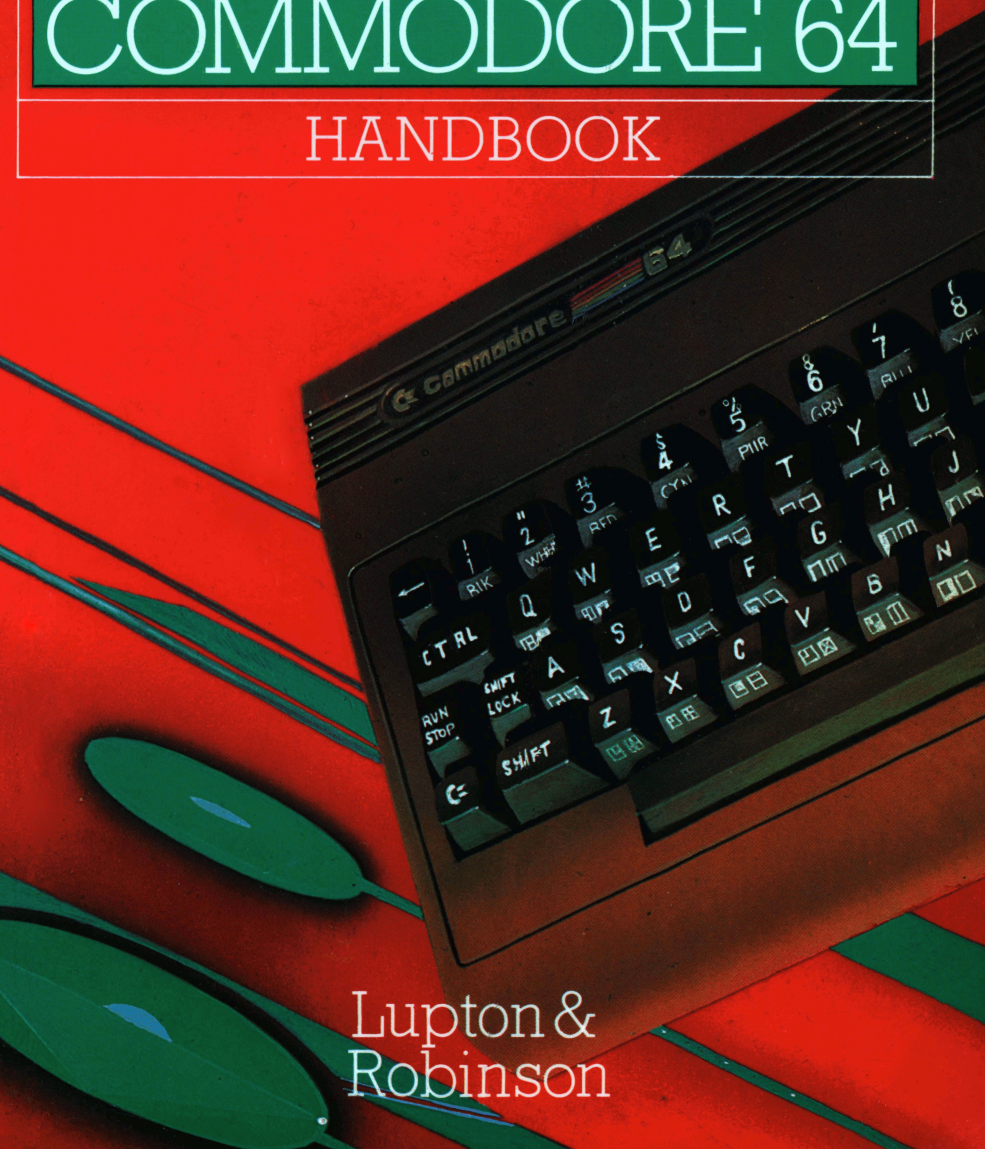
CENTURY COMMUNICATIONS

THE ADVANCED COMMODORE 64 HANDBOOK



Lupton & Robinson

THE ADVANCED COMMODORE 64 HANDBOOK

THE ADVANCED COMMODORE 64 HANDBOOK

PETER LUPTON & FRAZER ROBINSON

CENTURY COMMUNICATIONS

LONDON

Copyright © Peter Lupton and Frazer Robinson 1984

All rights reserved

First published in Great Britain in 1984
by Century Communications Ltd,
12 - 13 Greek Street,
London W1V 5LE

