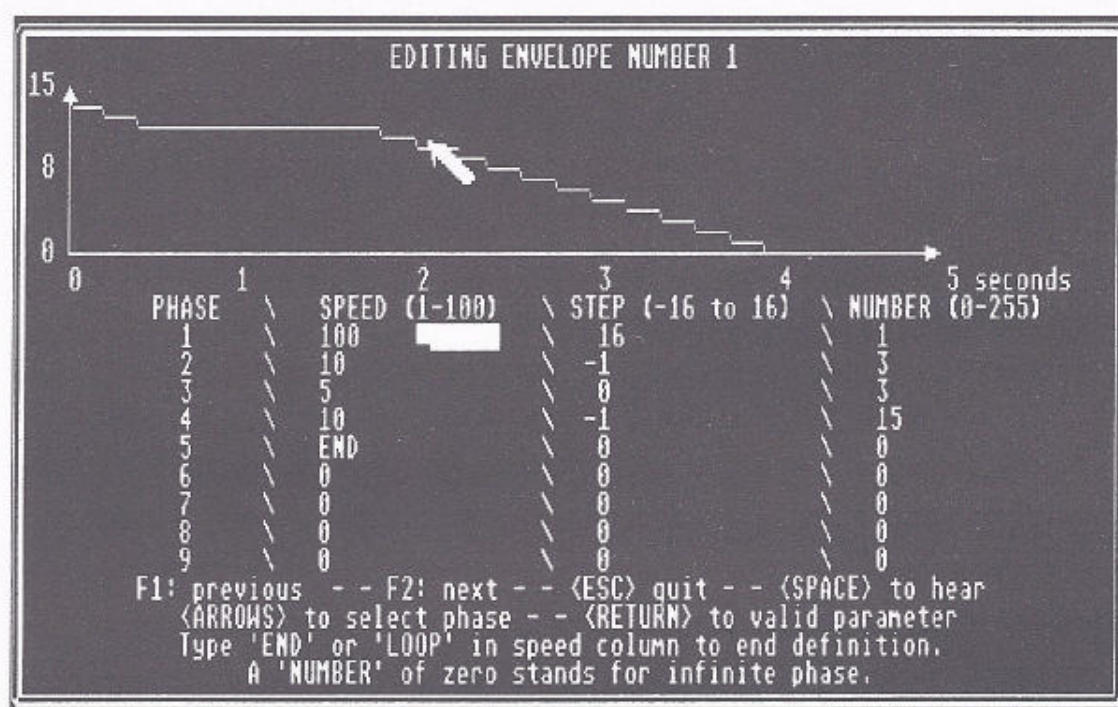
```

Run the program and try each of the values between 1 and 8. You will probably agree that these produce some very interesting effects.

The music bank (memory bank 3) contains eight separate tunes each one using a different tremolo setting.

ENVELOPES

The other effect offered by the music definer is envelopes. Envelopes control the volume of the note as it is played and once again offer a range of interesting effects. STOS allows sixteen envelopes to be used and these can be defined using the envelope editor which is supplied as part of the music definer accessory and which is shown below:



The first eight envelopes have already been set by STOS but can be re-defined by the programmer if required. These are illustrated in the following program.

Load the following:

10 rem THE SOUND ENVELOPE


```
20 rem PROGRAM = A:\MUSIC\ENVEL
30 :
40 key off : mode 0
50 :
60 rem WHAT EFFECT ?
70 input "Enter envelope value (1-8)";E
80 if E < 1 or E > 8 then print "Must be in range 1-8" :
goto 70
90 :
100 rem PLAY MUSIC
110 music E
120 :
130 wait 150
140 goto 70
```

Run the program and enter a few values to see the effect of the envelopes.

The music bank (memory bank 3) contains eight separate tunes each with a different envelope setting.

GENERATING SOUND EFFECTS

The music definer is fine for producing complete pieces of music but we may only require a few sound effects. STOS offers three default sound effects plus a number of facilities for creating your own.

Enter the following:

```
click off
boom
```

This generates a noise that could be used for an explosion.

Enter the following:

shoot

This generates a noise that could be used for a gun firing.

and finally enter the following:

bell

Although fairly basic, the bell sound is good for attracting the users attention.

We have already seen how the *play* command is used to play musical notes but STOS also offers the *noise* command which can be used to generate various effects. The format of the *noise* command is shown below:

noise VOICE, PITCH

where VOICE indicates the number of the voice that the noise should be played on, and if this is omitted, all three voices are used. PITCH indicates the pitch of the noise in the range 0-31.

To see the effect of this, load the following:

```
10 rem THE NOISE COMMAND
20 rem PROGRAM = A:\MUSIC\NOISE
30 :
40 click off
50 volume 15
60 :
70 for P=1 to 31
80 noise P
90 print "NOISE: ";P
```



```
100 wait key
110 next P
```

Run the program and keep pressing any key to hear the different pitches of noise. This sound is not very impressive on its own but we can also add envelopes to it.

Enter the following:

```
click off
volume 16
noise 10
envel 8,1000
```

This produces a train type sound.

The volume of a voice is normally in the range 1-15 but a setting of 16 indicates that an envelope should be applied to the voice. The *noise* command starts the noise and the *envel* command shapes the sound. The format of the *envel* command is shown below:

```
envel TYPE, SPEED
```

where TYPE indicates the envelope number in the range 1-15. Remember that envelopes can be defined using the music definer accessory. The speed indicates the speed of envelope in the range 0 (fast) to 66535 (slow).

A vast number of effects are available and the following program will allow you to experiment with these.

Load the following:

```
10 rem SOUND EFFECTS USING NOISE
20 rem PROGRAM = A:\MUSIC\NOISETST
```

```
30 :  
40 key off : mode 0 : click off  
50 :  
60 rem ENTER PARAMETERS  
70 input "Enter noise pitch (1-31):";P  
80 input "Enter envelope type (1-15):";E  
90 input "Enter envelope speed (0-66535):";S  
100 :  
110 rem START NOISE  
120 noise P  
130 :  
140 rem APPLY ENVELOPE  
150 envel E,S  
160 :  
170 goto 70
```

Run the program and try a few different parameters.

The envelopes can also be applied to normal musical notes using the play command. A program to let you experiment with this is shown below.

Load the following:

```
10 rem SOUND EFFECTS USING PLAY  
20 rem PROGRAM = A:\MUSIC\PLAYTST  
30 :  
40 key off : mode 0 : click off  
50 :  
60 rem ENTER PARAMETERS  
70 input "Enter note pitch (0-96):";P  
80 input "Enter envelope type (1-15):";E  
90 input "Enter envelope speed (0-66535):";S  
100 :  
110 rem PLAY NOISE
```



```
120 envel E,S  
130 play P,50  
140 :  
150 goto 70
```

Run the program and try a few different parameters.

All of the sound effects and music covered so far have been generated by the computers sound chip. Whilst we can produce some fairly good effects we have seen nothing yet to rival that found in commercial quality games. Therefore, before leaving this chapter, we shall unveil the mystery and show exactly how these amazing effects and sound tracks are achieved.

SAMPLED SOUND

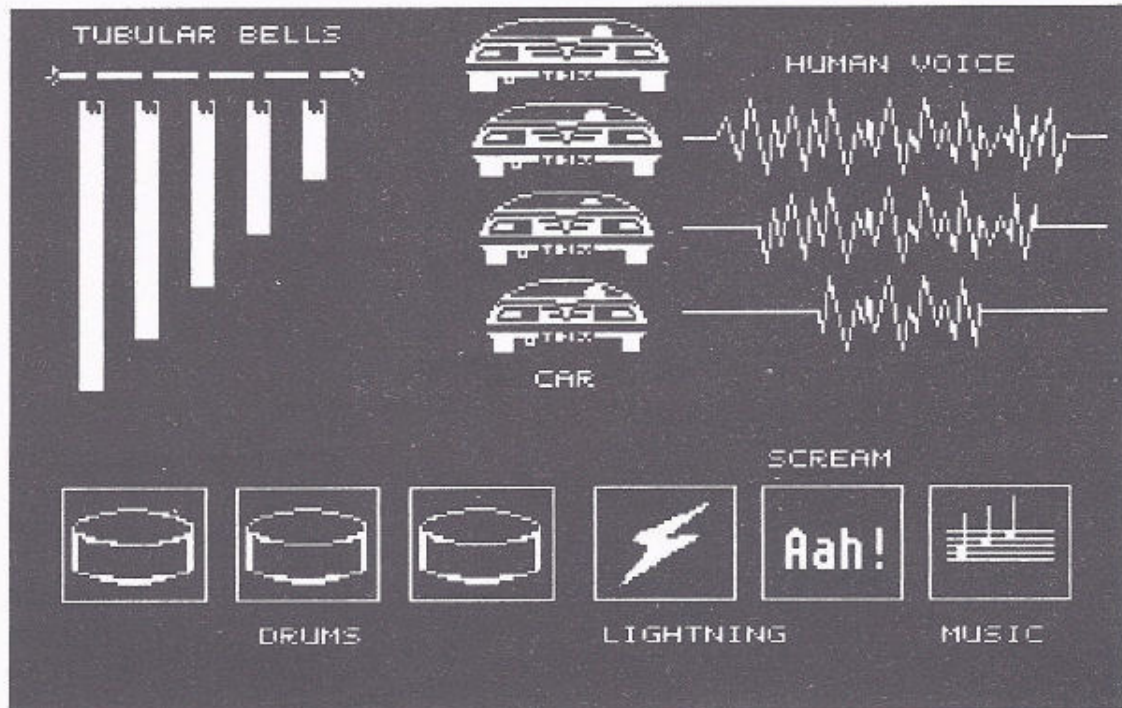
A few years ago professional games programmers payed little attention to the sound side of things because the computer could only produce very basic sounds. Nowadays, however, the music is just as important as the graphics and a lot of time and effort goes in to producing realistic sound effects and studio quality sound tracks.

When playing a game you could well believe that a tape recorder was playing in the background. Well, in a way you would be correct, as this high quality sound is achieved by 'recording' live sounds in to the computers memory and then playing them back as and when required - this recording process is known as SOUND SAMPLING. Real life sounds are digitally recorded in to the computers memory and then played back via the sound chip. This is the same method that is used with Compact disks. The sounds are digitally recorded, or sampled, and stored on to the disk.

Before continuing let us listen to an example of just what can be achieved.

Leave the STOS editor and return to the desktop. You can do this by typing the command *system*. Insert disk 2 in to the disk drive and run the program "a:\music\sample.prg"

The main screen will appear as shown below:



Various sampled sounds can be heard by moving the mouse pointer over a picture and clicking the left mouse button.

Click the mouse on each of the tubular bells and you will hear them play.

Click the mouse on each of the car pictures and you will hear a car roar past at different speeds.

Click the mouse on each of the blue 'human speech' lines and you will hear a message telling you to "keep this frequency clear".

Click the mouse on each of the three drums in the first three boxes at

the bottom of the screen.

Click the mouse on the 'lightning' box to hear a very realistic lightning crack.

Click the mouse on the 'scream' box to hear a human scream.

Finally, click the mouse on the 'music' box to hear an example of sampled music.

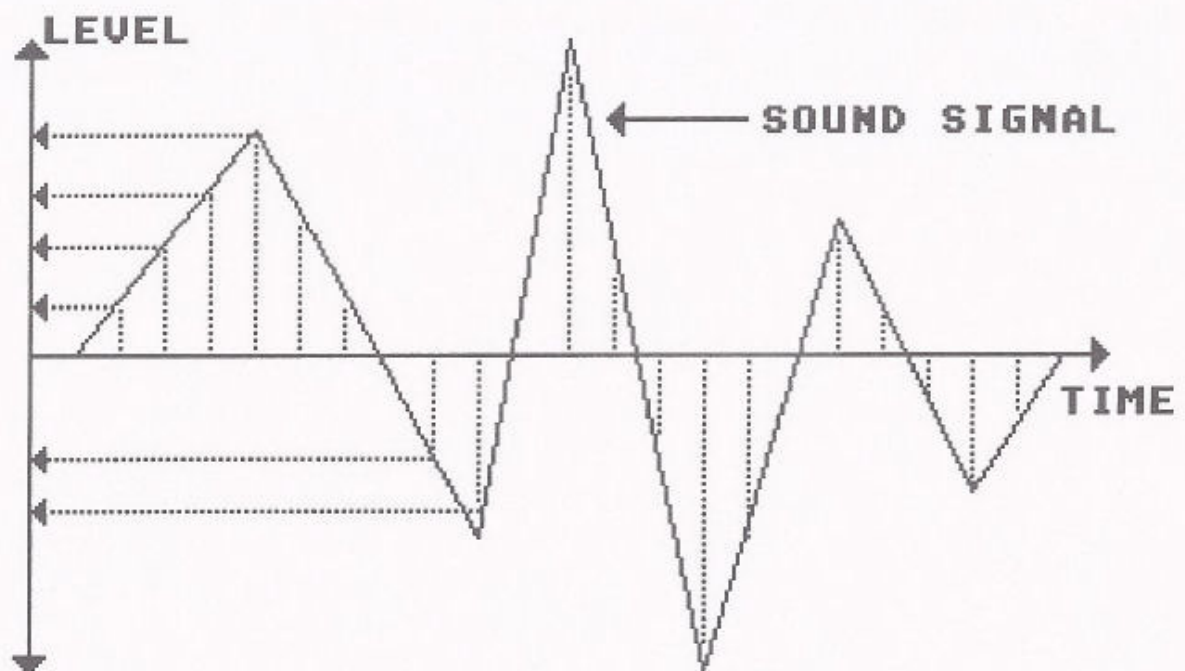
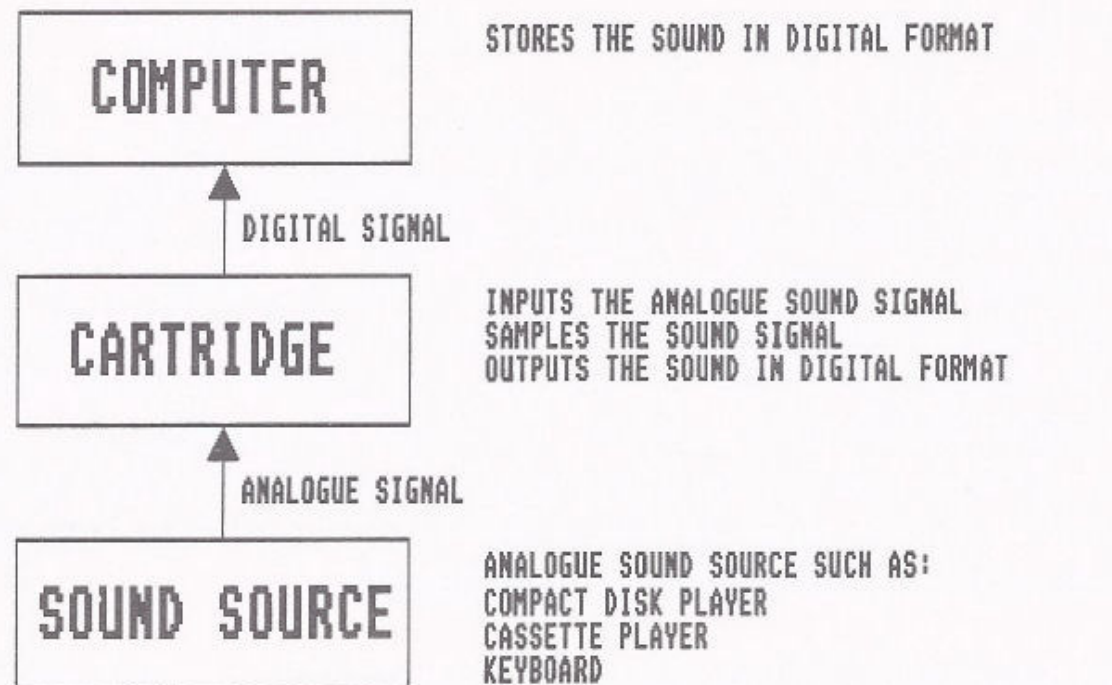
IMPRESSIVE - thought you would agree !

THE THEORY

Before we can reproduce these 'live' sounds within a program, we need a method of recording them in to the computer. The sounds need to be converted from normal analogue format to a digital format that can be stored in the computers memory. This requires external hardware, normally in the form of a small box that plugs in to the cartridge port on the side of the Atari ST. Luckily this is not expensive and complete sound sampling packages can be bought for as low as £30. The Company that produced STOS also offer STOS Maestro which is an 'add on' package that adds sound sampling capability to STOS. In addition to the hardware, it comes with lots of software that adds new commands to the STOS interpreter for playing and manipulating sound samples. The last program was produced using STOS Maestro.

To understand exactly how sound sampling works we need to look at a little theory, but do not worry as the concept is incredibly easy.

The sampling cartridge converts the analogue signal in to digital form as illustrated opposite.



The actual process of sampling involves converting the analogue sound source to a series of numbers which can then be stored in digital form. This is performed by taking a sample, or checking the

level, of the sound at regular intervals over a period of time.

The sound source is checked at regular intervals and the level expressed as a value. The rate at which the sound is sampled is known as the FREQUENCY and is normally expressed in KHz (kiloHertz). The sampling rate at which STOS Maestro samples sounds can be set in the range 5KHz to 32KHz. At the lowest end of the scale the sound will be sampled 5000 times a second and at the top end the sound will be sampled 32,000 times a second. The higher the sampling rate the better the final reproduction of the sound, but 32,000 samples would require a lot more memory than 5,000 samples and thus it is not always practical to use the highest sampling rate.

When designing a program that uses sampled sound, the programmer must think carefully about the type of sounds required and the amount of memory that these may need for storage. It is better to have a few good sounds rather than many poor sounds. Sound effects can normally pass quite well at low sampling rates whilst music requires higher sampling rates to maintain the fidelity.

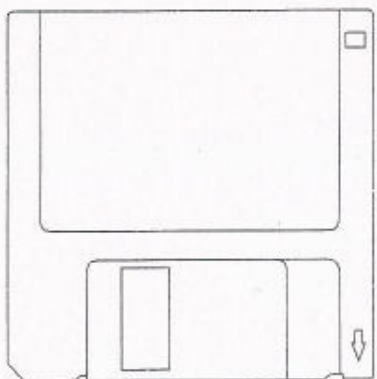
Chapter 22

File Management

In this chapter the commands relating to the management and organisation of data files are explored. There are two distinct areas regarding file management. First there are the general 'housekeeping' functions, the storage, retrieval and organisation of program files. Secondly there are the facilities for storing external files of information for use with programs. For example, in chapter 24 you will see how to write a program that acts as a name and address book, maintaining a file of names, addresses and telephone numbers separate to the main program.

You are probably already aware that, when the computer is switched off, it loses the contents of its memory. If you cast your mind back you may remember the *unnew* command. This recovers a program that has been erased using the *new* command, but when the computer is switched off, the contents of memory cannot be recovered. When writing a long and complicated program it is very important indeed that you save the program regularly to disk. The author normally goes one step further and saves the program to two separate disks just in case one should become damaged or corrupt. This may seem rather over cautious but you never know when a power cut may strike!

FLOPPY DISKS



The Atari has an in-built 3.5" floppy disk drive which is used as a storage medium for files. You may wonder why the disks are referred to as 'floppy' when in fact they look quite solid. A standard 3.5" disk is shown opposite, and if you slide back the metal, protective, shutter you will see that

the actual disk is indeed of a 'floppy' material. The ST allows a second floppy disk drive to be added and also a hard disk drive. The technicalities of the disk drives are beyond the realms of this course but a hard disk drive can be considered as a very large floppy disk which has capacity to store a lot more information and the ability to read and write the information at very high speeds.

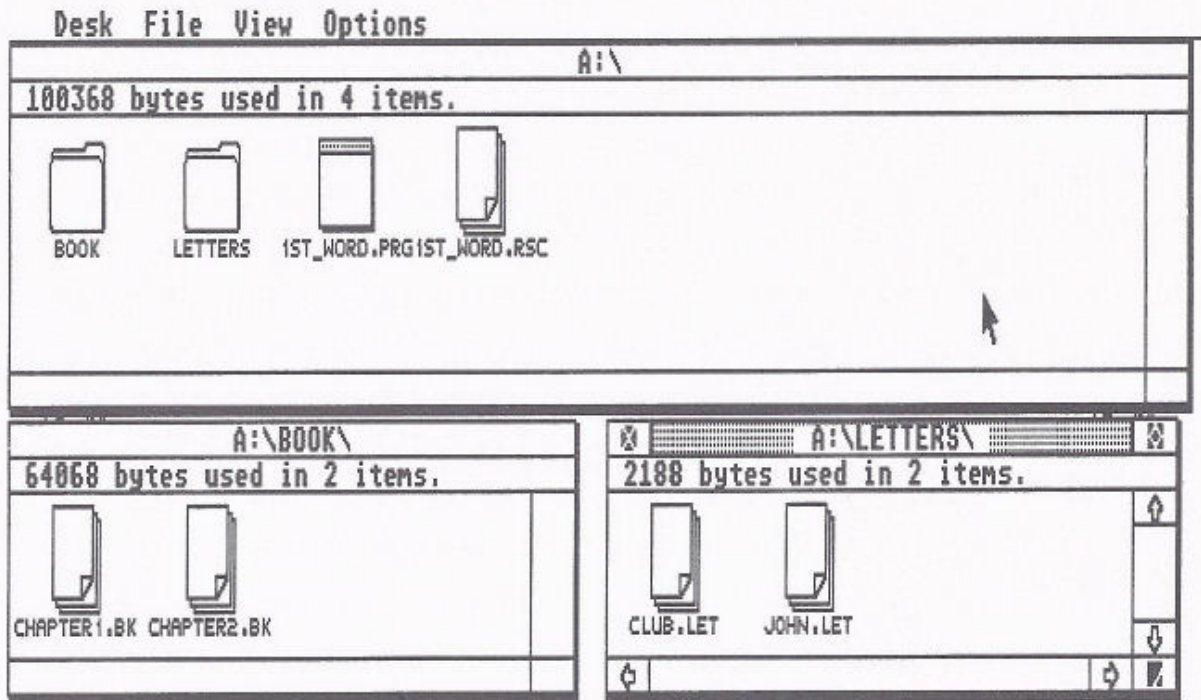
You may have heard disk drives referred to as single and double sided. All 3.5" disks nowadays are double sided, that is they can store information on both sides. Early Atari computers only had single sided disk drives which meant that, although the floppy disk may be double sided, the computer could only actually use one side. Nowadays all Atari disk drives are double sided and the whole disk is available for data storage. This double sided concept is often misunderstood with the disk being compared to a record. The disk does not have to be taken out of the disk drive and physically turned over to use the other side. The computer's disk drive has two heads which can access both sides simultaneously. The computer therefore considers the disk as one large area with information being freely spread over both sides.

Information is stored on the disk in magnetic form and the disk must be formatted before it can be used. This process separates the disk into a number of areas known as tracks and sectors and writes various

'control' information that is required for keeping track of subsequent data stored on the disk.

FILES AND FOLDERS

The computer keeps a list of all the files stored on the disk in an area known as the *directory*. This is located at the beginning of the disk and files can be further divided in to areas known as *sub-directories* or *folders*. This hierarchical filing arrangement is illustrated below.



Suppose you were writing a book. The disk would contain a word processor program and an increasing number of data files as each chapter was completed. The same disk could also be used for writing letters to friends and prospective book publishers but you would not really want the letter files mixed with the book files. The answer is to set up two folders, one for book files and one for letter files. In this way the word processor could be used for a whole range of projects, each having its own folder full of files.

Looking back to the diagram on the previous page, the top box shows the main or ROOT directory of the disk and this contains three different types of icon. An icon is a symbol that represents a specific type of file.



WORDPRO.PRG

This is a program icon and indicates that the file contains program data. In this case the program is a word processor. The program can be executed by double clicking on the icon from the GEM desktop.



JOHN.LET

This is a data icon and indicates that the file contains general data for use with another program. In this case the file contains a letter which is used with the word processor program. It is quite common to see data files ending in .RSC such as that shown on the

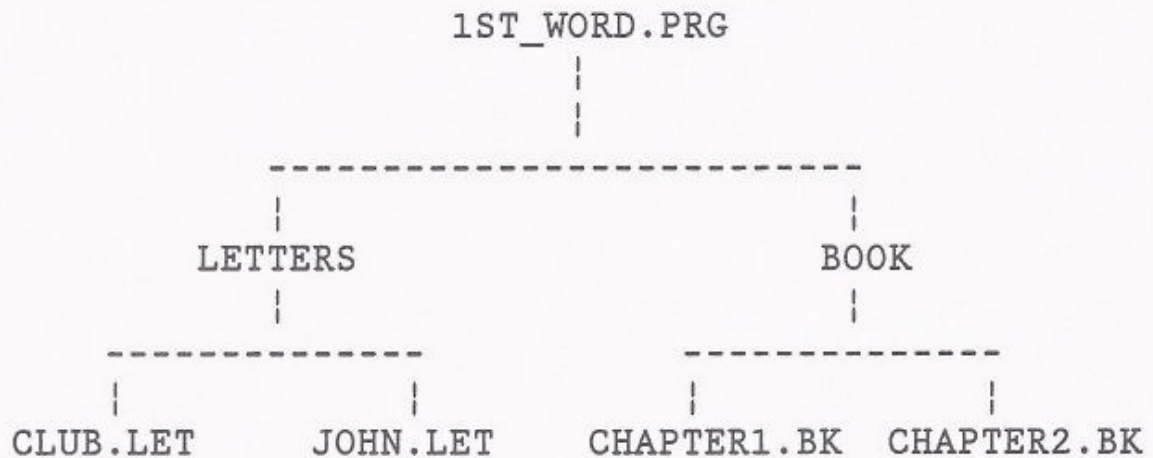
example disk here (1ST_WORD.RSC). These are known as resource files and contain information required by the main program to create dialogue boxes, drop down menus, etc.



LETTERS

This is a folder icon and is not strictly a file. It indicates a sub-directory which can be viewed by double clicking on the icon. Folders are used to group related files together. Our example disk is used for word processing and contains two separate folders named LETTERS

and BOOK. The letters folder contains any letters that are written and the book folder contains a file for each chapter of the book. The folders therefore form a tree arrangement as shown on the next page.



Look at the format of the file specification. The filename consists of two parts, an eight character name and a three character extension which is separated from the main name by a full stop. Both the name and extension can be chosen by the user, but the extension normally relates to the type of file as shown below:

.BAS indicates that the file is a basic program produced using an interpreter such as STOS.

.PRG .TOS .TTP indicate executable program files, that is files that can be executed by double clicking on them from the GEM desktop.

.PI1 .PI2 .PI3 .NEO indicate Degas and Neochrome picture files.

.MBK is used by STOS to indicate memory bank files.

.DOC .DAT normally used to indicate data files for use with word processors, databases, etc.

Using extensions in this way, different types of files can be easily grouped and identified.

The GEM desktop can be used to view the contents or directory of a disk and perform general file functions such as copying files/disks, renaming files, deleting files, creating and removing folders, etc. The

GEM desktop is not directly available within a program and so STOS offers its own commands for performing such operations. We shall now take a look at these commands and see how they can be used within our programs.

READING DIRECTORIES

The directory of a disk can be displayed using the *dir* command. This lists the directory of the current disk drive and current path (or folder) in alphabetical order. It can be used in direct mode or as a command within a program.

Place a disk in drive A: and enter the following:

dir

The directory will be displayed, and although not the same, will resemble the format of that shown below.