ISBN 0 7126 0420 0

Printed in Great Britain in 1984 by
Hazell, Watson and Viney
Aylesbury,
Bucks.

Typeset by Spokesman, Bracknell

NOTE : The programs in this book have been typeset directly from the Commodore 64.

CONTENTS

Acknowledgements

Introduction

1	BASIC and How it Works	1
2	Machine Code on the 64	24
3	Bit-Mapped Graphics - Part 1	52
4	Bit-Mapped Graphics - Part 2	70
5	Display Interrupts	91
6	Programs and People	105
7	Making Music	121
8	The 1541 Disk Drive	138
9	Advanced Disk Operations	154
10	The MPS801 Printer	193

APPENDICES

1	Abbreviations	207
2	DOS Error Messages	208
3	Summary of DOS Commands	213
4	BASIC Tokens and Abbreviations	214
5	Musical Notation	217
6	Kernal Routines	221
7	Graphics Loader Programs	227
8	The 6510 Instruction Set	240
9	SID Registers	248
10	VIC II-Registers	250

Index 252

ACKNOWLEDGEMENTS

Thanks to Bob Dinsdale – 'Illustrator of the Year' – for all his work, and to those colleagues and friends who continue to support and encourage us.

Special thanks to Kath for cooking and to Celia for doing the washing up when we hadn't the time to do either.

Finally, Anne, Dave, Paul, Noel and Jan, Simon and Angela, Ray and Ina, Rob and Alison, Smokey the cat and Sandy the dog all deserve a mention for putting up with us on our rare weekends off.

INTRODUCTION

This book is a companion volume to our *Commodore 64 Handbook*, and takes the reader with a working knowledge of BASIC and the Commodore 64 on to a deeper understanding of the machine.

A description of how BASIC works and how it can be enhanced by adding new commands is given, along with programs to demonstrate the techniques used.

Many of the programs in this book are written in machine code, and an introduction to the 6510 microprocessor, its instruction set and the 64's ROM routines is given. All the machine code programs are well documented, but the reader who isn't familiar with machine code programming is provided with all the information needed to make use of them in his own programs.

A set of machine code utilities which give the functions HIRES, LORES, PLOT, DRAW and FILL are provided, together with many programs and ideas for their use.

The ways in which music can be created and stored are covered, along with advice in transcribing sheet music to the 64.

A chapter is devoted to interrupts and how they may be used to create mixed mode displays, allow more than eight sprites on the screen and other useful features.

The 1541 disk drive, the DOS and the various file structures it supports are described, and the final chapter covers the use of printers with the 64.

The book is arranged to allow you to pick out a topic at random and explore it in depth, and we hope it contains something of interest and value for every Commodore 64 owner.

Peter Lupton and Frazer Robinson 1984

CHAPTER 1

BASIC AND HOW IT WORKS

You are probably aware that the microprocessor in your 64 doesn't understand BASIC and needs to be programmed in its own language - machine code. To enable the 64 to run your BASIC programs it comes equipped with a large machine code program called the *BASIC interpreter.* As the name suggests this program interprets a BASIC program - converting it into a series of machine actions, the instructions for which are subroutines of the interpreter program.

Machine code is covered in more detail later in this book - this chapter deals with how the 64 stores and runs BASIC programs and shows how you can get more out of your 64 by understanding the techniques involved.