```
Drive A, path;  
* -----> STOS  
BASIC206.PRG    2048  
INVADERS.BAS    26700  
LION      .PI2    32066  
SUNSET     .PI1    32066  
TEMP       .BAS    1973
```

94853 bytes used.

The top line shows the disk drive and the path (folder/directory). In this case the disk drive is A and the directory shows the root (or main) directory.

Folders are the next items to be listed. On this disk there is just one folder named STOS.

The list continues with a list of files that are in the current directory. The number after each filename indicates the size of the file or the amount of space that it occupies on the disk.

STOS ends the directory listing by displaying the total amount of space used by the files in the displayed directory.

Notice that the list is printed down the screen, and when a disk contains a lot of files, the list can scroll off the top of the screen. This can be overcome by adding a /W switch to the directory command. This causes the files to be printed in table form across the page. The size of each file is omitted when displayed in this form. Enter the following:

dir/w

```
Drive A, path;
*STOS                BASIC206.PRG
INVADERS.BAS         LION      .PI2
SUNSET .PI1          TEMP      .BAS
94853 bytes used.
```

The *dir* command can be extended to show the exact range of files that are required.

Suppose we only wanted to list the STOS basic programs on the disk. We know that these files end with the extension .BAS and hence can ask for a directory displaying only files with this extension.

dir "*.bas"

```
drive A, path;
* -----> STOS
INVADERS.BAS    26700
TEMP .BAS      1973
28673 bytes used.
```


The asterisk (*) is known as a *wildcard* indicating that any 8 characters will be ok. The computer therefore lists all files with a .bas extension whilst ignoring the main file name. The wildcard can also be used as an extension. Suppose we wanted to list all files with the name ACCOUNTS:

```
dir "accounts.*"

drive A, path;
ACCOUNTS.PRG    145045
ACCOUNTS.DAT    5500

150545 bytes used.
```

Suppose we wanted to list all the Degas pictures on the disk:

```
dir "*.pi?"

drive A, path;
* -----> STOS
LION      .PI2    32066
SUNSET    .PI1    32066

64132 bytes used.
```

Degas pictures have the extension PI1, PI2 or PI3 depending on the screen resolution they are produced in. A question mark indicates that any character at that position is acceptable. The above example therefore lists all files with an extension starting with ".pi".

The last few examples have displayed the directory of the current drive, which in this case, was A. The directories of other drives can also be displayed by specify the drive letter as shown below:

```
dir "b:*.*"    this lists all the files on the external floppy drive B.
dir "c:*.bas"  this lists all the basic program files on drive C.
```

SELECTING FILES

The previous examples have shown the use of *dir* in direct mode but we shall now get back to programming and see how to access files from within a program. There are many occasions when the programmer requires the user to specify a file that is stored on disk. For example, consider a word processor program that allows a document to be saved and then recalled at a later date. Let us look at the ways in which we could re-load the document. We shall assume that all the document files have the extension ".doc" which is quite common for word processing programs.

```
10 dir "a:*.doc"  
20 input "Enter required filename";F$
```

The *dir* command in line 10 lists all of the files with a ".doc" extension on disk drive A.

The document files may be contained within a folder called DOCUMENTS and hence line 10 would be changed to:

```
10 dir "a:\documents\*.doc"
```

Notice that folders are specified by placing them within two slanted lines. Folders can themselves contain further folders and hence multiple folders can be specified. For example:

```
10 dir "a:\documents\letters\*.doc"
```

The only problem with our program so far is that we have assumed that the users files are on drive A. This may not be the case and so we need to add the ability for the user to specify the required drive.

```
10 input "Which disk drive (A,B,C, etc)";D$  
20 drive$ = D$
```



```
30 dir$ = "documents"  
40 dir "*.doc"  
50 input "Enter required filename: ";F$
```

Line 20 contains the *drive\$* function. The variable *drive\$* is maintained by STOS and contains the letter of the current drive. This variable can be modified by the programmer and hence line 20 sets the required disk drive. Line 30 contains the *dir\$* command. The variable *dir\$* is again maintained by STOS and contains the current directory. By setting the current directory to "documents", we no longer need to specify the directory as part of the *dir* command. Line 40 therefore displays the directory of the "documents" folder on the drive specified by the user.

STOS also allows the disk drive to be specified by number rather than letter with drive A being denoted drive 0. The previous program could also be written as:

```
10 input "Which disk drive (0,1,2, etc)";D  
20 drive = D  
30 dir "\documents\*.doc"  
40 input "Enter required filename: ";F$
```

Note that, if the user enters an invalid drive, STOS returns a DRIVE NOT CONNECTED error. In a bid to make programs as user friendly as possible, it would be nice if we advised the user of the drives available on the system before asking them to make a choice. STOS provides the facility for obtaining such a drive map but it is a little tricky and is not the ideal solution to the problem as we shall see later. STOS maintains a variable *drvmap* which holds a list of the connected drives in binary form.

Enter the following:

```
print bin$(drvmap)
```

Provided you do not have a hard disk drive connected, this will produce the following:

%11

Each bit of the number represents a drive with drive A starting on the right hand side. In this example the first two bits are set to one indicating that drives A and B are present. By using a loop we can test each bit and hence list all the available drives.

Load the following:

```
10 rem LIST AVAILABLE DISK DRIVES
20 rem PROGRAM = A:\FILES\DRIVES
30 for A=0 to 25
40 if btst (A,drvmap) then print "Drive: ";chr$(65 + A);"
  is connected."
50 next A
```

Run the program and it will list the disk drives currently available to your system.

The *btst* command in line 40 tests bit A of the variable *drvmap* to see if it is set to a value of 1 or a value of 0.

Note that *drvmap* always reports that both drive A and drive B are present even if drive B is not connected. This is not a fault of STOS but is due to the way in which the Atari ST disk operations work. If a user only has drive A, they still need the facility to copy disks and hence drive A doubles as both drive A and B. Therefore, if drive B is selected when it is not physically present, the computer will expect to find the disk in drive A.

Now you may feel that the previous few pages have been rather long winded. We only want to ask the user to select a file and STOS

normally offers a quick and easy solution to most programming needs. Well you would be right, STOS does actually offer a very quick and easy method to allow the user to select files from disk, but the previous examples illustrate facilities that are available should you require them in the future. All the concepts just covered can be rounded in to one command *fileselect\$*. The format of this command is shown below:

F\$ = fileselect\$(PATH\$, TITLE\$, BORDER)

where PATH\$ indicates the path or file selection criteria. TITLE\$ indicates a title to be displayed at the top of the file selector box and BORDER indicates the border style for the box. Note that both TITLE\$ and BORDER are optional and that the border styles are the same as those for windows.

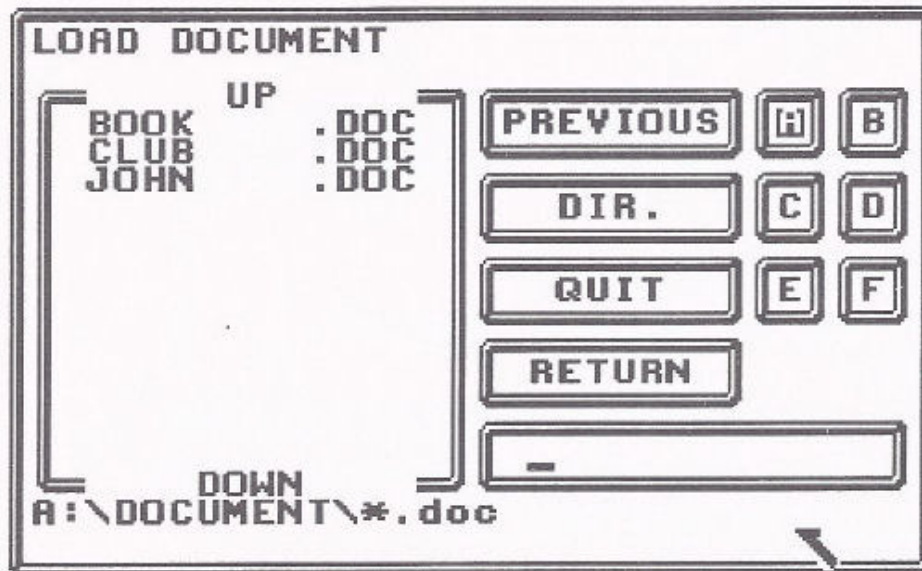
Our previous program (loading a file in to the word processor) can now be produced using a single line of code as shown below:

```
10 F$=fileselect$("\documents\*.doc","LOAD DOCUMENT", 4)
```

This produces the file selector box as shown on the next page.

At the top of the box is the title which is optional. Down the left hand side is a list of the files in the current directory. Next to this are four operations boxes which we shall look at in the minute, and on the right hand side of the box, is a list of the available disk drives. This file selector has drives A-F available of which drives C-F are hard disk partitions.

The user can select a disk drive by clicking the mouse pointer over one of the disk drive boxes. The new directory for this drive is then displayed in the file selection box. When there are too many files to fit in the file selection box, the UP and DOWN markers are used to scroll through the files. Clicking the mouse pointer on UP scrolls the



entire list up and clicking the mouse pointer on DOWN scrolls the entire list down. To select a file, the user either double clicks the mouse pointer over the required filename or types the filename and clicks the mouse pointer over the RETURN box. Double clicking the mouse pointer over the QUIT box aborts the whole operation and removes the file selector box from the screen. Double clicking the mouse pointer over the PREVIOUS box steps the current path back one directory and displays the new list of files. The DIR box allows the user to change the disk. When the disk has been changed, the user clicks the mouse pointer over the DIR box to read in and display the directory of the new disk.

If the user selects a file, the filename is returned to the assigned variable which in this case is F\$. If the user quits without selecting a file, the variable will remain empty.

To illustrate the file selector box further, let's look at another example.

Load the following:

```
10 rem DISPLAY DEGAS PICTURES
20 rem PROGRAM = A:\FILES\SELECT
30 :
40 rem SET SCREEN
50 key off : mode 0 : flash off
60 :
70 rem LOAD AND DISPLAY PICTURE
80 F$ = fileselect$("a:\pictures\*.pi1",
"LOAD A DEGAS PICTURE",12)
90 load f$
```

Place disk 1 (the disk that accompanies the course) in to the disk drive and run the program. A file selector will be displayed allowing you to choose from a list of picture files. Select one of the pictures and it will be loaded and displayed on the screen. Notice how the variable F\$ is used directly with the *load* command.

OTHER FILE COMMANDS

In this section we shall take a brief look at the other file related commands offered by STOS before moving on to data filing techniques and database applications.

LOAD

Load is used to load or transfer a file from disk in to the computers memory. STOS can handle a vast array of files including program information, picture files, sound files, sprite files, etc.

```
load "game.bas"
```

```
load "c:\programs\game.bas"
```

Notice how the folder and drive can be specified.

If you are not quite sure where the required file is, you can use the *fload* command. This displays a file selector box and allows the user to 'look around', find and load the correct file. To call the file selector enter the following:

```
fload ""
```

SAVE

Save is used to save the information in the computers memory to a file on disk. The format of *save* is as follows:

```
save "game.bas"
```

```
save "c:\programs\game.bas"
```

Note again the use of drive and folder specifications. Note also that the contents of memory banks can be saved to disk. For example, suppose that you had a low resolution, degas picture file in memory bank 14. This could be saved to disk as shown below:

```
save "picture.pil",14
```

If you would rather use the file selector to save a file you can call this as shown below:

```
fsave ""
```

RENAME

Rename allows the name of a file already existing on the disk to be changed. The format of *rename* is as follows:

```
rename OLDNAME$ to NEWNAME$
```


rename "a:game.bas" to "a:invaders.bas"

KILL

Kill allows a file to be deleted from the disk. Note that the file cannot be recovered so use this with care. There does seem to be a slight problem when the *kill* command tries to kill a file on a different disk drive to that currently set. In this situation the command attempts to kill the file on the current disk rather than that specified by the user. To ensure that the command works correctly, change to the appropriate directory before killing the file. For example, suppose you want to kill the file "chapt1.doc" which is inside a folder named "documents".

You could enter the following:

```
kill "a:\documents\chapt1.doc"
```

but to guarantee success you would be better to change to the required disk drive and directory first. For example:

```
drive = 0  
dir$ = "documents"  
kill "chapt1.doc"
```

MKDIR

The make directory facility allows a new directory to be created on the disk. For example.

```
md "myfiles"  
  
md "c:newprogs"
```

RMDIR

The remove directory facility allows a directory to be deleted from the

disk. Note that the directory must be empty else STOS will return an error message.

PREVIOUS

This steps the current directory back one level. For example, suppose the current directory is "a:\database\accounts\". If a *previous* command is issued, the current directory will step back to the previous level and become "a:\database".

DFREE

Disk Free returns the amount of free space on the current disk. This can be used direct or assigned to a variable as shown below:

```
print dfree
```

```
D = dfree
```

Well that just about covers the general housekeeping facilities. We shall now move on to see how to store and retrieve our own data.

SEQUENTIAL AND RANDOM ACCESS FILING

STOS offers two filing systems - *sequential access* and *random access*. Both store information on the disk in list form but differ in the way that information is accessed and updated. Sequential files only allow the information to be read from the disk in exactly the same order that it was originally written. This is fine for many applications but suppose that you wish to amend information stored half way through the file. You would have to load the whole file, make the changes and then save the whole file back to disk again. Random access, as the name suggests, offers random access to the file. This allows a single item of information to be loaded, amended and saved back to disk without affecting the rest of the file.

When discussing files we often stumble across the terms *database*, *record* and *field*. A database is the name given to a collection of related data. For example, you could have a database that maintained peoples names and address or a database that kept track of your record collection, etc. A *field* is a single element of information and a *record* is a collection of related fields. The collection of records then form the *database*.

A database containing names and addresses may contain records as shown below:

```
-----RECORD 1-----  
Mr G Brough          <---Field 1  
1 Avondale Road      <---Field 2  
Brighton              <---Field 3  
Essex                 <---Field 4  
  
TEL: 0954 67384      <---Field 5  
-----
```

To illustrate the way in which files are created and handled, we shall write a program to maintain a database of peoples names and ages. We shall start by using sequential filing. Suppose that we want to store a list of information as shown below:

<u>Name</u>	<u>Age</u>
John	18
Tania	24
Sue	36
Peter	55

The database will thus contain 4 records each containing two fields (name and age).

The following examples will need to write information to a disk. You

should therefore format a new disk and place this in disk drive A before running the programs.

The following program opens a file and stores the above information on the disk.

Load the following:

```
10 rem OPENING A FILE FOR OUTPUT
20 rem PROGRAM = A:\FILES\SEQ_OPEN
30 :
40 rem READ THE DATA
50 dim NAME$(4),AGE(4)
60 for A=1 to 4
70 read NAME$(A),AGE(A)
80 next A
90 :
100 rem OPEN THE FILE
110 open out #1,"name.dat"
120 for A=1 to 4
130 print #1,NAME$(A)
140 print #1,AGE(A)
150 next A
160 :
170 rem CLOSE THE FILE
180 close #1
190 :
200 rem DATA
210 data "John", 18, "Tania", 24, "Sue", 36, "Peter",
```

55

Run the program and you will see the disk drive light illuminate as the information is written to the disk file. To confirm that the information has been saved on the disk, use the *dir* command to look at the disk directory.

Lines 50-80 read the data and assign it to variable arrays NAME\$() and AGE().

Line 110 opens a sequential file. The *open out* command indicates that the file should be opened to output information. The number (#1) following the *open out* command is known as the *channel number*. STOS allows multiple files to be opened simultaneously and these are identified by the channel number. For example, if three files were opened these would be numbered #1, #2 and #3.

Lines 120-150 write the information to the file. The information is printed, using the *print* command, to channel #1.

When all the information has been written, the file is closed using the *close #1* command.

We now have the file on disk. The next program will open the file and read the contents back in to the computer.

Load the following:

```
10 rem READ FROM SEQUENTIAL FILE
20 rem PROGRAM = A:\FILE\SEQ_READ
30 :
40 rem OPEN THE FILE FOR INPUT
50 open in #1,"name.dat"
60 :
70 rem READ THE INFORMATION
80 dim NAME$(4),AGE(4)
90 for A = 1 to 4
100 input #1, NAME$(A)
110 input #1, AGE(A)
120 next A
130 :
140 rem CLOSE THE FILE
```

```
150 close #1
160 :
170 rem DISPLAY THE INFORMATION
180 for A = 1 to 4
190 print NAME$(A),AGE(A)
200 next A
```

Run the program and the information will be input from the disk file and displayed on the screen.

Line 50 opens the file for input using the *open in* command, lines 80-120 input the data using the *input #1* command, line 150 closes the file and lines 180-200 display the information on the screen.

Suppose now that Tania has a birthday and her age changes from 24 to 25. The information needs to be changed. The only way to do this is to read the whole file in to memory, make the change and then save the whole file back to disk again.

Load the following:

```
10 rem UPDATE SEQUENTIAL FILE
20 rem PROGRAM = A:\FILES\SEQ_UPD
30 :
40 rem INPUT THE FILE
50 open in #1,"name.dat"
60 dim NAME$(4), AGE(4)
70 for A = 1 to 4
80 input #1,NAME$(A)
90 input #1,AGE(A)
100 next A
110 close #1
120 :
130 rem UPDATE INFORMATION
130 AGE(2) = 25
```



```
140 :  
150 rem SAVE UPDATE FILE BACK TO DISK  
160 open out #1,"name.dat"  
170 for A = 1 to 4  
180 print #1,NAME$(A)  
190 print #1,AGE(A)  
200 next A  
210 close #1
```

Run the program. You will not see anything happen on the screen but the data will be read from, amended and written back to the disk. To check that the data has actually been amended, read the data in using the previous program again. You can do this as shown below.

Enter the following:

```
load "a:\files\seq_read  
run
```

The list of information will be displayed on the screen and you will see that Tania's age is now listed as 25.

Sequential access files are ideal for storing lists of information but do present problems when information needs updating. There is no facility for changing single elements of information and the file must always be considered as a whole. Random access filing overcomes this problem but does require a little more effort in setting up. We shall now tackle the previous problem again using random access techniques.

Load the following:

```
10 rem OPEN A RANDOM ACCESS FILE  
20 rem PROGRAM = A:\FILES\RAN_OPEN  
30 :
```

```
40 rem OPEN FILE
50 open #1,"r","name.dat"
60 :
70 rem DEFINE FIELDS
80 field #1, 5 as NAME$, 2 as AGE$
90 :
100 rem READ DATA AND WRITE TO FILE
110 for A = 1 to 4
120 read NAME$
130 read AGE$
140 put #1,A
150 next A
160 :
170 rem CLOSE FILE
180 close #1
190 :
200 rem DATA
210 data "John", "18", "Tania", "24", "Sue", "36",
"Peter", "55"
```

Run the program and it will create a random access file named "name.dat" containing the name and age information.

The first thing to notice is that the age is represented as a string rather than a straight number. Random access files can only store text information and thus any numeric information must be converted to a string before writing to the file. This could be performed using the *str\$* function as shown below:

```
100 A=24
110 A$=str$(A)
```

For this example we have listed the age as a string within the line of data so no further conversion is necessary.

Line 50 opens the file. Notice the letter "R" which indicates that the file is a random access file. Sequential filing requires the *open in* or *open out* command depending on the direction of the data but random access files, once opened, can be freely read from and written to.

The next stage is to define the fields and this is carried out by line 80. When using sequential files we can simply print the variables to the file without worrying about their lengths. Random access files are different, we have to define the exact size of, and the exact format of the variables. Let's take a closer look at line 80.

```
80 field #1, 5 as NAME$, 2 as AGE$
```

This indicates that the file should have two fields - NAME\$ and AGE\$. The maximum length of NAME\$ will be five characters and the maximum length of AGE\$ will be two characters. These values are set to represent the longest variable that you wish to store. Our data consists of the names John, Tania, Sue and Peter, the longest names being Tania and Peter. These consist of five letters and thus the field must allow a maximum of five characters. If the program were to allow new names to be added then we would have to set the field to a higher value so as to allow for any unknown names that may be added at a later stage.

Once the fields have been defined we can start writing data to the file. Lines 110-150 form a *for..next* loop which reads the information from the data list and writes this to the file. Look carefully at line 140. The *put* command puts the information in to the file. The format of this command is shown below:

```
put CHANNEL, RECORD
```

where CHANNEL indicates the channel number for the file and RECORD indicates the record number. Remember that a record is a collection of related fields. In this example the file will contain four

records each of two fields.

Notice how the loop variable A has been used to represent the record number. At the end of the loop the file will contain records 1-4.

Line 180 closes the file.

Let us now write a routine to load the data back in and display it on the screen.

Load the following:

```
10 rem READ RANDOM ACCESS FILE
20 rem PROGRAM = A:\FILES\RAN_READ
30 :
40 rem OPEN FILE
50 open #1,"r","name.dat"
60 :
70 rem DEFINE FIELDS
80 field #1,5 as NAME$,2 as AGE$
90 :
100 rem READ INFORMATION
110 for A=1 to 4
120 get #1,A
130 print NAME$,AGE$
140 next A
```

Run the program. The data will be read from the file and displayed on the screen.

Line 50 opens the file and line 80 defines the fields. The fields must be defined each time the file is opened irrespective of whether you are reading or writing to the file.

Lines 110-140 read and display the information. Line 120 uses the *get*

command to get a record from the file and line 130 prints the information on the screen.

Random access files are rather fussy when compared to sequential files. The exact record structure has to be defined using the *field* command and the data can only consist of strings. However, you will probably agree that the extra effort required to produce such a file is well worth it when you consider how easy they are to update.

Load the following:

```
10 rem UPDATE RANDOM ACCESS FILE
20 rem PROGRAM = A:\FILES\LAN_UPD
30 :
40 rem OPEN FILE
50 open #1,"r","name.dat"
60 field #1,5 as NAME$,2 as AGE$
70 :
80 rem UPDATE TANIAS AGE
90 NAME$ = "Tania"
100 AGE$ = "25"
110 put #1,2
120 :
130 rem CLOSE FILE
140 close #1
```

Run the program and Tanias age will be changed from 24 to 25. To check that the update has actually taken place, load the previous program and read the file.

Enter the following:

```
load "a:\files\ran_read"
run
```

Random access files can access and update individual records without affecting the rest of the file.

Line 50 opens the file and line 60 defines the fields. Lines 90 and 100 assign the new information to the fields and line 110 puts the information to the file. The record number is 2 which represents the record for Tania. Line 140 then closes the file.

SEQUENTIAL OR RANDOM ACCESS

The choice of filing system will depend on the type of program. The majority of applications on the Atari ST use sequential filing with the complete data file being loaded and worked upon within memory. Obviously memory based data can be accessed a lot faster than disk based data and thus relatively small data storage requirements can be met through sequential filing.

Random access is normally reserved for larger database projects where a large amount of data is maintained on disk rather than in memory. Access to the data is slightly slower but individual records can be loaded and amended without the whole database having to be read in to memory.

There is one other area to consider when designing database applications and that is security. It is common practice, when using sequential filing, to load a file in to memory, update the data contained within and only save the file back to disk after a number of alterations have been made. Suppose though that there is a power cut. Whilst the original data may still be on disk, you will have lost all of the work in memory. This is another reason why random access is the preferred method for larger systems as information is saved back to disk after each transaction and thus the data is relatively safe.

Chapter 23

Creating a Word Processor

In this chapter we shall see how sequential filing can be used to produce a simple word processor program. The program can be used for writing and printing short letters and documents. These documents may be saved to disk and recalled at a later date for further editing or printing. As usual, we start by defining the exact requirements of the program.

DEFINITION

The program shall allow the computer to be used as a type writer. Documents can be entered using the keyboard and can be saved to disk or printed on a printer. Saved documents can be recalled from disk so that editing may continue at a later date. To perform these operations the program shall offer the following options:

1. Load a file.
2. Save a file.

3. Start a new document.
4. Print a document.
5. Quit the program.

These shall be selected from a pop-up menu which appears at the top left corner of the screen when the ESC key is pressed. The box can be removed from the screen by selecting an option or by pressing the ESC key a second time.

DESIGN

The program will require the following operational modules:

- (1) Set screen and document area.
- (2) Display option box.
- (3) Monitor editor operations.
- (4) Load a file from disk.
- (5) Save a file to disk.
- (6) Start a new document.
- (7) Print document.
- (8) Quit the program.

Let's load the program.

Load the following:

```
10 rem TYPE WRITER
20 rem PROGRAM = A:\TYPE\TYPE
30 :
40 rem SET SCREEN
50 key off : mode 1 : flash off : hide on
60 palette $0,$777,$77,$742
70 BLACK=0 : WHITE=1 : BLUE=2 : BROWN=3
80 :
```

```
90 rem OPEN WINDOW FOR EDITING DOCUMENT
100 pen BROWN
110 locate 0,24
120 centre "Press ESC to display option list"
130 pen WHITE
140 windopen 1,0,0,80,23
150 title " TYPE WRITER "
160 :
170 rem DISPLAY OPTIONS BOX
180 gosub 1100
190 :
200 rem WAIT FOR USER INPUT
210 while K$ = ""
220 K$ = inkey$
230 wend
240 :
250 rem CHECK FOR CURSOR UP KEY
260 if asc(K$)=0 and scancode=72 and Y>0 then
dec Y : locate X,Y
270 :
280 rem CHECK FOR CURSOR DOWN KEY
290 if asc(K$)=0 and scancode=80 and Y<19 then
inc Y : locate X,Y
300 :
310 rem CHECK FOR CURSOR LEFT KEY
320 if asc(K$)=0 and scancode=75 and X>0 then
dec X : locate X,Y
330 :
340 rem CHECK FOR CURSOR RIGHT KEY
350 if asc(K$)=0 and scancode=77 and X<76 then
inc X : locate X,Y
360 :
370 rem CHECK FOR ESC KEY
380 if K$ = chr$(27) then gosub 1100
390 :
```



```
400 rem CHECK FOR RETURN KEY
410 if K$ = chr$(13) and Y < > 19 then inc Y : X=0 :
locate X,Y
420 :
430 rem CHECK FOR BACKSPACE KEY
440 if K$ = chr$(8) and X > 0 then dec X : locate X,Y
: print " "; : locate X,Y
450 :
460 rem CHECK FOR DEL KEY
470 if asc(K$)=127 then for X1=X to 76 : locate
X1,Y : print chr$(scrn(X1 + 1,Y)); : next X1 : locate X,Y : goto
530
480 :
490 rem CHECK FOR NORMAL CHARACTERS
500 if asc(K$) > 31 and X < 76 then print K$; : inc X :
goto 530
510 if asc(K$) > 31 and X = 76 and Y < > 19 then print
K$ : X=0 : inc Y
520 :
530 K$ = ""
540 goto 210
550 :
560 rem LOAD A FILE
570 show on
580 F$ = file select$("* .doc", "LOAD FILE")
590 hide on
600 if F$ = "" then return
610 clw
620 open in #1, F$
630 for Y=0 to 19
640 input #1, INFO$
650 print INFO$
660 next Y
670 close #1
680 X=0 : Y=0 : home
```

```
690 return
700 :
710 rem SAVE A FILE
720 show on
730 F$ = file select$("* .doc", "SAVE FILE")
740 hide on
750 if F$ = "" then return
760 open out #1, F$
770 for Y1 = 0 to 19
780 INFO$ = ""
790 for X1 = 0 to 76
800 INFO = scrn(X1, Y1)
810 INFO$ = INFO$ + chr$(INFO)
820 next X1
830 print #1, INFO$
840 next Y1
850 close #1
860 return
870 :
880 rem NEW DOCUMENT
890 clw
900 X = 0 : Y = 0 : home
910 return
920 :
930 rem PRINT DOCUMENT
940 for Y1 = 0 to 19
950 for X1 = 0 to 76
960 INFO = scrn(X1, Y1)
970 INFO$ = INFO$ + chr$(INFO)
980 next X1
990 lprint INFO$
1000 INFO$ = ""
1010 next Y1
• 1020 return
1030 :
```



```
1040 rem QUIT PROGRAM
1050 windel 1
1060 cls
1070 print "GOODBYE-Thank you for using Type
Writer"
1080 end
1090 :
1100 rem DISPLAY OPTIONS BOX
1110 pen BLUE
1120 windopen 2,0,0,30,12,4
1130 title " TYPE WRITER OPTIONS "
1140 print : print " 1...Load File"
1150 print " 2...Save File"
1160 print " 3...New File"
1170 print " 4...Print File"
1180 print " 5...Quit"
1190 print : print " ENTER OPTION"
1200 OPT$=input$(1)
1210 OPT=val(OPT$)
1220 if asc(OPT$)=27 then windel 2 : return
1230 if OPT < 1 or OPT > 5 then 1200
1240 windel 2
1250 on OPT gosub 560,710,880,930,1040
1260 return
```

CONSTRUCTION AND CODING

Run the program and select a few options. There is a demonstration file on the disk and this can be loaded as follows:

1. Select option 1 - Load File.
2. Using the file selector, select the document named "TYPE.DOC".
3. The document can be edited using the cursor keys, delete key, etc. and can be printed using the PRINT FILE OPTION.

Let's look at the code for the program and its operation should soon become clear.

SET SCREEN AND DOCUMENT AREA

```
40 rem SET SCREEN
50 key off : mode 1 : flash off : hide on
60 palette $0,$777,$77,$742
70 BLACK=0 : WHITE=1 : BLUE=2 : BROWN=3
80 :
90 rem OPEN WINDOW FOR EDITING DOCUMENT
100 pen BROWN
110 locate 0,24
120 centre "Press ESC to display option list"
130 pen WHITE
140 windopen 1,0,0,80,23
150 title " TYPE WRITER "
160 :
170 rem DISPLAY OPTIONS BOX
180 gosub 1100
```

Line 50 switches off the function key window, sets the screen resolution to medium, switches off colour flashing and hides the mouse pointer. Line 60 sets the colour palette and line 70 assigns variable names to the colours.

Lines 100-150 display the main editing area. Line 100 changes the pen colour to brown, line 110 locates the cursor at the bottom of the screen and line 120 displays the message along the centre of the line. Line 130 changes the pen colour to white, line 140 opens a window and line 150 assigns a title for the window. Notice that the "Press ESC to display option list" message is displayed outside of the main window. This allows the main window to be cleared whilst maintaining this message at the bottom of the screen.

To allow the user to select various options, we are going to create a 'pop up' option list that is activated and deactivated using the ESC key. The code for this shall form a subroutine so that it may be called as and when required. When the program is first run we want to display the option list so line 180 passes control to the sub routine.

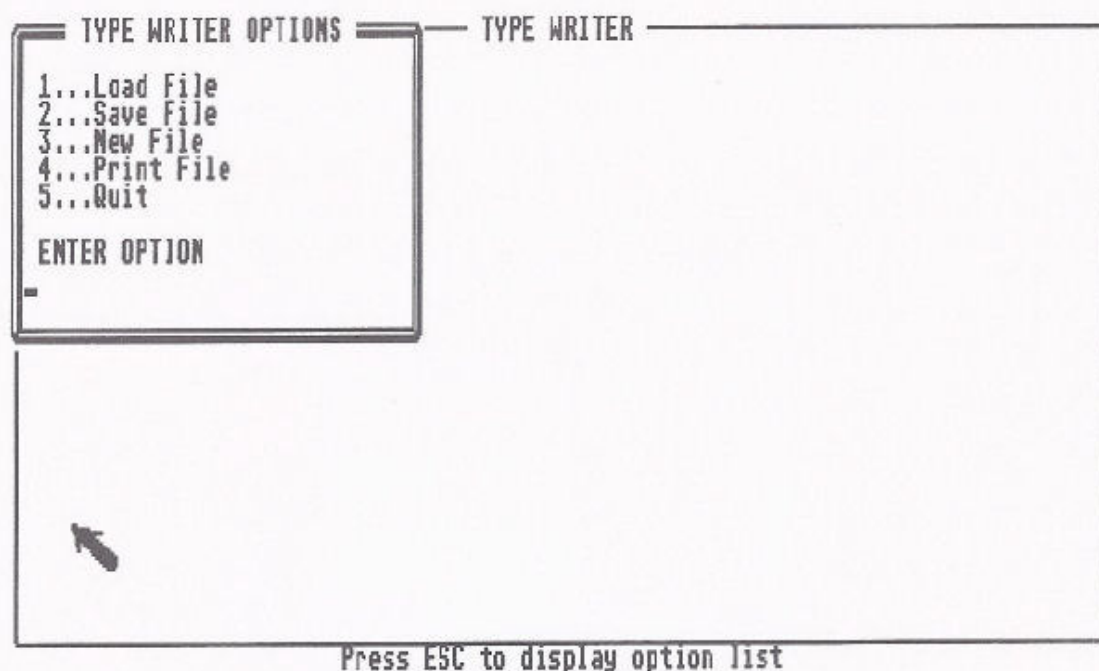
DISPLAY OPTION BOX

```
1100 rem DISPLAY OPTIONS BOX
1110 pen BLUE
1120 windopen 2,0,0,30,12,4
1130 title " TYPE WRITER OPTIONS "
1140 print : print " 1...Load File"
1150 print " 2...Save File"
1160 print " 3...New File"
1170 print " 4...Print File"
1180 print " 5...Quit"
1190 print : print " ENTER OPTION"
1200 OPT$=input$(1)
1210 OPT=val(OPT$)
1220 if asc(OPT$)=27 then windel 2 : return
1230 if OPT < 1 or OPT > 5 then 1200
1240 windel 2
1250 on OPT gosub 560,710,880,930,1040
1260 return
```

Line 1110 sets the pen colour to blue. Line 1120 opens a small window in the top left corner of the screen and line 1130 assigns a title for the window. Lines 1140-1180 print the various options inside the window. Line 1200 waits for a key to be pressed and assigns the entered character to variable OPT\$. Line 1210 then assigns the numeric value of variable OPT\$ to variable OPT. Line 1220 checks to see if the ESC key has been pressed. The ascii code for the ESC key is 27, so we compare the ASCII value of variable OPT\$ with the

value 27. If the ESC key has been pressed, the window is removed from the screen and program execution passed back to the calling program. If the ESC has not been pressed, the program continues on to line 1230. Line 1230 checks that the entered information is within the desired range, and if not, returns the program back to line 1200 where it waits for another key press. If the information is in the range 1 to 5 then the window is removed from the screen and line 1250 passes control to the appropriate section of code for the chosen option.

The opening display with the main window and the pop-up option box is shown below:



If the user selects option 1 then program execution passes to the load routine.

LOAD A FILE FROM DISK

```
560 rem LOAD A FILE
570 show on
```

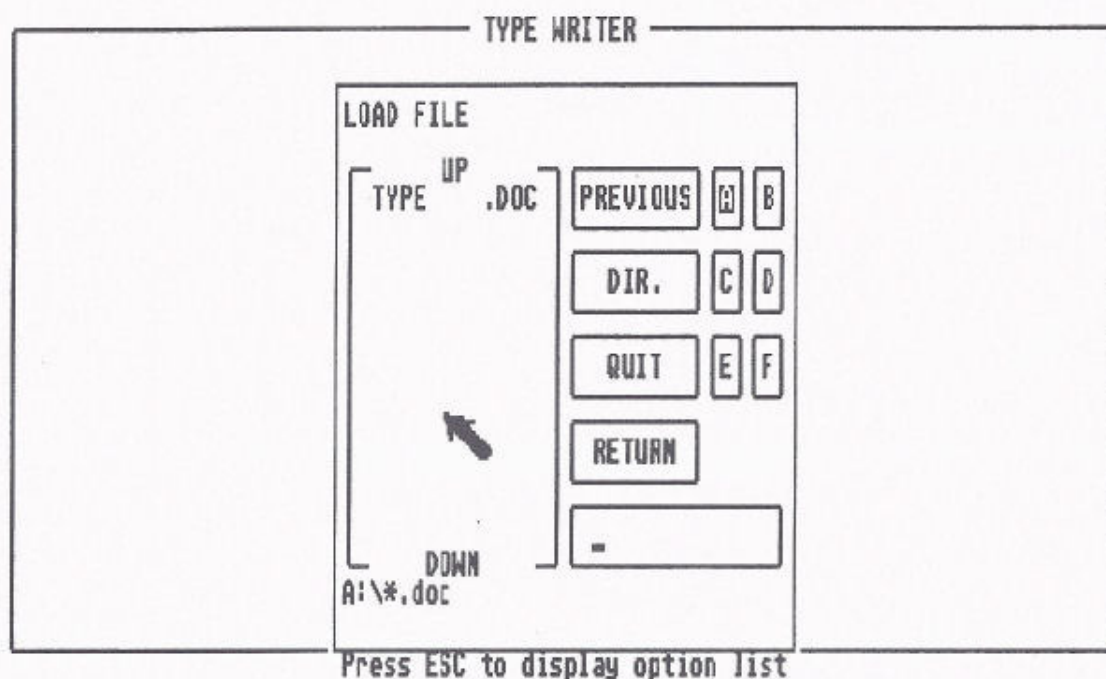