Interpreter and Kernal ROMs

The ROM memory in the 64 contains two machine code programs - the basic interpreter mentioned above and the *kernal.* The kernal is responsible for all the functions needed to operate the 64 - reading the keyboard, arranging the memory, operating the screen editor and so forth. Both the interpreter and the kernal reserve some of the RAM for their own use, generally speaking in the first 2k of memory.

HOW BASIC IS STORED

When you type in a line of a BASIC program, routines in the kernal take the character for each key press and store it in screen memory. When you press RETURN the entire line is copied from the screen into memory. The BASIC interpreter reads the first few characters of the line to see if they form a line number and, if so, the line is inserted into the stored program. This system allows you to edit a line using the cursor keys without retyping the entire line.

BASIC programs are normally stored starting from location 2048. To see this type in the following one line program and then enter the immediate mode commands following it.

```
10 PRINT "TEST": GOTO 10

FOR Z=2048 TO 2068:PRINT Z,PEEK(Z):
NEXT Z
```

The display will look like the two left hand columns in the list below if you have typed the program in exactly as it appears:

ADDRESS	CONTENTS	COMMENT
2048	0	The first byte of a program is always zero
2049	19	Low byte of link pointer
2050	8	High byte of link pointer
2051	10	Low byte of line number
2052	0	High byte of line number

2053	153	PRINT
2054	32	(space)
2055	34	"
2056	84	T
2057	69	E
2058	83	S
2059	84	T
2060	34	"
2061	58	:
2062	137	GOTO
2063	32	(space)
2064	49	1
2065	48	0
2066	0	Null
2067	0	Null
2068	0	Null

The comment to the right of each item in the list show what each location contributes to the program line.

The contents of location 2048 are always zero if a BASIC program is stored in the 64.

Locations 2049 to 2065 hold the BASIC line and have the following functions:

The first two bytes, 2049 and 2050, indicate the position of the next line of BASIC – more about this later.

Bytes 3 and 4 of the line (2051 and 2052) are the line number in the order low byte, high byte. In this case the number is 10 – that is 10 + 256*0.

The next byte contains the value 153 which is a shorthand for **PRINT**. This is because the 64 stores BASIC programs in a compressed form using a system of *tokens* whereby single byte codes are used to represent BASIC reserved words. This method offers significant memory savings over storing commands as a series of ASCII characters and also increases the speed at which programs run. When you pressed RETURN and entered the program line the kernal program recognised the **PRINT** as BASIC reserved word by comparing it with a list kept in ROM starting at location 41118. The token 153, used to replace the **PRINT** command, represents the position of that command in the list with 128 added to it, so **PRINT** is the 25th item in the list. The following program displays a section of the BASIC reserved word table from the ROM.

```
10    REM DISPLAY RESERVED WORDS
15    PRINT "{CLS}"CHR$(14)
20    FOR Z=41118 TO 41250
30    A$ = CHR$(PEEK(Z))
40    PRINT A$;
50    IF ASC(A$)<91 THEN 70
60    N=N+1 : PRINT CHR$(13);N,
70    NEXT Z
```

You will see that **PRINT** is the 25th item in the list as mentioned above.

When a program is listed the tokens are converted back into the BASIC words they represent, making the system transparent to the user.

Another advantage of the token system is that it allows you to type BASIC commands in an abbreviated form, since only the first two or three characters are checked by the interpreter. A full list of BASIC reserved words, abbreviations and tokens is given in Appendix 4.

The next eight bytes in the list are simply the ASCII representation of the characters you typed as part of the program line.

Location 2062 is another token, this time 137 representing the **GOTO** command, and is followed by a space. The next two bytes are the line number following the **GOTO** command, 10, but stored in their ASCII form, *not* as a binary number. The end of a BASIC program is indicated by three consecutive zeros.

Let's return to the two bytes labelled 'link pointer' at the start of the list. These two bytes form the low and high bytes respectively of the address where the next line of the program is stored. In this case they point to address 2067 (19 + 8*256 = 2067), which is the end of the program.