```

580 F$=file select$("*.*", "LOAD FILE")
590 hide on
600 if F$="" then return
610 clw
620 open in #1,F$
630 for Y=0 to 19
640 input #1,INFO$
650 print INFO$
660 next Y
670 close #1
680 X=0 : Y=0 : home
690 return

```

Line 570 shows the mouse pointer and line 580 displays a file selector box so that the required file may be chosen. This is illustrated below:



word processor documents to have a DOC extension so we have kept with this tradition in our examples. The file selector therefore lists only those files with a ".DOC" extension and assigns the entered filename to the variable F\$. Line 600 checks to see if a filename has been entered and ends the subroutine if no name has been entered. If a valid name has been entered, the program moves on to line 610. Line 610 clears the main window to make way for the new document. Line 620 opens the file for input. Lines 630-660 form a *for..next* loop that reads the information from the file and displays it on the screen. Line 640 reads one line of information from the file and assigns it to variable INFO\$. Line 650 then prints the variable INFO\$ to the screen. The loop continues until all of the information has been input and displayed. Line 670 then closes the file. Line 680 resets the cursor location to position 0,0 and moves the cursor to the home position (top left corner). The variables X and Y are used to keep track of the position of the cursor as we shall see later. Line 690 returns control to the calling program.

SAVE A FILE TO DISK

If the user selects option 2, program execution is passed to the save routine which is shown below:

```
710 rem SAVE A FILE
720 show on
730 F$=file select$("*.*doc","SAVE FILE")
740 hide on
750 if F$="" then return
760 open out #1,F$
770 for Y1=0 to 19
780 INFO$=""
790 for X1=0 to 76
800 INFO=scrn(X1,Y1)
810 INFO$=INFO$+chr$(INFO)
```



```
820 next X1
830 print #1,INFO$
840 next Y1
850 close #1
860 return
```

Line 720 shows the mouse pointer and line 730 displays a file selector box so that a filename may be selected and assigned to variable F\$. Notice again that only files with ".DOC" extensions are listed. Note also, that if an existing filename is specified, the file will be overwritten by the new information. Line 740 hides the mouse pointer and line 750 checks to see that a filename has been entered. Line 760 opens the file for output. Look closely at lines 770-840 as these are a little trickier. Before the file can be saved, we have to change the information displayed on the screen to a form that can be output to the file. To do this we read the information, one character at a time, in to a variable which can be saved to the file.

Two *for...next* loops are used to track the character position across the page (X1) and the line position down the page (Y1). We know that our documents are limited to 20 lines so the variable Y1 can range from 0 to 19. The number of characters across the screen is 77 and hence variable X1 can range from 0 to 76. We would normally have 80 characters on a medium resolution screen, but two character positions are taken up by the border of the window down the left and right hand sides of the screen plus one character position at the right hand edge which is left blank for neatness. Let's take a closer look at this segment of code:

```
770 for Y1=0 to 19
780 INFO$=""

790 for X1=0 to 76
800 INFO=scrn(X1,Y1)
810 INFO$=INFO$+chr$(INFO)
```

```
820 next X1

830 print #1,INFO$

840 next Y1
```

Lines 790-820 read each character across the screen using the *scrn* command at line 800. The *scrn* command reads the character at position X1,Y1 and returns the ascii code of the character which is assigned to variable INFO. Each character read is added to the variable INFO\$ at line 810. When the loop is complete, the variable INFO\$ will contain the whole line of information. Line 830 prints this information to the disk file. Variable Y1 is then updated to indicate the next line of information, the contents of variable INFO\$ are cleared and the whole process is repeated until all 20 lines have been saved to the file.

If the user selects option 3, program execution is passed to the new document routine.

START A NEW DOCUMENT

```
880 rem NEW DOCUMENT
890 clw
900 X=0 : Y=0 : home
910 return
```

This is far simpler than the last routine and does not really require any explanation. Line 890 clears the window and line 900 resets the cursor to the home position (top left corner). Line 910 returns program execution to the calling program.

If the user selects option 4, program execution is passed to the print document routine.

PRINT DOCUMENT

```
930 rem PRINT DOCUMENT
940 for Y1=0 to 19
950 for X1=0 to 76
960 INFO=scrn(X1,Y1)
970 INFO$=INFO$+chr$(INFO)
980 next X1
990 lprint INFO$
1000 INFO$=""
1010 next Y1
1020 return
```

This works in a similar fashion to the save routine except that the information is output to the printer rather than a file on disk. Lines 950-980 form a *for..next* loop which reads each character across the screen and adds them to variable INFO\$. When the line is complete, it is output to the printer using the *lprint* command. This routine is within a second loop which repeats the process until all twenty lines have been printed.

If the user selects option 5, program execution is passed to the quit program routine.

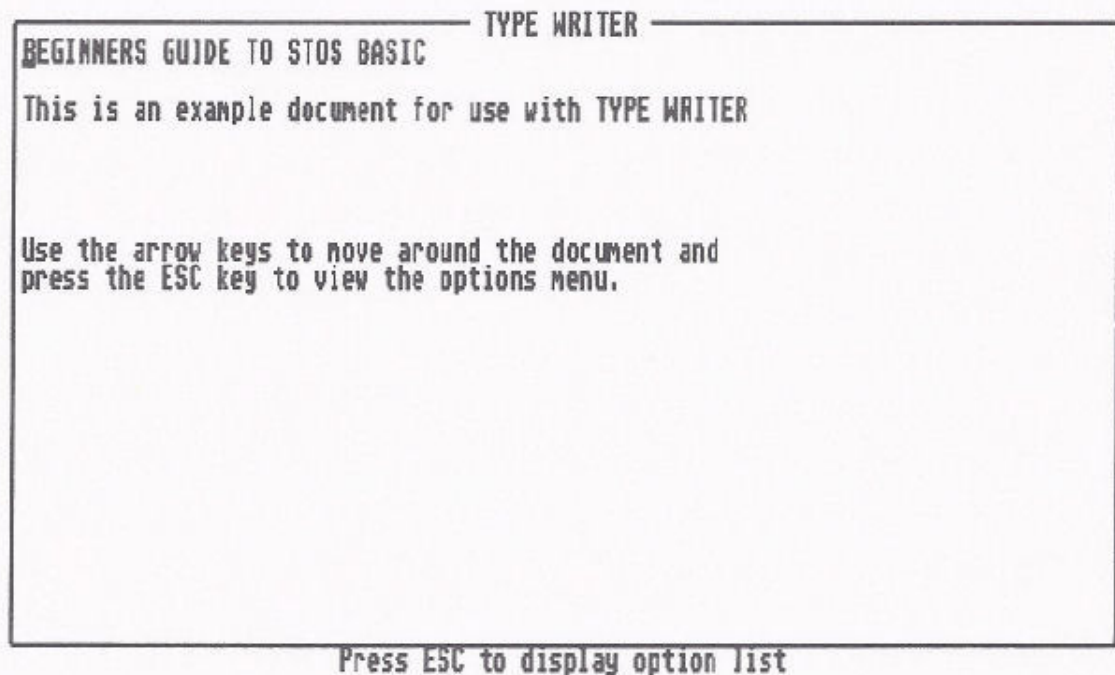
QUIT THE PROGRAM

```
1040 rem QUIT PROGRAM
1050 windel 1
1060 cls
1070 print "GOODBYE-Thank you for using Type Writer"
1080 end
```

Line 1050 clears the window from the screen and line 1060 clears the screen. The *cls* command is required to clear the message at the

bottom of the screen which was outside of the window. Line 1070 displays a good bye message and line 1080 ends the program. Note that any document that was in memory will be lost.

Well that has covered all of the menu options and all that remains is to look at the module for controlling the input and editing of documents. All documents are entered in the main window as shown in the diagram below:



In addition to accepting typed characters, we must also offer the user a facility for moving around the document and for correcting mistakes. The program must therefore accept character keys, cursor keys, the delete key, the backspace key and the ESC key with the following actions being performed.

- * Typed character will be displayed at the current cursor position.
- * The cursor may be moved around the document using the cursor (arrow) keys without upsetting any of the information already displayed.

- * [®]The return key will move the cursor to the start of the next line.
- * The delete key will erase the character at the current cursor position and move the rest of the line back by one character to fill the deleted position.
- * The backspace key will delete one character to the left of the current cursor position and move the cursor back one space to the deleted position.

```
200 rem WAIT FOR USER INPUT
210 while K$=""
220 K$=inkey$
230 wend
```

Lines 200-230 monitor the keyboard and the *while..wend* loop is continuously executed until a key is pressed.

The first area to check is the arrow keys. The arrow keys do not return an ASCII code so we have to check the scancode.

```
250 rem CHECK FOR CURSOR UP KEY
260 if asc(K$)=0 and scancode=72 and Y>0® then dec Y :
locate X,Y
```

Line 260 checks for the cursor up key which is represented by scancode 72. If the key is pressed we need to move the cursor up one line by subtracting a value of one from variable Y and re-locating the cursor at the new position. If the variable Y is equal to 0 (zero) then the cursor is already on the top line and hence is not re-located.

```
280 rem CHECK FOR CURSOR DOWN KEY
290 if asc(K$)=0 and scancode=80 and Y<19 then inc Y :
locate X,Y
```

Line 290 checks for the cursor down key which is represented by scancode 80. If the key is pressed we need to move the cursor down one line by adding one to the value of variable Y and re-locating the cursor at the new position. If the variable Y is equal to 19 then the cursor is already on the bottom line and hence is not re-located.

```
310 rem CHECK FOR CURSOR LEFT KEY
```

```
320 if asc(K$)=0 and scancode=75 and X>0 then dec X :  
locate X,Y
```

Line 320 checks for the cursor left key which is represented by scancode 75. If the key is pressed we need to move the cursor one character to the left by subtracting a value of one from variable X and re-locating the cursor at the new position. If the variable X is equal to 0 (zero) then the cursor is already at the left hand side of the screen and hence is not re-located.

```
340 rem CHECK FOR CURSOR RIGHT KEY
```

```
350 if asc(K$)=0 and scancode=77 and X<76 then inc X :  
locate X,Y
```

Line 350 checks for the cursor right key which is represented by scancode 77. If the key is pressed we need to move the cursor one character to the right by adding a value of one to variable X and re-locating the cursor at the new position. If variable X is equal to 76 then the cursor is already at the right hand edge of the screen and hence is not re-located.

The next key to check is the ESC key. When the ESC key is pressed, the pop-up menu is displayed.

```
370 rem CHECK FOR ESC KEY
```

```
380 if K$=chr$(27) then gosub 1100
```

The ESC key is represented as CHR\$(27) and line 380 compares the

pressed key with this value. If the ESC key has been pressed, program execution jumps to line 1100 where the 'pop-up' option box is displayed.

We shall now test for the RETURN key which moves the cursor to the start of the next line. This is similar to the down arrow operation except that the cursor is also moved to the left hand edge of the new line.

```
400 rem CHECK FOR RETURN KEY
410 if K$=chr$(13) and Y < > 19 then inc Y : X=0 : locate
X,Y
```

The RETURN key is represented by CHR\$(13) and line 410 compares the pressed key with this value. When the RETURN key is pressed we have to position the cursor at the left hand edge of the next line. We achieve this by adding one to the value of variable Y and setting the value of variable X to 0 (zero). If the variable Y is equal to 19 then the cursor is already on the bottom line and hence is not re-located.

The next operation is to check for the backspace and delete keys so that the user can correct any mistakes.

```
430 rem CHECK FOR BACKSPACE KEY
440 if K$=chr$(8) and X > 0 then dec X : locate X,Y : print
" "; : locate X,Y
```

The BACKSPACE key is represented by CHR\$(8) and line 440 compares the pressed key with this value. If the BACKSPACE key has been pressed we need to delete the character to the left of the current cursor position. The cursor is moved back one position and the character at this position is deleted by printing a blank space. As a consequence of the *print* command, the cursor will move to the next character position so another *locate* command is used to re-locate it back one space again. If the variable X is equal to 0 (zero), the cursor

is already at the left hand side of the screen and hence the operation cannot be carried out.

```
460 rem CHECK FOR DEL KEY
470 if asc(K$)=127 then for X1=X to 76 : locate X1,Y :
print chr$(scrn(X1+1,Y)); : next X1 : locate X,Y : goto 530
```

The DELETE key is a little trickier to follow so look carefully at the above segment of code. When the DELETE key is pressed we must move all of the characters to the right of the cursor back one position. This results in the character at the current cursor position being overwritten. The following example illustrates this:

Welcome to the Beginners Guide to STOS Basic

^

The cursor is positioned over the letter 'u' in the word 'Guide' and the DELETE key pressed. All the letters to the right of this position are moved back resulting in the following:

Welcome to the Beginners Gide to STOS Basic

The DELETE key is represented by character 127 and line 470 compares the pressed key with this value. If the key has been pressed, the variable X1 is used within a loop to represent each character position from the current cursor position through to the end of the line. Each character is read from the screen using the *scrn* command and is moved to the left by one character position. When the loop has completed, all the characters will have been moved and the cursor is re-position back to the original position.

Finally we need to check for normal characters and display them on the screen.

```
490 rem CHECK FOR NORMAL CHARACTERS
500 if asc(K$)>31 and X<76 then print K$; : inc X : goto
```


530

```
510 if asc(K$) > 31 and X=76 and Y < > 19 then print K$ :  
X=0 : inc Y
```

Normally, displayable characters are represented by ascii values above 31. We have already seen that the DELETE key is also represented by an ascii code above this value, but we have already tested for and serviced this and thus it will not interfere with this section of the program. Two separate lines are required for displaying characters. If the cursor has reached the end of the current line, the character is displayed and the cursor moved to the start of the next line. If the cursor is not at the end of the current line, the character can simply be displayed. Line 500 is executed when the cursor is away from the end of the line and line 510 is executed when the cursor is at the last character position on the line.

Line 500 prints the character at the current cursor. The cursor is automatically moved on one position as a result of the *print* command so the variable X is increased by one.

Line 510 prints the character at the current cursor position and then sets the cursor position at the start of the next line. If the variable Y is equal to 19 then the cursor is already on the bottom line and hence the operation is aborted.

We can now go back and wait for another key press, but the variable K\$ must be cleared first else the program would assume the previous input again.

```
530 K$=""  
540 goto 210
```

That brings us to the end of the type writer program.

Chapter 24

Creating a Database

This chapter continues on the theme of file handling and concentrates on the design of database programs. A database is a program that stores, retrieves and manipulates various pieces of related information. Database programs are one of the most useful applications available on computers and most people can find a good use for them. For example, you might like to maintain a database of your record collection - type in the name of a certain song and the computer will tell you instantly which record it is on. You may wish to store details of club membership, customer information, etc. One of the most common applications for a database is for storing peoples names, addresses and telephone numbers. There are many commercial programs available that tackle this requirement but there is no need to spend your hard earned cash when, as a programmer, you can write your own!

We shall now develop a database that can store names, addresses and telephone numbers. Most database programs offer a range of standard operations and we shall incorporate these in to our names and address database.

DEFINITION

We shall produce a database program that will hold a maximum of 250 records. Each of these records will contain the following information or fields:

Name
Address 1
Address 2
Address 3
Address 4
Address 5
Telephone Number
Notes

The database could be developed using either sequential or random access filing, but as the database is fairly small, all of the information will fit in to memory and thus we shall use a sequential file.

The program is to offer the following options:

FILE OPERATIONS

OPEN: Load address book information from a file previously saved on disk.

CLOSE: Save address book information to a file on disk so that it may be recalled at a later date.

RECORD VIEW OPERATIONS

FIRST: display the first record in the address book.

LAST: display the last record in the address book.

NEXT: display the next record in the address book.

PREV: display the previous record in the address book.

RECORD OPERATIONS

ADD: add a new record to the address book.

EDIT: edit an existing record in the address book.

DELETE: delete an existing record from the address book.

SEARCH: find a record by searching for user specified information.

OTHER OPERATIONS

HARDCOPY: print an address label to the printer.

QUIT: leave the address book and end the program.

DESIGN

The program shall be constructed from the following modules.

- (1) Set screen and display.
- (2) Ask user to select option.
- (3) Open file - load data from disk.
- (4) Close file - save data to disk.
- (5) Display first record.
- (6) Display last record.
- (7) Display next record.
- (8) Display previous record.
- (9) Add a new record.
- (10) Edit an existing record.
- (11) Delete an existing record.
- (12) Search for a specific record.
- (13) Print an address label.
- (14) Quit the program.

(15) Display a record.

Let's start by loading and trying the program.

Load the following:

```
10 rem ADDRESS BOOK
20 rem PROGRAM = A:\DATABASE\DATABASE
30 :
40 rem SET SCREEN
50 key off : mode 1 : flash off : hide on
60 palette $0,$777,$77,$742
70 BLACK=0 : WHITE=1 : BLUE=2 : BROWN=3
80 :
90 rem INITIALISE VARIABLES
100 TRECS=0
110 dim NAME$(250), ADR1$(250), ADR2$(250),
ADR3$(250), ADR4$(250), ADR5$(250), TEL$(250),
INFO$(250)
120 :
130 rem DISPLAY MAIN SCREEN
140 paper BLACK : pen BLUE
150 windopen 1,0,0,19,25
160 title " OPTIONS "
170 pen WHITE
180 print
190 print " O = Open"
200 print " C = Close"
210 print
220 print " F = First"
230 print " L = Last"
240 print " N = Next"
250 print " P = Prev"
260 print
270 print " A = Add"
```

```
280 print " E = Edit"
290 print " D = Delete"
300 print " S = Search"
310 print
320 print " H = Hardcopy"
330 print
340 print " Q = Quit"
350 pen BLUE
360 windopen 2,20,0,60,25
370 title " ADDRESS BOOK "
380 :
390 rem WAIT FOR USER TO SELECT OPTION
400 OPT$ = input$(1)
410 OPT$ = upper$(OPT$)
420 if OPT$ = "A" then gosub 560
430 if OPT$ = "C" then gosub 870
440 if OPT$ = "O" then gosub 1160
450 if OPT$ = "F" then gosub 1460
460 if OPT$ = "L" then gosub 1500
470 if OPT$ = "N" then gosub 1540
480 if OPT$ = "P" then gosub 1580
490 if OPT$ = "D" then gosub 1620
500 if OPT$ = "S" then gosub 1900
510 if OPT$ = "E" then gosub 2120
520 if OPT$ = "H" then gosub 2410
530 if OPT$ = "Q" then 2590
540 goto 390
550 :
560 rem ADD A NEW RECORD
570 clw
580 inc TRECS
590 pen WHITE
600 print "ADD INFORMATION TO ADDRESS BOOK"
610 print
620 input "Enter Name: ";NAME$(TRECS)
```



```
630 print
640 input "Enter Address 1: ";ADR1$(TRECS)
650 input "Enter Address 2: ";ADR2$(TRECS)
660 input "Enter Address 3: ";ADR3$(TRECS)
670 input "Enter Address 4: ";ADR4$(TRECS)
680 input "Enter Address 5: ";ADR5$(TRECS)
690 print
700 input "Enter Telephone No: ";TEL$(TRECS)
710 print
720 input "Enter Note: ";INFO$(TRECS)
730 pen BROWN
740 print
750 print "Ok to store information (Y/N)"
760 repeat
770 A$=input$(1)
780 A$=upper$(A$)
790 until A$="Y" or A$="N"
800 if A$="Y" then print "INFORMATION SAVED"
810 if A$="N" then print "OPERATION ABORTED" :
dec TRECS
820 print "Press any key to continue"
830 wait key
840 clw
850 return
860 :
870 rem SAVE DATA TO DISK
880 clw
890 pen WHITE
900 print "CLOSE ADDRESS BOOK"
910 print
920 print "Press any key to save data to disk"
930 wait key
940 print
950 print "SAVING INFORMATION"
960 open out #1,"a:\database\address.dat"
```

```
970 for CREC = 1 to TRECS
980 print #1,NAME$(CREC)
990 print #1,ADR1$(CREC)
1000 print #1,ADR2$(CREC)
1010 print #1,ADR3$(CREC)
1020 print #1,ADR4$(CREC)
1030 print #1,ADR5$(CREC)
1040 print #1,TEL$(CREC)
1050 print #1,INFO$(CREC)
1060 next CREC
1070 close #1
1080 print "FILE SAVED"
1090 CREC = 0
1100 print
1110 print "Press any key to continue"
1120 wait key
1130 clw
1140 return
1150 :
1160 rem LOAD DATA FROM DISK
1170 clw
1180 pen WHITE
1190 print "OPEN ADDRESS BOOK FILE"
1200 print
1210 print "press any key to load file"
1220 wait key
1230 print "LOADING FILE"
1240 TRECS = 0
1250 open in #1,"a:\database\address.dat"
1260 repeat
1270 inc TRECS
1280 input #1,NAME$(TRECS)
1290 input #1,ADR1$(TRECS)
1300 input #1,ADR2$(TRECS)
1310 input #1,ADR3$(TRECS)
```



```
1320 input #1,ADR4$(TRECS)
1330 input #1,ADR5$(TRECS)
1340 input #1,TEL$(TRECS)
1350 input #1,INFO$(TRECS)
1360 until eof(1)
1370 close #1
1380 print "FILE LOADED"
1390 CREC = 0
1400 print
1410 print "Press any key to continue"
1420 wait key
1430 clw
1440 return
1450 :
1460 rem DISPLAY FIRST RECORD
1470 if TRECS < > 0 then CREC = 1 : gosub 2750
1480 return
1490 :
1500 rem DISPLAY LAST RECORD
1510 if TRECS < > 0 then CREC = TRECS : gosub 2750
1520 return
1530 :
1540 rem DISPLAY NEXT RECORD
1550 if CREC < TRECS then inc CREC : gosub 2750
1560 return
1570 :
1580 rem DISPLAY PREVIOUS RECORD
1590 if CREC > 1 then dec CREC : gosub 2750
1600 return
1610 :
1620 rem DELETE A RECORD
1630 if CREC = 0 then 1880
1640 pen WHITE
1650 print
1660 print "Ok to delete the above entry (Y/N)"
```

```
1670 repeat
1680 A$ = input$(1)
1690 A$ = upper$(A$)
1700 until A$ = "Y" or A$ = "N"
1710 if A$ = "N" then print "OPERATION ABORTED"
: goto 1840
1720 dec TRECS
1730 for A = CREC to TRECS
1740 NAME$(A) = NAME$(A + 1)
1750 ADR1$(A) = ADR1$(A + 1)
1760 ADR2$(A) = ADR2$(A + 1)
1770 ADR3$(A) = ADR3$(A + 1)
1780 ADR4$(A) = ADR4$(A + 1)
1790 ADR5$(A) = ADR5$(A + 1)
1800 TEL$(A) = TEL$(A + 1)
1810 INFO$(A) = INFO$(A + 1)
1820 next A
1830 print "ENTRY DELETED"
1840 CREC = 0
1850 print "press any key to continue"
1860 wait key
1870 clw
1880 return
1890 :
1900 rem SEARCH FOR A RECORD
1910 clw
1920 pen WHITE
1930 print "SEARCH FOR A RECORD"
1940 print
1950 input "Enter name to search for: ";S$
1960 S$ = upper$(S$)
1970 print
1980 print "SEARCHING FOR: ";S$
1990 L = len(S$)
2000 for A = 1 to TRECS
```



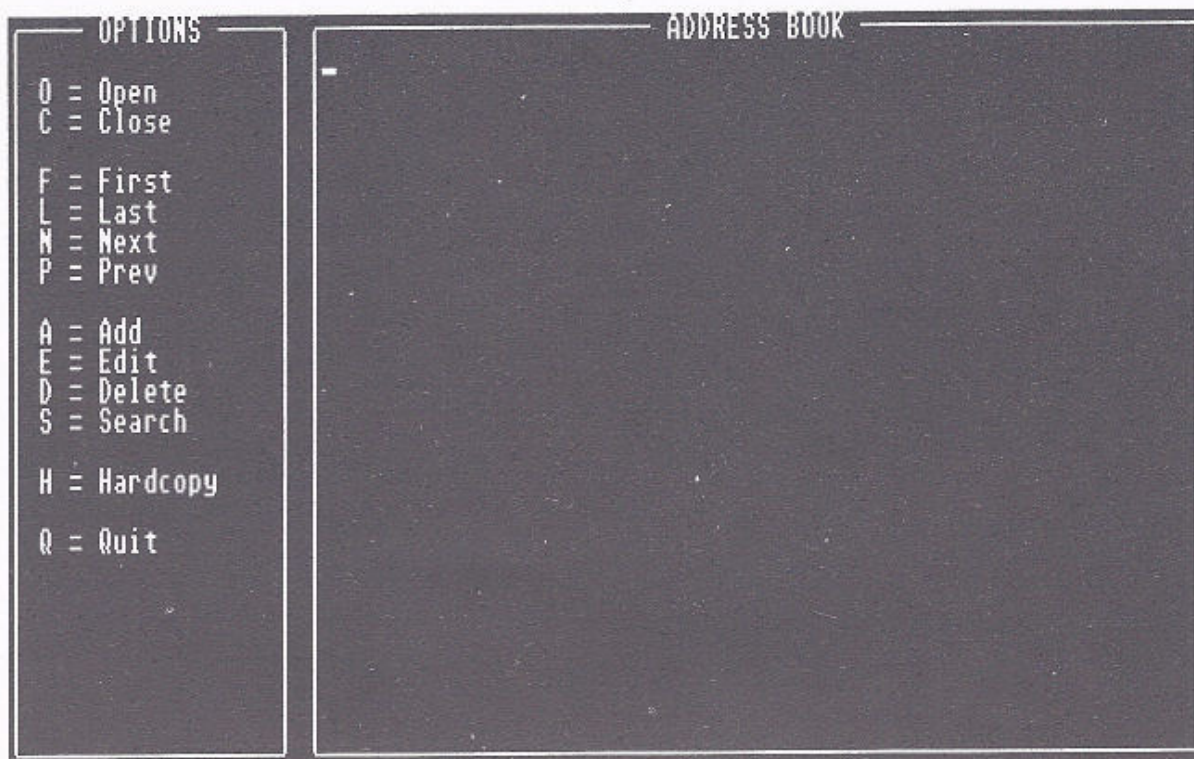
```
2010 L$ = left$(NAME$(A),L)
2020 L$ = upper$(L$)
2030 if L$ = S$ then CREC = A : gosub 2750 : goto
2100
2040 next A
2050 print "NO MATCH FOUND"
2060 CREC = 0
2070 print "Press any key to continue"
2080 wait key
2090 clw
2100 return
2110 :
2120 rem EDIT A RECORD
2130 if CREC = 0 then 2390
2140 print "NEW DETAILS"
2150 input "Name: ";TEMP_NAME$
2160 input "Address 1: ";TEMP_ADR1$
2170 input "Address 2: ";TEMP_ADR2$
2180 input "Address 3: ";TEMP_ADR3$
2190 input "Address 4: ";TEMP_ADR4$
2200 input "Address 5: ";TEMP_ADR5$
2210 input "Telephone No: ";TEMP_TEL$
2220 input "Note: ";TEMP_INFO$
2230 pen BROWN
2240 print "Ok to update the entry (Y/N)";
2250 repeat
2260 A$ = input$(1)
2270 A$ = upper$(A$)
2280 until A$ = "Y" or A$ = "N"
2290 if A$ = "N" then 2380
2300 NAME$(CREC) = TEMP_NAME$
2310 ADR1$(CREC) = TEMP_ADR1$
2320 ADR2$(CREC) = TEMP_ADR2$
2330 ADR3$(CREC) = TEMP_ADR3$
2340 ADR4$(CREC) = TEMP_ADR4$
```

```
2350 ADR5$(CREC)=TEMP_ADR5$
2360 TEL$(CREC)=TEMP_TEL$
2370 INFO$(CREC)=TEMP_INFO$
2380 gosub 2750
2390 return
2400 :
2410 rem PRINT ADDRESS LABEL
2420 if CREC=0 then 2570
2430 pen WHITE
2440 print "Ok to print address label (Y/N)"
2450 repeat
2460 A$=input$(1)
2470 A$=upper$(A$)
2480 until A$="Y" or A$="N"
2490 if A$="N" then print "LABEL NOT PRINTED" :
goto 2570
2500 lprint NAME$(CREC)
2510 lprint ADR1$(CREC)
2520 lprint ADR2$(CREC)
2530 lprint ADR3$(CREC)
2540 lprint ADR4$(CREC)
2550 lprint ADR5$(CREC)
2560 print "LABEL PRINTED"
2570 return
2580 :
2590 rem LEAVE ADDRESS BOOK AND END
2600 clw
2610 pen WHITE
2620 print "LEAVE ADDRESS BOOK"
2630 print "Remember to close address book before
leaving"
2640 print
2650 print "Do you wish to leave (Y/N)"
2660 repeat
2670 A$=input$(1)
```



```
2680 A$ = upper$(A$)
2690 until A$ = "Y" or A$ = "N"
2700 if A$ = "N" then clw : goto 390
2710 windel 2
2720 windel 1
2730 end
2740 :
2750 rem DISPLAY RECORD
2760 clw
2770 pen WHITE
2780 print "DISPLAY RECORD"
2790 pen BROWN
2800 print
2810 print "      Name: "; : pen WHITE : print
NAME$(CREC)
2820 print
2830 pen BROWN : print "      Address 1: "; : pen
WHITE : print ADR1$(CREC)
2840 pen BROWN : print "      Address 2: "; : pen
WHITE : print ADR2$(CREC)
2850 pen BROWN : print "      Address 3: "; : pen
WHITE : print ADR3$(CREC)
2860 pen BROWN : print "      Address 4: "; : pen
WHITE : print ADR4$(CREC)
2870 pen BROWN : print "      Address 5: "; : pen
WHITE : print ADR5$(CREC)
2880 print
2890 pen BROWN : print "      Telephone No: "; : pen
WHITE : print TEL$(CREC)
2900 print
2910 pen BROWN : print "      Note: "; : pen WHITE :
print INFO$(CREC)
2920 return
```

Run the program and you will see the main screen as shown below:



As you can see, the options are listed down the left hand side and are selected by pressing the appropriate letter. There is a demonstration data file supplied on the disk so let's load this.

Press the letter 'O'

Message displayed - press any key to load file

Press the RETURN key

Message displayed - file loaded, press any key to continue

Press the RETURN key

The file has now loaded but the screen is blank. You can view the records using the FIRST, LAST, NEXT and PREV options.

Press the letter F

The record for MT Software will be displayed on the screen. Now try pressing the N,P and L keys to step backwards and forwards through

the records. The database currently contains three records. Try the other options to see how each one works. (A)dd lets you add new records, (E)dit lets you edit existing records, (D)elele lets you delete an existing record and (S)earch searches for a specific record by name. Note that you only have to enter the first part of the name. For example, "MT" should be enough to find "MT Software". Option (H)ardcopy prints an address label and (Q)uit ends the program. Before leaving the program, be sure to close the file using the "C" option. This writes the file back to disk.

Let's look at the program listing.

SET SCREEN AND DISPLAY

```
40 rem SET SCREEN
50 key off : mode 1 : flash off : hide on
60 palette $0,$777,$77,$742
70 BLACK=0 : WHITE=1 : BLUE=2 : BROWN=3
```

Like all programs, we start by setting the required screen resolution and a suitable colour palette. In this case we shall use medium resolution.

```
90 rem INITIALISE VARIABLES
100 TRECS=0
110 dim NAME$(250), ADR1$(250), ADR2$(250),
ADR3$(250), ADR4$(250), ADR5$(250), TEL$(250), INFO$(250)
```

Line 100 assigns the value 0 (zero) to variable TRECS. TRECS maintains the total number of records that are held in the address book. When the program first starts there are no records in memory and thus this is set to zero. Line 110 dimensions variable arrays which will be used for storing the various information. For this example we have limited the number of records to 250 but this can be increased if you wish to increase the capacity of the database.

```
130 rem DISPLAY MAIN SCREEN
140 paper BLACK : pen BLUE
150 windopen 1,0,0,19,25
160 title " OPTIONS "
170 pen WHITE
180 print
190 print " O = Open"
200 print " C = Close"
210 print
220 print " F = First"
230 print " L = Last"
240 print " N = Next"
250 print " P = Prev"
260 print
270 print " A = Add"
280 print " E = Edit"
290 print " D = Delete"
300 print " G = Get"
310 print
320 print " H = Hardcopy"
330 print
340 print " Q = Quit"
350 pen BLUE
360 windopen 2,20,0,60,25
370 title " ADDRESS BOOK "
```

The main screen consists of two windows as shown in the previous picture. Lines 150-340 open window 1 on the left hand side of the screen and display the list options of in it. Lines 350-370 open a second, larger, window which is used for displaying the address book records.

ASK USER TO SELECT OPTION

```
390 rem WAIT FOR USER TO SELECT OPTION
400 OPT$=input$(1)
410 OPT$=upper$(OPT$)
420 if OPT$="A" then gosub 560
430 if OPT$="C" then gosub 870
440 if OPT$="O" then gosub 1160
450 if OPT$="F" then gosub 1460
460 if OPT$="L" then gosub 1500
470 if OPT$="N" then gosub 1540
480 if OPT$="P" then gosub 1580
490 if OPT$="D" then gosub 1620
500 if OPT$="G" then gosub 1900
510 if OPT$="E" then gosub 2120
520 if OPT$="H" then gosub 2410
530 if OPT$="Q" then 2590
540 goto 390
```

Lines 400-540 wait for the user to enter an option and then pass program execution to the appropriate subroutine. Line 400 waits for a key press and assigns this to variable OPT\$. Line 410 converts the contents of variable OPT\$ to upper case. This is necessary to ensure that the comparison routines work even if the user enters a lower case character. Lines 420- 530 check the entered character against the valid options and call the appropriate subroutine. If an invalid option is entered, line 540 passes control back to line 390 where the user gets another chance to enter a valid option.

We shall now look at each of the subroutines in the same order that they appear on the option list

OPEN FILE - LOAD DATA FROM DISK

```
1160 rem LOAD DATA FROM DISK
1170 clw
1180 pen WHITE
1190 print "OPEN ADDRESS BOOK FILE"
1200 print
1210 print "press any key to load file"
1220 wait key
1230 print "LOADING FILE"
1240 TRECS=0
1250 open in #1,"a:\database\address.dat"
1260 repeat
1270 inc TRECS
1280 input #1,NAME$(TRECS)
1290 input #1,ADR1$(TRECS)
1300 input #1,ADR2$(TRECS)
1310 input #1,ADR3$(TRECS)
1320 input #1,ADR4$(TRECS)
1330 input #1,ADR5$(TRECS)
1340 input #1,TEL$(TRECS)
1350 input #1,INFO$(TRECS)
1360 until eof(1)
1370 close #1
1380 print "FILE LOADED"
1390 CREC=0
1400 print
1410 print "Press any key to continue"
1420 wait key
1430 clw
1440 return
```

Line 1170 clears the window, line 1180 sets the pen colour to white and line 1190 prints a title message. Lines 1210-1220 prompt the user to press a key to load the file. This gives the user time to place the

data disk in the drive if necessary. Line 1240 assigns the value 0 (zero) to the variable TRECS (total records). Line 1250 opens the file "address.dat" on channel #1 for input. Lines 1260-1360 form a *repeat..until* loop which reads in the data. Line 1270 increments the number of total records as each new record is read in. Lines 1280-1350 read the data in to each variable. Notice how the variable TRECS is used to indicate the number of the variable within the array. Line 1360 checks to see if there is any more data left in the file. If there is more data, the loop is repeated again else the program moves on to line 1370 where the file is closed and line 1380 confirms that loading is complete . Line 1390 sets the variable CREC (current record) to 0. While TRECS maintains the total number of records, CREC maintains the record currently displayed or active. As no record has actually been displayed yet, we set the current record to 0 (zero). Lines 1410-1420 wait for the user to press a key, line 1430 clears the window and line 1440 returns control to the calling program.

CLOSE FILE - SAVE DATA TO DISK

```
870 rem SAVE DATA TO DISK
880 clw
890 pen WHITE
900 print "CLOSE ADDRESS BOOK"
910 print
920 print "Press any key to save data to disk"
930 wait key
940 print
950 print "SAVING INFORMATION"
960 open out #1,"a:\database\address.dat"
970 for CREC=1 to TRECS
980 print #1,NAME$(CREC)
990 print #1,ADR1$(CREC)
1000 print #1,ADR2$(CREC)
```

```
1010 print #1,ADR3$(CREC)
1020 print #1,ADR4$(CREC)
1030 print #1,ADR5$(CREC)
1040 print #1,TEL$(CREC)
1050 print #1,INFO$(CREC)
1060 next CREC
1070 close #1
1080 print "FILE SAVED"
1090 CREC=0
1100 print
1110 print "Press any key to continue"
1120 wait key
1130 clw
1140 return
```

The CLOSE routine is very similar to the OPEN routine except that we are writing information to a file rather than reading information from a file. Lines 880-900 clear the window, set the pen colour to white and print a title message. Lines 920-930 wait for the user to press a key and allow time for the data disk to be prepared. Line 960 opens the file "address.dat" for output on channel #1. Lines 970-1060 form a *for..next* loop which prints each element of the arrays to the file. Variable CREC (current record) is used as the loop counter and represents the values 1 to TRECS. When all of the records have been written to disk, line 1070 closes the file. Line 1090 clears variable CREC as no record is currently displayed. The user is asked to press any key, the window is cleared and control is passed back to the calling program.

We can now look at the interesting parts, the actual control of the data itself.

DISPLAY FIRST RECORD

When the user enters option F(first), the program should display the first record on the screen. The first thing to consider is whether any records actually exist and we can easily find out by checking the total records pointer (TRECS). If TRECS=0 there are no records available and hence the first record cannot be displayed. If TRECS < > 0 then we can go ahead and display the first record. To do this we simply set the current record pointer to number one (CREC=1) and display the record. The code for this is shown below:

```
1460 rem DISPLAY FIRST RECORD
1470 if TRECS < > 0 then CREC=1 : gosub 2750
1480 return
```

Line 1470 compares variable TRECS against the value 0 (zero) to check that there are actually some records in memory. If TRECS is not equal to zero then there are some records available and the current record (CREC) is set to the first record (1). A subroutine at line 2750 is then called to display the record on the screen. We shall look at the display routine later.

DISPLAY LAST RECORD

The next option is L(ast) and this displays the last record. Once again we must check that there are some records in memory by interrogating variable TRECS (total records). If records exist we can go ahead and display the last record. To do this we have to set the current record pointer to the last record. The variable TRECS indicates the total number of records so if we make CREC equal to TRECS it will indicate the last record. The code for this is shown below:

```
1500 rem DISPLAY LAST RECORD
1510 if TRECS < > 0 then CREC=TRECS : gosub 2750
```

1520 return

Line 1510 checks for records and sets the current record pointer (CREC) to the last record (indicated by TRECS). The display routine is then called.

DISPLAY THE NEXT RECORD

The next option is N(ext). This displays the next record on the screen. We need to check that there are more records to come and that the last record has not already been reached. We can do this by comparing the current record pointer (CREC) with the total number of records (TRECS), and if CREC is smaller than TRECS, there are more records to come. The value of CREC (current record) can be increased by one and the record displayed.

```
1540 rem DISPLAY NEXT RECORD
1550 if CREC < TRECS then inc CREC : gosub 2750
1560 return
```

Line 1550 checks that the current record is lower than the total number of records, and if so, increments the current record. The record is then displayed.

DISPLAY PREVIOUS RECORD

The final record viewing option is P(revious) which displays the previous record on the screen. We need to check that there are previous records to display and that the current record is not already the first record. If CREC (current record pointer) is greater than one then there are previous records available. The current record pointer (CREC) can be reduced by one and the record displayed. The code is shown over the page.


```
1580 rem DISPLAY PREVIOUS ENTRY
1590 if CREC > 1 then dec CREC : gosub 2750
1600 return
```

Line 1590 checks that CREC is greater than one, and if so, decrements it. The record is then displayed.

ADDING NEW RECORDS

The A(dd) option allows new records to be added to the address book. The code for this is shown below:

```
560 rem ADD A NEW RECORD
570 clw
580 inc TRECS
590 pen WHITE
600 print "ADD INFORMATION TO ADDRESS BOOK"
610 print
620 input "Enter Name: ";NAME$(TRECS)
630 print
640 input "Enter Address 1: ";ADR1$(TRECS)
650 input "Enter Address 2: ";ADR2$(TRECS)
660 input "Enter Address 3: ";ADR3$(TRECS)
670 input "Enter Address 4: ";ADR4$(TRECS)
680 input "Enter Address 5: ";ADR5$(TRECS)
690 print
700 input "Enter Telephone No: ";TEL$(TRECS)
710 print
720 input "Enter Note: ";INFO$(TRECS)
730 pen BROWN
740 print
750 print "Ok to store information (Y/N)"
760 repeat
770 A$=input$(1)
```

```
780 A$=upper$(A$)
790 until A$="Y" or A$="N"
800 if A$="Y" then print "INFORMATION SAVED"
810 if A$="N" then print "OPERATION ABORTED" : dec
TRECS
820 print "Press any key to continue"
830 wait key
840 clw
850 return
```

Whenever a new record is added, we need to increment the total number of records pointer to take account of the new record. Line 580 therefore increments variable TRECS. Lines 590-720 display a title message and request the user to enter the various information. Notice the use of TRECS with the other variables to indicate the element of the arrays. Before permanently adding the information to the database, it is always a good idea to give the user a chance to review the entered information as he or she may wish to cancel the operation. This confirmation process is performed by lines 750-790. Line 770 waits for the user to press a key and line 780 converts the entered character to upper case. If the character 'Y' (yes) is entered, no further action is required as the total number of records has already been updated and the new information already accepted. If the character 'N' (no) is entered then the total number of records pointer is decreased by one. It is important to note that the information previously entered is still assigned to the variables, but only those records up to and including TRECS form part of the database and hence the new record is effectively deleted. If the user does not enter 'Y' or 'N' they are returned back to try again. Lines 820-830 wait for the user to press a key, line 840 clears the window and line 850 returns control to the calling program.

EDIT EXISTING RECORD

The E(dit) option allows existing records to be modified. The user may need to update the details because a person has changed address or may simply wish to correct a previous spelling mistake. The existing record could be deleted and a new one added, but it is far easier for the user and more professional for the programmer to provide an EDIT option. The code for this is shown below:

```
2120 rem EDIT EXISTING ENTRY
2130 if CREC=0 then 2390
2140 print "NEW DETAILS"
2150 input "Name: ";TEMP_NAME$
2160 input "Address 1: ";TEMP_ADR1$
2170 input "Address 2: ";TEMP_ADR2$
2180 input "Address 3: ";TEMP_ADR3$
2190 input "Address 4: ";TEMP_ADR4$
2200 input "Address 5: ";TEMP_ADR5$
2210 input "Telephone No: ";TEMP_TEL$
2220 input "Note: ";TEMP_INFO$
2230 pen BROWN
2240 print "Ok to update the entry (Y/N)";
2250 repeat
2260 A$=input$(1)
2270 A$=upper$(A$)
2280 until A$="Y" or A$="N"
2290 if A$="N" then 2380
2300 NAME$(CREC)=TEMP_NAME$
2310 ADR1$(CREC)=TEMP_ADR1$
2320 ADR2$(CREC)=TEMP_ADR2$
2330 ADR3$(CREC)=TEMP_ADR3$
2340 ADR4$(CREC)=TEMP_ADR4$
2350 ADR5$(CREC)=TEMP_ADR5$
2360 TEL$(CREC)=TEMP_TEL$
2370 INFO$(CREC)=TEMP_INFO$
```