To see how the link pointer works, try adding the following line to the program and examining memory locations 2066 to 2074.

```
20    REM

FOR Z=2066 TO 2074:PRINT Z, PEEK(Z):
NEXT Z
```

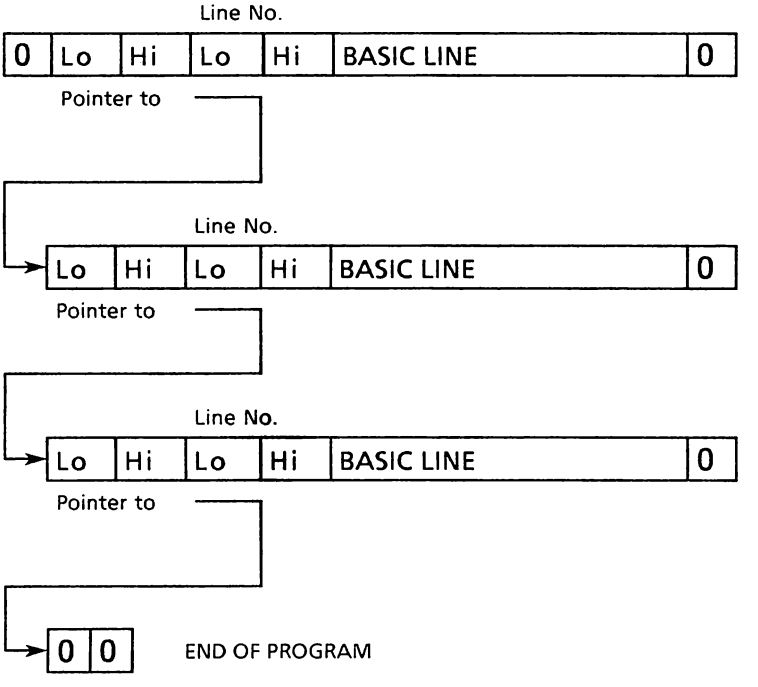
You will get a list like this:

2066	0	A null for the end of line 10
2067	25	Low byte of link pointer
2068	8	High byte of link pointer
2069	20	Low byte of line number
2070	0	High byte of line number
2071	143	REM token
2072	0	Null
2073	0	Null
2074	0	Null

You should be able to see that there is now only one zero after line 10, and that location 2067 contains the low byte of the link pointer to the new end of the program. Remember that the link pointer for line 10 pointed to location 2067 ?

As before the third and fourth bytes contain the low byte/ high byte representation of the line number and 143 is the token for **REM**. The program again terminates in three consecutive zeros.

The diagram opposite summarises how BASIC programs are stored.



The linked nature of a BASIC program

Renumbering Programs

We can make use of this knowledge to renumber BASIC programs. When a program is being developed it is often useful to be able to 'open up' gaps between lines to fit in new lines, and the next program allows you to do just that.

```
60000 REM RENUMBER
60010 INPUT"START";S
60020 INPUT"END";E
60030 INPUT"NEW START";NS
60040 INPUT"INCREMENT";I
60050 A=2049
60060 Q=PEEK(A+2)+256*PEEK(A+3)
60070 IF Q<S THEN 61000
```

```
60080 IF Q>E THEN PRINT"FINISHED":
      END
60090 POKE A+2,NS-INT(NS/256)*256
60095 POKE A+3,INT(NS/256):NS=NS+I
61000 A=PEEK(A)+256*PEEK(A+1)
61010 IF A=0 THEN END
61020 GOTO 60060
```

The program is numbered so it can form the last part of any program you are developing. Whenever you need to renumber simply type:

```
RUN 60000
```

and enter the line numbers of the beginning and end of the program to be renumbered and the new start number and increment.