```

2380 gosub 2750
2390 return

```

A record can only be edited if it is already displayed on the screen (the current record) so line 2130 checks to see if there is a record currently displayed on the screen. If there is not a current record (CREC=0), program execution is passed to line 2390 where the subroutine ends with no further action. Lines 2140-2220 input the updated or new information which is assigned to temporary variables. The existing record information remains on the screen as shown below:

OPTIONS	ADDRESS BOOK
O = Open	DISPLAY RECORD
C = Close	Name: MT SOFTWARE
F = First	Address 1: Greensward House
L = Last	Address 2: The Broadway
N = Next	Address 3: Totland
P = Prev	Address 4: Isle of Wight
	Address 5: PO39 0BX
A = Add	Telephone No: 0983 756056
E = Edit	Note: Atari Software
D = Delete	NEW DETAILS
S = Sort	Name: MTS(UK)
G = Get	Address 1: Greensward House
H = Hardcopy	Address 2: The Broadway
Q = Quit	Address 3: Totland
	Address 4: IOW
	Address 5: PO39 0BX
	Telephone No: 0983 756056
	Note: Software for Atari ST/STE
	Ok to update the entry (Y/N)_

The existing information is maintained until the user has entered the updated information and confirmed that this new information is to replace the existing information. The new information is therefore input to temporary variables (temp_name\$, etc.) so that the original variables may be maintained. Lines 2240-2280 check that the user does still want to carry out the update operation. If the user enters 'N' (No), program execution jumps to line 2380 where the display routine

is called to re-display the original record details. No further action is required as the original information is still stored within the database. If the user enters 'Y' (Yes), we have to change the existing information with the new entered information. This is performed by lines 2300-2370 which copy the temporary variable information to the main variables.

We use temporary variables rather than loading directly in to the main database variables because the user may wish to change their mind about editing the data, and hence the main database variables are maintained with the old contents until the user confirms that the information is to be updated. The display routine is then called at line 2380 to display the new record information and the subroutine ends at line 2390.

DELETE AN EXISTING RECORD

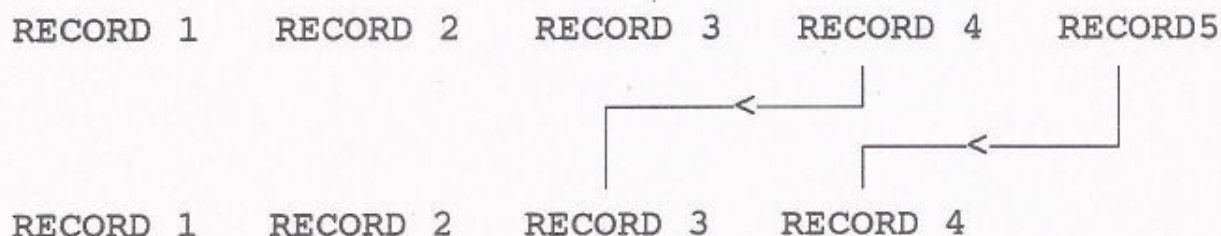
The D(elete) option allows a record to be completely deleted from the database and the code for this is shown below:

```
1620 rem DELETE AN ENTRY
1630 if CREC=0 then 1880
1640 pen WHITE
1650 print
1660 print "Ok to delete the above entry (Y/N)"
1670 repeat
1680 A$=input$(1)
1690 A$=upper$(A$)
1700 until A$="Y" or A$="N"
1710 if A$="N" then print "OPERATION ABORTED" :
goto 1840
1720 dec TRECS
1730 for A=CREC to TRECS
1740 NAME$(A)=NAME$(A+1)
```

```
1750 ADR1$(A)=ADR1$(A+1)
1760 ADR2$(A)=ADR2$(A+1)
1770 ADR3$(A)=ADR3$(A+1)
1780 ADR4$(A)=ADR4$(A+1)
1790 ADR5$(A)=ADR5$(A+1)
1800 TEL$(A)=TEL$(A+1)
1810 INFO$(A)=INFO$(A+1)
1820 next A
1830 print "ENTRY DELETED"
1840 CREC=0
1850 print "press any key to continue"
1860 wait key
1870 clw
1880 return
```

A record can only be deleted if it is already displayed on the screen (the current record) so line 1630 checks to see if there is a record currently displayed on the screen. If there is not a current record (CREC=0), program execution is passed to line 1880 where the subroutine ends with no further action. Lines 1660-1710 ask the user to confirm that they want to delete the record currently displayed on the screen. If the user enters 'N' (No) an "OPERATION ABORTED" message is displayed and program execution passed to the end of the subroutine. If the user enters 'Y' (Yes) we have to delete the current record. If a record is deleted, the total number of records will be one less so variable TRECS is decreased by one at line 1720. To delete the record we shuffle all the proceeding records back one as shown on the next page.

Suppose we want to delete record 3, records 4 and 5 are moved back over writing and therefore removing record 3.

BEFORE

This shuffling process is carried out by lines 1730-1820 which form a *for..next* loop. If we consider the above example where there are 5 records, the loop will start at record 3 ($A=3$) and continue through to the total number of records ($TRECS=5$). To illustrate the process let's consider the first run through the loop where variable $A=3$.

```
1740 NAME$(A)=NAME$(A+1)
```

this line equates to:

```
NAME$(3)=NAME$(4)
```

The contents of record 4 are copied in to record 3 thereby deleting the contents of record 3. This is completed for the address, telephone number and notes before moving on. The loop continues until all of the records have been successfully moved. Line 1840 sets the current record (CREC) to 0 (zero) as the original record has been deleted and should no longer be displayed on the screen. Lines 1850-1860 wait for the user to press a key, line 1870 clears the window and line 1880 passes control back to the calling program.

SEARCHING FOR A RECORD

The S(earch) option searches the database for a specified name and displays the appropriate record if a match is found. The code for this is shown on the opposite page.

```
1900 rem SEARCH ADDRESS BOOK
1910 clw
1920 pen WHITE
1930 print "SEARCH FOR A RECORD"
1940 print
1950 input "Enter name to search for: ";S$
1960 S$=upper$(S$)
1970 print
1980 print "SEARCHING FOR: ";S$
1990 L=len(S$)
2000 for A=1 to TRECS
2010 L$=left$(NAME$(A),L)
2020 L$=upper$(L$)
2030 if L$=S$ then CREC=A : gosub 2750 : goto 2100
2040 next A
2050 print "NO MATCH FOUND"
2060 CREC=0
2070 print "Press any key to continue"
2080 wait key
2090 clw
2100 return
```

When searching for a name, the user only needs to enter the first few letters and not the whole name. The search routine searches the first part of each name to see if a match can be found. Note that the routine is not case conscious and will still find the information regardless of whether it is upper or lower case.

Line 1950 waits for the user to enter a name and assigns this to variable S\$. Line 1960 converts the variable S\$ to uppercase. Line 1980 displays a message to let the user know that searching is about to take place. Line 1990 calculates the length of variable S\$ (the number of characters it contains) and assigns this value to variable L. To check for a match we have to extract the first 'L' number of characters from the left hand side of each name and compare this with

S\$. We want to include all records in the search so we start by specifying a *for..next* loop that ranges from 1 through to the total number of records (TRECS). To see how the process works let's follow the first run through the loop where the loop counter, variable A, equals 1. Consider that the user wants to search for the name John, therefore S\$ = "JOHN":

```
2010 L$=left$(NAME$(A),L)
```

this equates to L\$=left\$(NAME\$(1),4)

[variable L = 4 as "JOHN" consists of 4 characters]

variable L\$ will contain the first four characters of variable NAME\$(1). Line 2020 converts variable L\$ to upper case. Now that L\$ contains the same number of characters as S\$, we can make the comparison to see if they are the same. This is carried out by line 2030. If they are the same, a match has been found and the record can be displayed. The current record (CREC) is set to equal that of the found record and the subroutine is called to display the record on the screen. When the subroutine is complete, control is passed back and the routine ends.

If no match is found against any of the records, the program displays an appropriate message, sets the current record (CREC) to zero and ends the subroutine.

PRINT AN ADDRESS LABEL

The (H)ardcopy option allows a copy of the name and address to be sent to the printer. This is ideal for printing sticky address labels. The code is shown below:

```
2410 rem PRINT ADDRESS LABEL
```

```
2420 if CREC=0 then 2570
2430 pen WHITE
2440 print "Ok to print address label (Y/N)"
2450 repeat
2460 A$=input$(1)
2470 A$=upper$(A$)
2480 until A$="Y" or A$="N"
2490 if A$="N" then print "LABEL NOT PRINTED" : goto
2570
2500 lprint NAME$(CREC)
2510 lprint ADR1$(CREC)
2520 lprint ADR2$(CREC)
2530 lprint ADR3$(CREC)
2540 lprint ADR4$(CREC)
2550 lprint ADR5$(CREC)
2560 print "LABEL PRINTED"
2570 return
```

Line 2420 checks that a record is displayed on the screen, and if not, ends the subroutine. Lines 2430-2480 ask the user to confirm that they wish to print a label. If the user enters "N" (No), the subroutine ends else the program passes on to line 2500 where the name and address information is sent to the printer.

Remember that the *lprint* command is exactly the same as the *print* command except that the information is directed to the printer rather than the screen.

QUIT THE PROGRAM

The Q(uit) option ends the program and any data will be lost unless previously saved using the C(lose) option. The code for this is shown on the opposite page.


```
2590 rem LEAVE ADDRESS BOOK AND END
2600 clw
2610 pen WHITE
2620 print "LEAVE ADDRESS BOOK"
2630 print "Remember to close address book before leaving"
2640 print
2650 print "Do you wish to leave (Y/N)"
2660 repeat
2670 A$=input$(1)
2680 A$=upper$(A$)
2690 until A$="Y" or A$="N"
2700 if A$="N" then clw : goto 390
2710 windel 2
2720 windel 1
2730 end
```

Line 2600 clears the window and lines 2620-2690 ask the user to confirm their wish to end the program. If the user enters "N" (No) then the program returns back to the main screen else the windows are deleted and the program ends.

DISPLAY A RECORD

The final subroutine is that which displays a record on the screen. Many of the options call this subroutine which is reproduced below:

```
2750 rem DISPLAY RECORD
2760 clw
2770 pen WHITE
2780 print "DISPLAY RECORD"
2790 pen BROWN
2800 print
2810 print "      Name: "; : pen WHITE : print
NAME$(CREC)
```

```
2820 print
2830 pen BROWN : print "    Address 1: "; : pen WHITE :
print ADR1$(CREC)
2840 pen BROWN : print "    Address 2: "; : pen WHITE :
print ADR2$(CREC)
2850 pen BROWN : print "    Address 3: "; : pen WHITE :
print ADR3$(CREC)
2860 pen BROWN : print "    Address 4: "; : pen WHITE :
print ADR4$(CREC)
2870 pen BROWN : print "    Address 5: "; : pen WHITE :
print ADR5$(CREC)
2880 print
2890 pen BROWN : print "    Telephone No: "; : pen
WHITE : print TEL$(CREC)
2900 print
2910 pen BROWN : print "    Note: "; : pen WHITE : print
INFO$(CREC)
2920 return
```

This routine requires no explanation. *Print* commands are used to print the various information as shown over the page.

OPTIONS	ADDRESS BOOK
O = Open	DISPLAY RECORD
C = Close	Name: MT Software
F = First	Address 1: Greensward House
L = Last	Address 2: The Broadway
N = Next	Address 3: Totland
P = Prev	Address 4: Isle of Wight
A = Add	Address 5: PO39 0BX
E = Edit	Telephone No: 0983 756056
D = Delete	Note: Publisher of Computer Software
S = Search	-
H = Hardcopy	
Q = Quit	

Chapter 25

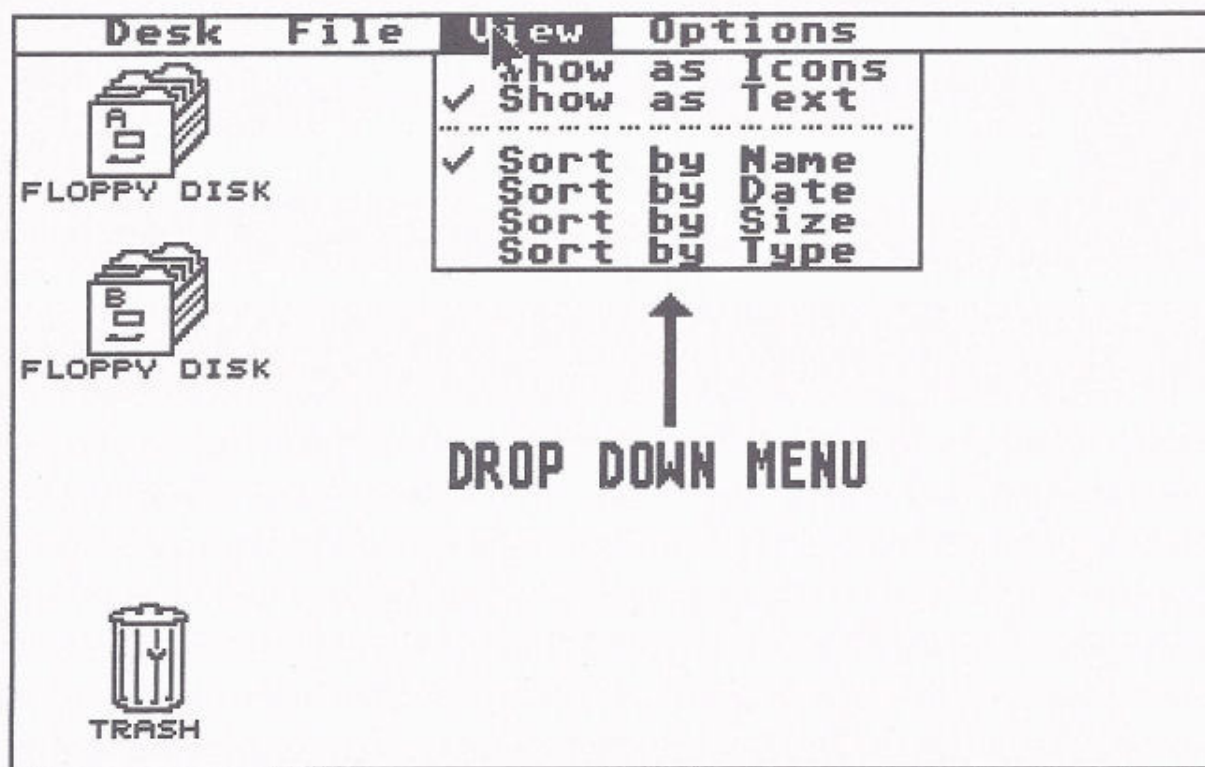
Menus, Dialogue & Information Boxes

We have already covered a number of methods for obtaining input from the user but quite often the programmer may require the user to select one option from a list. The easiest method is to simply list the options, each with its own number that can be entered to select the appropriate option. This is a straightforward, user friendly method which carries out the function quite well, but our programs can be made to look much more professional using alternative forms of menus.

In this chapter we shall look at the facilities offered by STOS for the creation and maintenance of drop down and pull down menus and shall also develop some programs for producing slightly more advanced menu systems. There are many methods that can be used to produce a menu and the following ideas and programs serve only to illustrate some the effects that can be achieved with relatively simple programming.

DROP DOWN & PULL DOWN MENUS

The Atari ST is hailed as one of the most user friendly computers and this is probably due to GEM (Graphical Environment Management). A major feature of GEM is the drop down menus that drop down from the top of the screen and allow various options to be selected using the mouse. You have already come across these on the main desktop which is displayed whenever the computer is switched on.



STOS offers a range of commands that allow us to incorporate these menus within our programs. Apart from providing a convenient method of communicating with the user, they make a program look very professional indeed. STOS allows us to generate menus that are a lot more powerful than even those offered by GEM and the good news is that they are incredibly easy to program. There are two types of menu available:-

DROP DOWN MENUS: drop down menus 'drop down', or are

displayed as soon as the mouse pointer passes over the menu heading. Drop Down menus are preferred by many programmers as they are the same as those offered by GEM and hence offer a degree of compatibility. This being the case, there are times when these are not convenient. For example, if using an art program the user may wish to draw on the screen using the mouse. If the mouse pointer moves too close to the menu bar it could trigger a menu to drop down when it is not really required. STOS therefore offers a second type of menu system.

PULL DOWN MENUS: pull down menus are very similar to the drop down variety, but when the mouse pointer moves on to the menu heading, the menu does not drop down until the left mouse button is pressed. This system is used on the Apple Mac range of computers and is preferred by some users. Different programmers may have their own preference as to which type they prefer, but you could be really flash and give the user the chance to select which type he/she would like; it is of course the user who will be using the menus.

We shall now see how to program these menus.

DEFINING THE MENU STRUCTURE

The first stage of programming drop down and pull down menus is to design the menu structure. This involves listing the menu bar headings and the various options for each menu.

We shall design a very simple program to illustrate how menus are set up and managed. The program will allow the user to change the style of the mouse pointer by selecting the required style from a menu.

There shall be two menus as shown over the page.

MENU BAR TITLE ---->	MOUSE	QUIT
MENU ENTRY ----->	Arrow	Quit
	Pointing Hand	
	Clock Face	

Load the following:

```
10 rem DROP DOWN MENUS - CHANGE MOUSE
20 rem PROGRAM = A:\MENUS\MOUSEMEN
30 :
40 key off : mode 0
50 :
60 rem DEFINE MENU STRUCTURE
70 menu$ (1) = " MOUSE "
80 menu$ (1,1) = " Arrow"
90 menu$ (1,2) = " Pointing Hand "
100 menu$ (1,3) = " Clock Face"
110 menu$ (2) = " QUIT "
120 menu$ (2,1) = " Quit "
130 :
140 rem TURN ON MENU
150 menu on 5,1
160 curs off
170 :
180 rem MONITOR MENU AND CHANGE MOUSE
190 MB = mnbar : MS = mnselect
200 if MB = 1 and MS < > 0 then change mouse MS
210 if MB = 2 and MS = 1 then menu off : end
220 :
230 goto 190
```

Run the program. Move the mouse pointer up to the menu bar and select MOUSE. When the menu drops down, select the required mouse pointer style and the pointer will change. Use the QUIT menu to end the program.

Let us look at the program.

Lines 60-120 define the structure of the menu. The *menu\$* command is used to define both the headings and the individual entries for each menu.

```
70 menu$(1)=" MOUSE "  
80 menu$ (1,1)=" Arrow "  
90 menu$ (1,2)=" Pointing Hand "  
100 menu$ (1,3)=" Clock Face"
```

Notice how the entries are listed after the main heading. STOS allows a maximum of 10 menus to be defined.

Now that the menu has been defined, it can be displayed on the screen and line 150 carries out this operation using the *menu on* command. The format of the *menu on* command is shown below:

```
menu on BORDER, TYPE
```

where BORDER indicates the style of border and TYPE indicates the type of menu. The border styles are the same as those for windows and TYPE indicates whether the menu should be 'drop down' or 'pull down'. A value of 1 indicates a drop down menu and a value of 2 indicates a pull down menu. The menu in our program will thus have border style 5 and will be of the drop down variety.

Line 160 removes the text cursor from the screen as it is not currently required. When a menu is switched on, STOS restores the text cursor and colour flashing even if they have previously been switched off. I am not sure whether this is intentional or just a bug, but the problem is easily solved by switching off the cursor or colour flashing after the *menu on* command.

Once a menu has been switched on we can check to see if the user

selects an option. STOS reserves two variables MNBAR and MNSELECT that are used for checking the menus. MNBAR contains the number of the selected menu heading and MNSELECT contains the value of the selected menu item. If no menus have been selected, the values of MNBAR and MNSELECT will be zero. So by checking these two variables we can establish the option that has been selected.

Referring back to the program, line 190 assigns the value of MNBAR to variable MB and the value of MNSELECT to variable MS.

Lines 200 and 210 check the values of variables MB and MS and carry out the appropriate operation. Line 200 checks for menu one (MOUSE). If $MN=1$ and $MS < > 0$ then the mouse pointer is changed. The check $MS < > 0$ ensures that a menu item has actually been selected because the user could have selected the menu heading without selecting one of the entries. Line 210 ends the program if the "quit" option is selected. Notice the *menu off* command that switches off the menu and clears it from the screen.

Line 230 sends program execution back to line 190 to check the status of the menus again.

We could make the menus 'pull down' by changing line 150.

Enter the following:

150 menu on 5,2

Run the program again, and when the mouse pointer is on the menu bar, press a mouse button to make the menu come down.

The last example was fairly simple and to illustrate the full extent of the menu facilities we shall now look at a slightly larger example. Consider a drop down menu structure that might be used for a database program.

MENU BAR TITLE	FILE	RECORD	SORT
MENU ENTRY	Open	First	Sort A-Z
	Close	Last	Sort Z-A
	-----	-----	
	Quit	Next	
		Prev	

		Add	

The file menu allows data files to be opened and closed and also allows the user to leave the program. The record menu would allow the user to jump to the first record, the last record, the next record and the previous record and the sort menu would allow the records to be sorted in ascending or descending order.

The program serves only to illustrate the menu commands and will not actually perform the described tasks.

Load the following:

```

10 rem MENU STRUCTURES
20 rem PROGRAM = A:\MENUS\DATABASE
30 :
40 key off : mode 0
50 palette $0,$777,,$700,$70,$7
60 :
70 rem DEFINE MENU STRUCTURE
80 menu$ (1) = " FILE ",3,1
90 menu$ (1,1) = " Open ",0,3
100 menu$ (1,2) = " Close ",0,3
110 menu$ (1,3) = " ----- "
120 menu$ (1,4) = " Quit ",0,1
130 menu$ (1,3) off
140 :
150 menu$ (2) = " RECORDS ",4,1

```



```
160 menu$ (2,1) = " First ",0,4
170 menu$ (2,2) = " Last ",0,4
180 menu$ (2,3) = " ----- "
190 menu$ (2,4) = " Next ",0,1
200 menu$ (2,5) = " Prev ",0,1
210 menu$ (2,6) = " ----- "
220 menu$ (2,7) = " Add ",0,4
230 menu$ (2,3) off
240 menu$ (2,6) off
250 :
260 menu$ (3) = " SORT ",5,1
270 menu$ (3,1) = " Sort A-Z ",0,5
280 menu$ (3,2) = " Sort Z-A ",0,1
290 :
300 rem DISPLAY MENU
310 menu on 3,1
320 :
330 rem MONITOR THE MENU
340 on menu goto 1000,2000,3000
350 on menu on
360 goto 360
370 :
380 :
1000 rem FILE OPERATIONS
1010 menu freeze
1020 M = mnselect
1030 if M = 1 then print "File Open"
1040 if M = 2 then print "File Close"
1050 if M = 4 then print "Quit"
1060 goto 300
1070 :
2000 rem RECORDS OPERATIONS
2010 menu freeze
2020 M = mnselect
2030 if M = 1 then print "First Record"
```

```
2040 if M = 2 then print "Last Record"
2050 if M = 4 then print "Next Record"
2060 if M = 5 then print "Previous Record"
2070 if M = 7 then print "Add record"
2080 goto 300
2090 :
3000 rem SORT OPERATIONS
3010 menu freeze
3020 M = mnselect
3030 if M = 1 then print "Sort A-Z"
3040 if M = 2 then print "Sort Z-A"
3050 goto 300
```

Run the program.

The first thing you will notice is that each of the three menus is displayed in a different colour, and when the menu drops down, the actual options are also displayed in different colours.

Lines 80-280 define the menu structure. Notice the numbers at the end of the *menu\$* commands which indicate colour information. The first number indicates the paper colour and the second number indicates the pen colour. If these are omitted, the menus are produced using the current pen and paper settings as per the previous program.

Lines 130, 230 and 240 introduce the *menu\$()* *off* command. Related options can be separated within each menu using a line. The only problem is that STOS would consider this line as a normal menu entry, and thus to stop the user selecting it, we must switch it off. If you look at the menu you will notice, as you move the mouse down the list, that the lines cannot be selected. Menu options can be re-activated with the *menu\$()* *on* command.

Line 310 switches on the menu. The menu will have border style 3 and will 'drop down'.

Now that the menu is displayed, we can monitor it to see which options are selected. We saw in the previous example how variables `MNBAR` and `MNSELECT` can be checked to indicate when an option is selected. This method is fine for small menus but larger systems would require a long list of *if..then* checks to establish which menu had been activated. STOS therefore offers an alternative method for checking the menu as illustrated by lines 330-360:

```
330 rem MONITOR THE MENU
340 on menu goto 1000,2000,3000
350 on menu on
360 goto 360
```

Rather than keep checking the contents of `MNBAR` for activity, we can command STOS to automatically check the menus under interrupt. This means that the menus will be checked every 50ths of a second whilst the program goes on and does other things (if required). Before starting the checking process we must tell STOS where the program should jump to when a menu is selected. This is carried out using the *on menu goto* command and is illustrated in line 340 above.

The *on menu* command is similar to the *on goto* command and sends program execution to a particular line depending on which menu is selected. If the user selects menu 1 (FILE) the program branches to line 1000, if the user selects menu 2 (RECORDS) the program branches to line 2000 and if the user selects menu 3 (SORT) the program branches to line 3000. Once the various lines have been defined we can switch on menu monitoring using the *on menu on* command as shown in line 350, and the menus will now be automatically checked by STOS. Line 360 sends the program to line 360 and thus the program simply sits at this line until a menu is selected.

When a menu is selected, the program jumps to the appropriate line. Let us consider that the FILE menu is selected and that the program

jumps to line 1000. Before servicing the requested option we must stop the user from selecting another option. The *menu freeze* command freezes the menus and stops STOS from checking same until such time that another *menu on* command is issued. The MNSELECT variable can now be checked to ascertain the actual option selected. In this example a message is printed to show the selected option but in a real situation the program would perform the required operation. Line 1060 sends the program back to line 300 where the menu is re-activated and monitored.

ALTERNATIVE MENU SYSTEMS

Drop down and pull down menus offer a flexible and convenient method of input. The menu is constantly available at the top of the screen and allows a large range of options to be presented without obstructing the main screen area. The other advantage, as previously mentioned, is that most users will be familiar with the operation of such menus as they are used on the main GEM desktop. There are, however, a number of situations where drop down or pull down menus would not be the ideal choice.

For example, consider a game which is controlled using the joystick. If using drop/pull down menus the user would have to put down the joystick, use the mouse to select an option and then return to the joystick to continue with the game. It would be far more convenient to allow the user to select an option using the joystick. In the same way, a program that relies mainly on the keyboard (maybe a word processor) may benefit from keyboard selection of options rather than mouse selection of options.

An alternative method which is proving popular on the PC range of computers is the 'pop up' menu box. This is a box that 'pops up' on the screen displaying the various options available. When a selection is made, the box is removed from the screen allowing the user to

continue where they left off. We can produce this type of menu using STOS.

By way of an example, consider a menu that could be used within a game to offer the following options:

- (1) Set level to easy
- (2) Set level to medium
- (3) Set level to hard
- (4) See High Score Table
- (5) Start Game

We shall produce three 'pop up' menus that will allow these options to be selected. The first will be controlled using the number keys, the second using the cursor keys and the third using the joystick. The programs will be presented as subroutines which can be called from a main program each time that a menu is required. You can thus use the routines within your own programs.

POP-UP MENU BOX USING NUMBER KEY SELECTION

The first program opens a window in the centre of the screen and lists the available options. The user enters the number of the required option and the window is removed. Program control is then passed back to the main, calling program where the selected option can be serviced.

Load the following:

```
10 rem MAIN PROGRAM
20 rem PROGRAM = A:\MENUS\NUMBER
30 :
40 key off : mode 0 : flash off
50 palette $0,$777,$700
```

```
60 BLACK=0 : WHITE=1 : RED=2
70 dim M$(10)
80 :
90 rem CALL SUBROUTINE
100 NO_ENTRIES=5 : BCOL=RED : FCOL=WHITE :
T$=" GAMES MENU "
110 M$(1)="Set level easy"
120 M$(2)="Set level medium"
130 M$(3)="Set level hard"
140 M$(4)="Display high scores"
150 M$(5)="Start Game"
160 gosub 1000
170 print I
180 end
190 :
1000 rem MENU SUBROUTINE- NUMBER SELECTION
1010 rem HIDE MOUSE POINTER
1020 hide on
1030 :
1040 rem FIND LONGEST ENTRY
1045 LONGEST_ENTRY=0
1050 for A=1 to NO_ENTRIES
1060 L=len(M$(A))
1070 if L>LONGEST_ENTRY then
LONGEST_ENTRY=L
1080 next A
1090 :
1100 rem CALCULATE SIZE AND POSITION
1110 WIDTH=LONGEST_ENTRY+6
1120 HEIGHT=NO_ENTRIES+6
1130 X=(40-WIDTH)/2
1140 Y=(25-HEIGHT)/2
1150 :
1160 rem OPEN THE WINDOW
1170 paper BCOL : pen FCOL
```



```
1180 windopen 1,X,Y,WIDTH,HEIGHT,5
1190 title T$
1200 print
1210 :
1220 rem DISPLAY MENU ENTRIES WITHIN WINDOW
1230 for A = 1 to NO_ENTRIES
1240 print A;" ";M$(A)
1250 next A
1260 print
1270 centre "ENTER OPTION"
1280 :
1290 rem WAIT FOR USER INPUT
1300 I$ = input$(1)
1310 I = val(I$)
1320 if I < 0 or I > NO_ENTRIES then 1300
1330 :
1340 windel 1
1350 show on
1360 return
```

Run the program and select a couple of options. Once a selection has been made, the menu is removed from the screen and control is passed back to the calling program. In this example the main program continues to display the number of the selected option.

Let us look at the programs operation. The list of options are displayed within a window as this can be displayed and removed without affecting existing information on the screen. We must therefore calculate the size and position of the window, and once displayed, monitor the users input to see which option is selected.

Lines 10-180 form the main, calling program. Line 40 switches off the function key window, sets the screen resolution low and switches off colour flashing. Lines 50 and 60 set the colour palette and line 70 dimensions variable array M\$(). This is used to maintain the various

menu options.

Lines 100-150 define the various information required by the menu subroutine and line 160 calls the subroutine. Line 100 defines the following variables:

NO_ENTRIES	indicating the number of menu entries.
BCOL	indicating the background (paper) colour.
FCOL	indicating the foreground (pen) colour.
T\$	indicating the title for the menu.

Lines 110-150 assign each menu entry to the variable array M\$() and line 160 calls the menu subroutine.

Line 1020 hides the mouse pointer.

Lines 1050-1080 form a *for..next* loop which checks the length of each menu entry in turn. Line 1060 assigns the length of the current menu entry to variable L. Line 1070 compares the length of the current menu entry to that of previous entries. The variable LONGEST_ENTRY contains the length of the longest entry so far, and if the current entry is longer, it replaces the value currently assigned to LONGEST_ENTRY. Therefore, when all the menu entries have been checked, the variable LONGEST_ENTRY will contain the length of the longest entry. This can now be used to calculate the width of the window.

Line 1110 calculates the width of the window and assigns this to variable WIDTH. This is the length of the longest menu entry plus six. The value six is based on there being a space for each side of the window, a space each side of the menu entry and two spaces to display the option number.

Line 1120 calculates the height of the window and assigns the value to variable HEIGHT. This is calculated by adding six to the number

of menu entries. This allows for a space top and bottom (to make it look neat), a space top and bottom for the sides of the window and two extra spaces to display the "ENTER OPTION" message.

Line 1130 calculates the position across the screen at which the window should be displayed (X coordinate) and assigns this to variable X. This is calculated using the expression $X = (40 - \text{WIDTH}) / 2$. Look carefully at this as it is quite easy really. There are 40 characters across the screen (low resolution) so we subtract the width of the window from 40 and divide this by 2.

Line 1140 calculates how far down the screen the window should be displayed (Y coordinate) and assigns this to variable Y. This is calculated using the expression $Y = (25 - \text{HEIGHT}) / 2$. There are 25 lines down the screen so we subtract the height of the window from 25 and then divide the result by 2 to find the central position.

Line 1170 sets the pen and paper colours to that specified by the calling program, line 1180 opens the window and line 1190 adds the title.

Lines 1230-1250 form a *for..next* loop which displays the menu entries within the open window. Notice how the loop variable (A) is used to place a number at the beginning of each entry.

Lines 1300-1320 wait for the user to select an option. The *input\$* command is used in preference to *input* so that the user can just press the required number without having to press the Return key. Remember that the *input\$* command can only operate with string variables and hence line 1310 is required to convert the entered data to numeric form. Line 1320 checks that the selected number is within range.

Once an option has been successfully selected, the window is cleared from the display, the mouse pointer is restored and program execution

is returned to the calling program.

POP-UP MENU USING CURSOR KEY SELECTION

This time the up/down cursor keys are used to move a 'highlighting' bar up and down over the menu entries. When the required entry is highlighted, the Return key is pressed to register the selection.

Load the following:

```
10 rem MAIN PROGRAM
20 rem PROGRAM = A:\MENUS\CURSOR
30 :
40 key off : mode 0 : flash off
50 palette $0,$777,$700
60 BLACK=0 : WHITE=1 : RED=2
70 dim M$(10)
80 :
90 rem CALL SUBROUTINE
100 NO_ENTRIES=5 : BCOL=RED : FCOL=WHITE :
HCOL=BLACK : T$=" GAMES"
110 M$(1)="Set level easy"
120 M$(2)="Set level medium"
130 M$(3)="Set level hard"
140 M$(4)="Display high score table"
150 M$(5)="Start Game"
160 gosub 1000
170 print POS
180 end
190 :
1000 rem MENU SUBROUTINE - CURSOR SELECTION
1010 rem HIDE MOUSE POINTER
1020 hide on
```



```
1030 :
1040 rem FIND LONGEST ENTRY
1050 LONGEST_ENTRY = 0
1060 for A = 1 to NO_ENTRIES
1070 L = len(M$(A))
1080 if L > LONGEST_ENTRY then
LONGEST_ENTRY = L
1090 next A
1100 :
1110 rem CALCULATE SIZE AND POSITION
1120 WIDTH = LONGEST_ENTRY + 4
1130 HEIGHT = NO_ENTRIES + 4
1140 X = (40 - WIDTH) / 2
1150 Y = (25 - HEIGHT) / 2
1160 :
1170 rem OPEN THE WINDOW
1180 paper BCOL : pen FCOL
1190 windopen 1,X,Y,WIDTH,HEIGHT,5
1200 title T$
1210 print
1220 :
1230 rem DISPLAY MENU OPTIONS
1240 for A = 1 to NO_ENTRIES
1250 print " ";M$(A)
1260 next A
1270 :
1280 rem POSITION HIGHLIGHT BAR ON OPTION 1
1290 POS = 1
1300 :
1310 rem MONITOR THE MENU
1320 pen HCOL
1330 locate 1,POS
1340 print M$(POS)
1350 while inkey$ = "" : wend
1360 :
```

```
1370 rem cursor up key pressed
1380 if scancode = 72 and POS > 1 then pen FCOL :
locate 1,POS : print M$(POS) : dec POS
1390 rem cursor down key pressed
1400 if scancode = 80 and POS < NO_ENTRIES then
pen FCOL : locate 1,POS : print M$(POS) : inc POS
1410 rem enter key not pressed
1420 if scancode < > 28 then 1330
1430 :
1440 windel 1
1450 show on
1460 return
```

Run the program and select a few options.

The program code is much the same as the previous example and the only real difference lies in the method used for detecting the users input. In the previous example we were looking for number input which could be detected using the *input\$* command, but this time we need to detect the up cursor, down cursor and return keys. The cursor keys do not return ASCII codes and so the *inkey\$* and *scancode* facilities must be used instead of the *input\$* command.

Lines 1240-1260 display the menu entries within the window. Notice this time that a single space is printed in front of the entry rather than a number.

Line 1290 assigns the value 1 to variable POS. This variable maintains the position of the highlight bar and will change value as the user presses the up and down cursor keys. When the menu is initially displayed we want the highlight bar on the first menu entry and hence we assign a value of 1.

Line 1320 changes the current pen colour to the highlight colour. In this example the highlight colour is red.

Lines 1330/1340 locate the text cursor at the position of the menu entry and apply highlighting. The method used to highlight an option is simply to change the pen colour and print the menu entry again so that it is re-displayed in a different colour, and hence appears highlighted.

Line 1350 forms a *while..wend* loop which continues until a key is pressed.

Line 1380 checks to see if the up cursor key has been pressed by looking at the value of the scancode. In addition to the scancode, a check is also made on the position of the highlight bar. If it is already on menu entry one then the rest of the line is skipped and the program moves on to line 1390. If the up cursor key is pressed when the highlight bar is not on menu entry one, we have to restore the current highlighted entry to its previous colour and highlight the menu entry above.

Line 1400 checks to see if the down cursor key is pressed and also that the highlight bar is not already on the last menu entry. If the highlight bar is not already on the last menu entry, we can restore the highlighted entry to the standard colour and highlight the next menu entry below.

Line 1420 checks for a press of the Return key. If the Return key is detected then the user has selected the current menu entry. The window is removed from the screen, the mouse pointer is restored and program execution is passed back to the calling program. The number of the selected option will be maintained in variable POS, and in this example, the calling program displays the value of this.

POP UP MENU USING JOYSTICK SELECTION

This is exactly the same as the previous program except the menu

entries are highlighted by moving the joystick up and down and the required option is selected by pressing the fire button.

Load the following:

```
10 rem MAIN PROGRAM
20 rem PROGRAM = A:\MENUS\JOYSTICK
30 :
40 key off : mode 0 : flash off
50 palette $0,$777,$700
60 BLACK=0 : WHITE=1 : RED=2
70 dim M$(10)
80 :
90 rem CALL SUBROUTINE
100 NO_ENTRIES=5 : BCOL=RED : FCOL=WHITE :
HCOL=BLACK : T$=" GAMES"
110 M$(1)="Set level easy"
120 M$(2)="Set level medium"
130 M$(3)="Set level hard"
140 M$(4)="Display high score table"
150 M$(5)="Start Game"
160 gosub 1000
170 print POS
180 end
190 :
1000 rem MENU - JOYSTICK SELECTION
1010 rem HIDE MOUSE POINTER
1020 hide on
1030 :
1040 rem FIND LONGEST ENTRY
1050 LONGEST_ENTRY=0
1060 for A=1 to NO_ENTRIES
1070 L=len(M$(A))
1080 if L>LONGEST_ENTRY then
LONGEST_ENTRY=L
```



```
1090 next A
1100 :
1110 rem CALCULATE SIZE AND POSITION
1120 WIDTH=LONGEST_ENTRY + 4
1130 HEIGHT=NO_ENTRIES + 4
1140 X=(40-WIDTH)/2
1150 Y=(25-HEIGHT)/2
1160 :
1170 rem OPEN THE WINDOW
1180 paper BCOL : pen FCOL
1190 windopen 1,X,Y,WIDTH,HEIGHT,5
1200 title T$
1210 print
1220 :
1230 rem DISPLAY MENU OPTIONS
1240 for A=1 to NO_ENTRIES
1250 print " ";M$(A)
1260 next A
1270 :
1280 rem POSITION HIGHLIGHT BAR ON OPTION 1
1290 POS = 1
1300 :
1310 rem MONITOR THE MENU
1320 pen HCOL
1330 locate 1,POS
1340 print M$(POS)
1350 wait 8
1360 :
1370 rem cursor up key pressed
1380 if jup and POS > 1 then pen FCOL : locate 1,POS
: print M$(POS) : dec POS
1390 rem cursor down key pressed
1400 if jdown and POS < NO_ENTRIES then pen FCOL
: locate 1,POS : print M$(POS) : inc POS
1410 rem enter key not pressed
```

```
1420 if fire = 0 then 1320
1430 :
1440 windel 1
1450 show on
1460 return
```

The program code is the same as the previous example except that the *jup*, *jdown* and *fire* functions have been used to check the state of the joystick. The *wait* command in line 1350 controls the speed at which the highlight bar will move through the menu entries.

Whilst on the subject of communicating with the user, we shall take a quick look at another couple of techniques for adding to the general appearance and user friendliness of a program.

INFORMATION BOXES

Quite often the programmer may need to pass a message to the user. For example, the user may type in the wrong information and a quick message is required to put them right. A pop up information box displays information on the screen for the user to read. When the message has been read, the box is removed by pressing any key on the keyboard or either of the mouse buttons. Once again the program is presented as a subroutine so that it may be used within your own programs.

Load the following:

```
10 rem MAIN PROGRAM - INFORMATION BOX
20 rem PROGRAM = A:\MENUS\INFOBOX
30 :
40 key off : mode 1 : flash off
50 palette $0,$777,$7
60 BLACK=0 : WHITE=1 : RED=2
```



```
70 dim INFO$(10)
80 :
90 rem CALL SUBROUTINE
92 X=20 : Y=5 : NO_LINES=6 : BCOL=RED :
FCOL=WHITE
100 INFO$(1)="This is a pop-up information box."
110 INFO$(2)=""
120 INFO$(3)="When the user has finished reading"
130 INFO$(4)="the information, they can press any"
140 INFO$(5)="key on the keyboard or either mouse"
150 INFO$(6)="key to make the box disappear."
160 gosub 1000
170 end

1000 rem DISPLAY INFORMATION BOX
1010 rem FIND THE LONGEST LINE
1015 LONGEST_LINE=0
1020 for A=1 to NO_LINES
1030 L=len(INFO$(A))
1040 if L>LONGEST_LINE then LONGEST_LINE=L
1050 next A
1060 :
1070 rem CALCULATE SIZE OF WINDOW
1080 WIDTH=LONGEST_LINE+4
1090 HEIGHT=NO_LINES+3
1100 :
1110 rem DISPLAY INFO BOX
1120 paper BCOL : pen FCOL
1130 windopen 1,X,Y,WIDTH,HEIGHT,4
1140 curs off
1160 for A=1 to NO_LINES
1170 print " ";INFO$(A)
1180 next A
1190 :
1200 rem WAIT FOR USER TO EXIT
```