This is a very crude and simple renumbering program and doesn't take account of the line numbers following **GOTO**, **GOSUB** and **THEN** commands. To write a program to alter these is more difficult because these line numbers are stored in their ASCII form and so occupy a varying number of bytes. This means that if a line number specified by a **GOTO** command is three figures long and is required to become a four figure number when the program is renumbered, the entire BASIC program would need to be moved up in memory by one byte to create space. Such a program would not be impossible to write in BASIC but it would be quite slow!

A possible way around this would be always to start your program numbering at line 10000 - thereby ensuring that every **GOSUB**, **GOTO** or **THEN** reference was a five figure number. If you want to try this, refer to the table of BASIC commands and their tokens in Appendix 4.

EXECUTING A BASIC PROGRAM

A BASIC program stored in memory is of little use until it can be executed. This is achieved by typing **RUN** and pressing RETURN. The **RUN** is interpreted as an immediate command and the appropriate part of the BASIC interpreter program is called. This subroutine initialises all pointers in zero page and closes all open channels. The first line of the program is then interpreted by loading it a character at a time into the accumulator and passing control to the appropriate subroutines as a command is found. This process is carried out by a small machine code subroutine called CHRGET which is copied from ROM into zero page RAM (locations 115 to 138) when the 64 is switched on.

Below is a disassembly listing of CHRGET:

```
START    115    E6 7A        INC  $7A
         117    D0 02        BNE  $02
         119    E6 7B        INC  $7B
FETCH    121    AD B7 12     LDA  $12B7
         124    C9 3A        CMP  #$3A
         126    B0 0A        BCS  RETURN
         128    C9 20        CMP  #$20
         130    F0 EF        BEQ  START
         132    38           SEC
         133    E9 30        SBC  #$30
         135    38           SEC
         136    E9 D0        SBC  #$D0
RETURN   138    60           RTS
```

The first operation of CHRGET is to increment the low byte of the character pointer – two locations pointing to the next character of the BASIC text to be accessed. If incrementing the low byte of the character pointer causes an overflow, then the high byte is incremented. The character pointer is at locations $7A and $7B (122 and 123) which form part of the CHRGET program, so the program is

self modifying! The section of the program labelled FETCH reads the character pointed to by the character pointer into the accumulator. This location is variable and the value in the listing above is what happened to be present in our 64 when the listing was created.

If the character is a colon (:), signifying another statement on this line, then control jumps back to other routines in the interpreter which execute the command just fetched before returning for the next one.

If a space is encountered then the next character is read in.

This routine is very important since it gives you a chance to intercept the processing of BASIC programs and implement additional features, by replacing the code at the beginning of the routine by two JSR commands calling two of your own routines. The second routine simply copies that part of CHRGET overwritten by the JSR instructions while the first is the new user defined code.

This technique can be used to add new commands to the BASIC language as the following simple example shows:

ADDING NEW COMMANDS TO BASIC

To add a new command to BASIC there are two steps to follow:

a) Modify the first three bytes of CHRGET to perform a JSR to the routine which interprets and carries out the new command.

b) Modify the next three bytes of CHRGET to perform a JSR to a subroutine which is a copy of

the first six bytes of the original CHRGET routine.

For example, suppose you wanted to add a command which caused a short tone to be emitted by the TV speaker, as a warning in case of errors.

Here is a short machine code routine which performs the function which could be called BEEP.

```
START   LDA #37       ;set up SID
        STA 54271     ;registers
        LDA #100
        STA 54273
        LDA #15
        STA 54296
        LDA #54
        STA 54277
        LDA #168
        STA 54278
        LDA #33
        STA 54276
        LDA #122      ;store into
        STA 162       ;Jiffy clock
DELAY   LDA 162       ;loop until
        BPL DELAY     ;zero
QUIET   LDA #32       ;turn off the
        STA 54276     ;noise
        LDA #0
        STA 54296
        RTS
```

Even if you are not familiar with machine code you might recognise that the program simply sets up the necessary SID registers on channel one to make a tone, uses the jiffy clock to create a short pause and then turns the sound off.