```
1210 while inkey$ = "" and mouse key = 0 : wend
1220 windel 1
1230 return
```

Run the program and an information box will be displayed on the screen. Press any key or either of the mouse buttons and the box will close.

The calling program must define the following information:

X	the X text coordinate of the top left corner of the box.
Y	the Y text coordinate of the top left corner of the box.
NO_LINES	the number of lines of text to display in the box.
BCOL	the background or paper colour of the box.
FCOL	the foreground or pen colour of the box.
INFO\$()	the lines of information to be displayed in the box.

Lines 1010-1050 calculate the length of the longest line of text.

Lines 1070-1090 calculate the size of the window.

Lines 1110-1180 display the text inside the window.

Lines 1200-1230 wait for the user to press a key on the keyboard or a button on the mouse. When a key or mouse button is pressed, the window is removed from the screen and program execution is returned to the calling program.

DIALOGUE BOXES

One step on from the pop-up information box is the dialogue box. These are similar to information boxes except that they also allow the user to pass some information back. For example, we may like to ask the user if they want to play the game again. The user could then

specify yes or no.

Load the following:

```
10 rem DIALOGUE BOX
20 rem PROGRAM = A:\MENUS\DIALOG
30 :
40 key off : mode 0 : flash off
50 palette $0,$777,$7
60 BLACK=0 : WHITE=1 : BLUE=2
70 dim I$(3)
80 :
90 rem CALL SUBROUTINE
100 I$(1)="Would you like to play again"
110 B1$=" YES" : B2$=" NO"
120 pen WHITE : paper BLUE
130 gosub 1000
140 print WIN
150 end
160 :

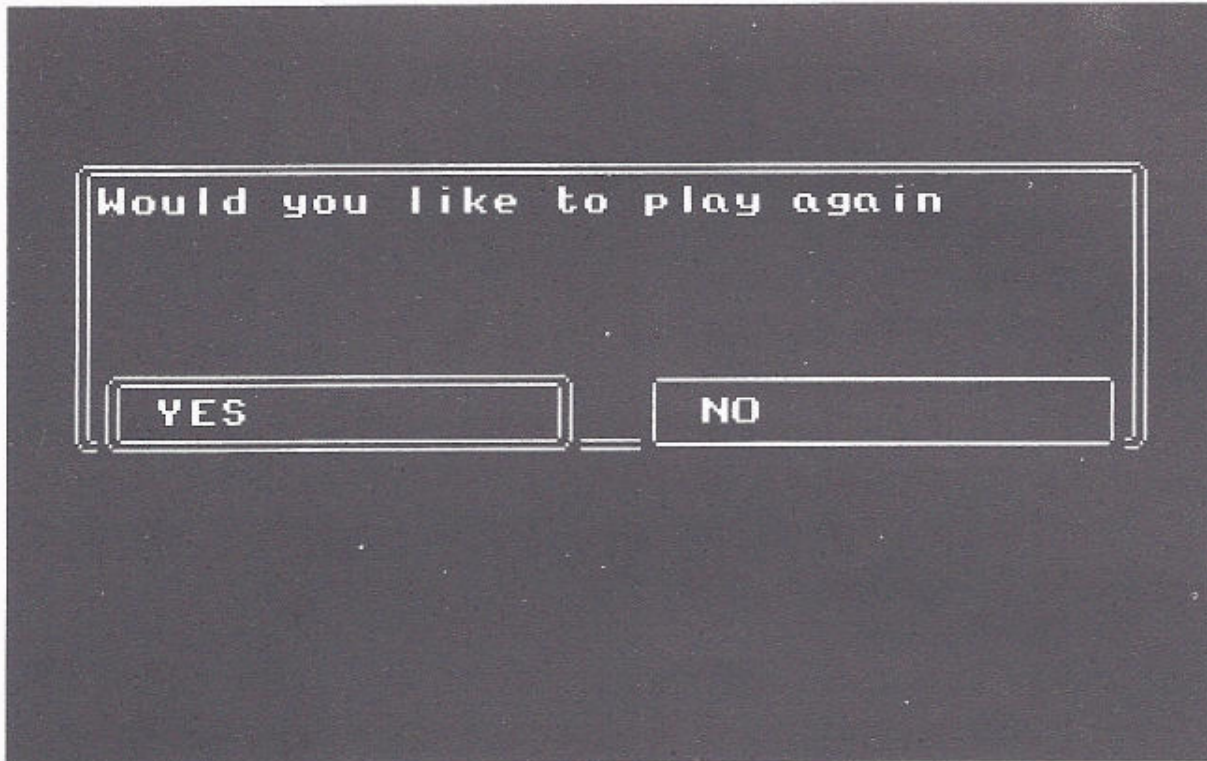
1000 rem DIALOGUE BOX SUBROUTINE
1010 rem OPEN MAIN WINDOW
1020 windopen 1,2,5,36,10,3
1030 :
1040 rem DISPLAY TEXT
1050 for A=1 to 3
1060 print I$(A)
1070 next A
1080 :
1090 rem DISPLAY BUTTONS
1100 windopen 2,3,12,16,3,3
1110 print B1$;
1120 set zone 1,28,100 to 145,115
1130 windopen 3,21,12,16,3,1
```

```
1140 print B2$;
1150 set zone 2,170,100 to 290,115
1160 :
1170 rem MONITOR THE ZONES
1180 WIN = 1
1190 :
1200 while mouse key = 0 or Z = 0
1210 Z = zone(0)
1220 if Z = 1 and WIN < > 1 then window 2 : border 3
: window 3 : border 1 : WIN = 1
1230 if Z = 2 and WIN < > 2 then window 3 : border 3
: window 2 : border 1 : WIN = 2
1240 wend
1250 :
1260 rem REMOVE DIALOGUE BOX AND EXIT
1270 windel 3
1280 windel 2
1290 windel 1
1300 return
```

Run the program and it will produce a dialogue box as shown on the opposite page.

A message or question is displayed at the top of the box and the user responds or makes a selection by clicking the mouse pointer on either of the buttons at the bottom of the box. This example allows a maximum of three lines of text with two buttons, but you could modify the subroutine to include more information or more buttons. The dialogue box, as per the other programs, has been presented as a subroutine so that it may be called a number of times from within the same program. Let's see how it works.

The following parameters must be defined before calling the subroutine:



I\$(): the variable array I\$ contains the lines of information that are to be displayed within the dialogue box. In this case the program allows three lines.

B1\$ & B2\$: these define the text that should be displayed within the two selection buttons. There is only room for limited information and hence this information should be limited to simple expressions such as YES, NO, EXIT, CONT, etc.

The subroutine uses the current pen and paper colours so these should also be set.

Line 1020 opens the main window on the screen. This has been designed for a low resolution screen but the size and position could be changed as required.

Lines 1040-1070 display the three lines of text within the window.

Lines 1090-1150 display the buttons. The buttons are themselves

windows which are displayed within the main dialogue box window. A zone is set up to cover each of the buttons and these are tested to detect the presence of the mouse pointer over the buttons. Notice that one of the buttons has border style 3 whilst the other has border style 1. As the mouse pointer moves over the buttons, they are highlighted using border style 3.

The dialogue box is now complete and we can monitor the mouse. Line 1180 assigns a value of one to variable WIN. The variable WIN maintains the number of the currently highlighted button and can thus equal either one or two.

Lines 1200-1240 form a *while..wend* loop which monitors the state of the mouse buttons and the position of the mouse pointer. Line 1210 checks the zones and assigns the result to variable Z.

When the mouse pointer moves over button one, line 1220 changes the highlighting from button two to button one. When the mouse pointer moves over button two, line 1230 changes the highlighting from button one to button two.

When a button has been selected the dialogue box can be removed from the screen. Lines 1270-1290 remove the windows from the screen and line 1300 passes program control back to the calling program.

MEMORY BANKS

When using 'drop down' or 'pull down' menus, STOS maintains the menu information in memory bank 15. Menu data is similar to sprite data in that the appropriate memory bank is automatically assigned by STOS and thus requires no reserving by the programmer. Therefore, if your program uses menus be careful not to reserve memory bank 15 for any other use.

Chapter 26

And Finally

Congratulations on completing the course and I hope that you have found it a worthwhile and enjoyable project. To bring the course to a close we shall take a brief look at methods of optimising the programming environment and program code.

CUSTOMISING THE EDITOR

The STOS editor offers a flexible and easy to use environment for the development of our programs but can be customised to operate in the way, you, the programmer require. You may already be happy with the editor but STOS is supplied with a program named CONFIG.BAS which allows the editor to be configured. It allows the default screen resolution and the colour palette to be defined, the contents of the function keys to be assigned, accessory programs to be automatically loaded, etc. When all these features have been defined, the editor will automatically resume as specified each time that it is loaded.

ENGLISH OR FRENCH EDITOR

STOS Basic was developed in France and the editor can generate French or English dialogue. English is set as default but this can be changed using either the CONFIG.BAS program or a direct command as shown below:

francais

changes the editor to French language.

english

changes the editor to English language.

UPPER OR LOWER CASE

The editor can also operate in either upper or lower case mode. You may have noticed throughout this course that all the program listings have displayed the program commands in lower case and the variable names in upper case. The programmer may, if they wish, reverse this so that program commands are displayed in upper case and variable names in lower case. This is set using the *upper* and *lower* commands which should not be confused with *upper\$* and *lower\$*.

UPPER

```
10 FOR a=1 TO 10  
20 PRINT a  
30 NEXT a
```

LOWER

```
10 for A=1 to 10  
20 print A  
30 next A
```

You may like to set this to what ever you prefer.

USING SPRITES

You may have noticed, when playing the Alien Attack game in chapter 20, that the movement of the gun sight was a little 'jerky' at times. STOS has to work very hard when controlling sprites. Simply

placing the sprites on the screen presents no problems but movement and animation require a lot of hard work. The positioning, moving, screen updating, etc. all takes time and this can lead to a reduction in speed and smoothness of the display. To ensure optimum performance of sprites always consider the following:

- * The larger the sprites the slower they will be move. Therefore, if your game requires high speed, keep the sprites as small as possible.
- * Limiting the number of sprites will increase speed. If you only require a couple of sprites they can be large, but if you require all fifteen sprites then they should be as small as possible.

The smoothness and speed of the sprite operations thus relies on the combination of size and number. The speed of a program can also be increased quite considerably by compiling the program - more on this in a moment.

The Alien Attack game used the full screen area, good sized sprites and a lot of movement. This was designed to illustrate programming technique but the smoothness of the sprite operations could be improved by any of the following methods:

- * The game play area could be reduced thus reducing the range of movement required by each sprite.
- * The size of the alien sprites could be reduced.
- * The number of alien sprites could be reduced.
- * The amount of simultaneous sprite movement could be reduced. The game moves all ten alien sprites down the screen at the same time but could be modified to move only five at a time.

INTERPRETERS AND COMPILERS

STOS Basic is known as an interpreter. This means that it interprets the commands entered by the programmer and converts them to the machines native language - 68000 machine code. The computer cannot directly understand commands such as *print*, and whilst some prefer to program direct in machine code, many programmers prefer to use a high level language such as STOS. Machine code is quite difficult to learn and use, requiring a lot of program code to perform even the simplest of tasks. Programs can therefore be produced a lot faster using high level languages - but there is one a drawback.

When you run a program an interpreter such as STOS has to convert each command to machine code before it can be executed. This conversion takes time and thus an 'interpreted' program will execute a lot slower than a machine code program. In addition to this the interpreter will also be required each time that you wish to run the program. This presents no problem for the hobbyist programmer, but if you wish to offer your software on a commercial basis, your potential customers may not have the STOS interpreter and thus would not be able to use your software. Luckily STOS offers two solutions to this problem.

(1) CREATE A RUN ONLY PROGRAM

This allows the programmer to produce a version of the program that can be executed directly from the GEM desktop without the need for the main STOS interpreter. The exact procedure for producing a 'run only' program is detailed in the STOS Users Guide. As the name suggests, the program can be run but cannot be listed and thus no one can actually gain access to the program listing. This method works by storing a protected copy of the STOS folder along with the program so that your program can still be interpreted but does not require loading in to the interpreter first.

(2) COMPILING A PROGRAM

The 'run only' format allows programs to be distributed but the final program is still interpreted. The second option therefore involves converting the complete program to machine code and hence doing away with the interpreter all together. This conversion process is known as *compiling* and the compiled program, now pure machine code, runs a lot faster as no interpreting of commands is required. Unfortunately the STOS compiler is not supplied as part of the main STOS package but it is not very expensive and is a worthwhile investment for the serious programmer.

OTHER 'ADD ON' PACKAGES

In addition to the compiler, the authors of STOS also offer three other 'add on' packages. The first is the Maestro package that we mentioned in chapter 21. This consists of a hardware cartridge plus associated software that allows sounds and music to be sampled and reproduced from within STOS. This is a very exciting product as it opens up a whole new world to the programmer and can also be obtained at a very reasonable price. The package comes with a vast array of software demonstrating how drum machines, sound effects and even complete sound sampling packages can be produced.

The second 'add on' package involves sprites. Sprites can be produced by the programmer or grabbed from pictures using the sprite designer accessory, but producing professional sprites does require a degree of artistic skill and time. With this in mind the producers of STOS offer a package of ready made, very professional looking sprites that you can include within your own programs.

The final 'add on' package is the 3D extension kit. This is designed for people interested in graphics and adds commands to STOS for generating and manipulating three dimensional graphics.

STOS EXTENSIONS

If the STOS Basic language is not powerful enough already, it can also be extended through the use of extensions. Extensions are separate blocks of commands that can be added to STOS for use within programs. These allow facilities to be added that STOS does not already have or commands to be added that perform a function better or in a different way to the standard commands. The production of extensions requires an understanding of machine code but there are a number of Companies and individuals producing such packages. These may allow such things as accessing extra colours on STE machines, displaying unsupported picture file formats, etc. and can normally be obtained from public domain libraries.

COMPATIBILITY

At the time of writing this course, STOS Basic is compatible with all Atari STFM/STE machines. Updating disks can be obtained from most public domain libraries which will bring any older versions of STOS up to the latest revision. Hopefully updates will soon become available to allow STOS to be used on Mega STE's and the FALCON.

INDEX

Acos	72	CLEARING THE SCREEN	8
ALIEN ATTACK	431	Click off	462
ANIMATION	286,382	Cls	9
Anim	382	Clw	199
Anim on	382	COLLISIONS	401
Appear	277	Detect single	402
Arc	244	Detect multiple	407
ART MASTER	335	Collide	401
Asc	30	COLOUR	
ASCII CODES	30	Defining individual	
asin	72	colours	35
atan	72	Defining complete	
Autoback off	305	colour palette	37
Autoback on	305	Flashing	39
Bar	233	Shifting	43
Bell	479	Colour	35
Bin\$	496	Cos	72
BONK THE GONK	313	DATABASE	535
Boom	478	Data	170
Border	188	DECISIONS	83
Box	230	Def scroll	290
CENTRED TEXT	25	Deg	74
Centre	26	dfree	248,503
Change mouse	160, 259	DIALOGUE BOXES	593
CHARACTERS SETS	205	Dim	55
Large Text	211	DIRECTORIES	489
Mixing sets	212	Dir	492
Chr\$	30	Dir\$	496
Circle	241	Draw	223
Clear	50	Drive	496

Drive\$	496	Francais	600
DROP DOWN MENUS	570	Fsave	501
Drvmap	496	FUNCTION KEYS	
Earc	246	Assigning Strings	157
EDITOR		Listing contents	157
Moving Around	2	Window	9
Control Keys	5	Get	511
Ellipse	241	Get palette()	266
English	600	Gosub	107
Envel	480	Goto	103
Epie	246	GRAPHICS	219
Erase	209	Arcs - circular	243
ERASING PROGRAMS	19	Arcs - elliptical	246
EXPANDING BOXES	310	Boxes - outlined	230
Fade	268	Boxes - filled	232
Field #	510	Circles	240
FILES	487	Ellipses	242
Drive map	497	Graphs - line	233
Files + folders	489	Graphs - bar	239
File selector	498	Lines - standard	222
Floppy disks	488	Lines - custom	226
Random Access	508	Painting	248
Reading directory	492	Pixels & points	221
Selecting Files	495	Polygons	230
Sequential Filing	495	Resolution	219
Fileselect\$	498	GUESS THE NUMBER	111
Fire	167	GUESS THE WORD	173
FIRING MISSILES	389	GUNS	
Fkey	155	Joystick control	397
FLASHING COLOURS	39	Mouse control	396
Flash	40	Hardcopy	310
Fload	501	Hcos	72
FLOPPY DISKS	488	Hide on	159
FOLDERS	489	Hsin	72
For..next	96	Htan	72

If..Then..Else	83, 85	LISTING A PROGRAM	9
INFORMATION BOXES	591	List	9
Ink	221	Load	500
Inkey\$	150	LOCATING TEXT	28
Input	62, 142	Locate	29
Input #	507	LOOPS	
Input\$()	148	For..next	96
Instr	136	Repeat..until	90
INVERSE TEXT	22	While..wend	93
Inverse off	22	Lower	600
Inverse on	22	Lower\$	134
Jdown	167	MATHS OPERATIONS	
Jleft	167	Basic mathematics	58
JOYSTICKS	166	Areas	60
Joy	168	Volumes	67
Jright	167	Pythagoras'	70
Jup	167	The sine Rule	72
Key	10	The cosine rule	76
KEYBOARD INPUT		The tangent rule	80
Inkey\$	149	MATHS QUIZ GAME	121
Input	142	MEMORY BANKS	205
Input\$()	148	MENU SYSTEMS	569
Line input	146	Menu	573
Scancodes	152	Menu freeze	579
Key off	10	Menu off	574
Key()	157	Menu on	573
Keylist	157	Menu\$	573
Kill	502	Menu\$() off	577
LARGE TEXT	211	Menu\$() on	577
Left\$	136, 137	Mid\$	136, 137
Len	132	Mkdir	502
Let	48	Mnbar	574
Limit mouse	163	Mnselect	574
Line input	146	Mode	24
LINE NUMBERS	7	MORSE CODE	404

MOUSE		Palette	37
Buttons	104	Paper	33
Hiding	159	Pen	33
Limiting	163	PI	64
Pointer Style	160	Pie	245
Position	162	Play	462
Showing	159	Plot	221
Mouse	259	Polygon	229
Mouse key	164	Polyline	227
Move freeze	443	POP UP MENUS	
Move on	387	Number Control	580
Move x	386	Cursor Control	585
Move y	388	Joystick Control	588
Movon()	401	Previous	503
MUSIC	461	PRINT ATTRIBUTES	
Envelopes	477	Underline	21
Multiple Voices	471	Shadow	21
Music Accessory	472	Invert	21
Musical Notes	467	Print	7
Single Tones	462	Print #	506
Sound Effects	478	PROGRAM CONTROL	
Sound Sampling	482	Gosub..return	106
Tremolo	475	Goto	103
Music	474	If..then	103
New	18	On..gosub	108
Noise	479	On..goto	105
NUMERIC VARIABLES	47	PROGRAM DESIGN	
On menu	578	Definition	112
On menu goto	578	Design	113
On..gosub	108	Construct & Code	113
On..goto	105	PULL DOWN MENUS	574
Open	510	Put	510
Open in	507	Qwindow	195
Open out	506	RANDOM ACCESS	508
Paint	250	Rad	75

Rbar	233	logic	258
Rbox	231	Picture formats	255
Read	170	Reducing	279
Reduce	279	Scrolling	288
Rem	11	Slide Shows	
REMARKS	11		265,275,278
REMOVING LINES	12	Zooming	279
REPLACING LINES	12	Screen copy	266,285
Rename	501	Screen swap	264
RENUMBERING	15	Scrn	527
Renum	15	SCROLLING	
Repeat..until	90	Diagonal	297
Reserve as datascreen		Horizontal	293
	267	Text lines	299
Reserve as screen	265	Vertical	289
Reserve as set	208	Windows	201
RESTORING PROGRAMS	19	Scroll	291
Return	108	Scroll down	203
Right\$	136, 137	Scroll off	202
Rmdir	502	Scroll on	202
Rnd	87	Scroll up	203
RUNNING A PROGRAM	8	SEMICOLONS	17
Run	8	SEQUENTIAL ACCESS	504
Save	501	Set line	226
SCANCODES	152	Set paint	248
Scancode	152	Set zone	253
SCANNED PICTURES	257	SHADED TEXT	22
SCREEN		Shade off	22
Animation	286	Shade on	22
Appearing	276	SHIFTING COLOURS	43
Copying	285	Shift	43
Fading	268	Shoot	479
Flipping	263	SHOOT THE SPOOK	411
Memory banks	265	Show on	160
Physic, back &		Sin	72

SLIDE SHOWS	265,275,278	Under on	22
SOUND EFFECTS	478	Unnew	19
Space\$	27	Upper	600
Sprite	377	Upper\$	134
Sprite off	378	USER INPUT	141
Sqr	70	Function keys	155
Step	101	Joystick	166
Str\$	509	Keyboard	142
STRING VARIABLES	127	Mouse	159
Addition	129	VARIABLES	
Creating strings	133	Arrays	54
Length	132	Names	53
Substrings	136	Types	51
Subtraction	129	Volume	462
Upper & Lower		Wait vbl	163
Case	134	While..wend	93
SPRITES	375	WINDOWS	185
Animation	382	Borders	187
Creating	378	Character Sets	205
Displaying	376	Closing a window	190
Hot spots	379	Coloured windows	198
Movement	385	Moving windows	196
Palette	380	Multiple windows	192
String	134	Opening a window	186
String\$	134	Scrolling	201
SUBROUTINES	106	Special Effects	213
Swap	124	Titles	192
System	483	Windel	191
Tab	27	Windmove	196
Tan	72	Windopen	187
Tempo	475	Window	195
Title	192	WORD PROCESSOR	515
Tremolo	475	X mouse	162
UNDERLINED TEXT	22	Xor	451
Under off	22	Y mouse.	162

Zone	253
Zoom	279