To incorporate the new command into BASIC, this machine code must be preceded by some code to recognise the new command. In this example we

will use the # symbol as the new command, to avoid confusing the issue with subroutines to decode the characters in a command.

Here is the program which modifies the CHRGET routine and incorporates a 'BEEP' command into BASIC:

```
1 *=$C000
10 CHRGET=115
90 !
95 !
100          LDY #0          ;Modify the
110 INIT     LDA TABLE,Y     ;first six
120          STA CHRGET,Y    ;bytes of
130          INY             ;CHRGET
140          CPY #6          ;routine
150          BNE INIT
160          RTS
190 !
191 !
200 PROG     JSR READ        ;get char
205          CMP #35         ;is it a #?
210          BNE RETURN      ;no
220          JSR BEEP        ;yes! BEEP
225          JSR COPY        ;update $7A
230 RETURN   RTS             ;and $7B
290 !
291 !
300 READ     LDA $7A         ;copy $7A
301          STA R2+1        ;and $7B
302          LDA $7B         ;into new
303          STA R2+2        ;read
309          INC R2+1        ;routine &
310          BNE R2          ;get next
320          INC R2+2        ;character
330 R2       LDA $0800
340          RTS
390 !
391 !
500 COPY     INC $7A         ;copy of
```

```
510           BNE EXIT        ;start of
520           INC $7B         ;CHRGET
530 EXIT      RTS             ;routine
590 !
591 !
800 BEEP      LDA #37
805           STA 54271
810           LDA #100
815           STA 54273
820           LDA #15
825           STA 54296
830           LDA #54
835           STA 54277
840           LDA #168
845           STA 54278
850           LDA #33
855           STA 54276
860           LDA #122
865           STA 162
870 DELAY     LDA 162
875           BPL DELAY
880 QUIET     LDA #32
885           STA 54276
890           LDA #0
895           STA 54296
900           RTS
990 !
991 !
1000 TABLE    JSR PROG        ;new start
1010          JSR COPY        ;of CHRGET
```

Below is a BASIC loader for the machine code:

```
5     REM BASIC LOADER FOR BEEP
      COMMAND
10    FOR Z=49152 TO 49264
20    READ X:POKE Z,X:NEXT Z
20000 DATA 160,0,185,106,192,153,
      115,0,200,192,6,208,245,96,32,
      28,192
```

```
20010 DATA 201,35,208,6,32,57,192,
      32,50,192,96,165,122,141,47,19
      2,165
20020 DATA 123,141,48,192,238,47,
      192,208,3,238,48,192,173,97,8,
      96,230
20030 DATA 122,208,2,230,123,96,169,
      37,141,255,211,169,100,141,1,2
      12,169
20040 DATA 15,141,24,212,169,54,141,
      5,212,169,168,141,6,212,169,33
      ,141
20050 DATA 4,212,169,122,133,162,
      165,162,16,252,169,32,141,4,21
      2,169,0
20060 DATA 141,24,212,96,32,14,192,
      32,50,192,64
```

NOTE The code occupies some of the memory used by the graphics routines, which will be overwritten by this program. To start the program type:

```
SYS 49152
```

From now on every time a # is detected in the program, the BEEP routine will be called. The # can be treated like any other BASIC command with one exception – it must not be the first command on a line.

How the Program Works

Upon initialisation the program modifies the first six bytes of the CHRGET routine to read:

JSR $C00E

JSR $C032

The routine keeps its own pointer to the current byte of the BASIC program in the READ subroutine, and this is updated to keep up with that used by CHRGET. If a # symbol is detected, the BEEP subroutine is called, the CHRGET pointers are incremented to avoid reading the # twice and control is returned to the CHRGET routine. If the character isn't a # then control is passed back to CHRGET.

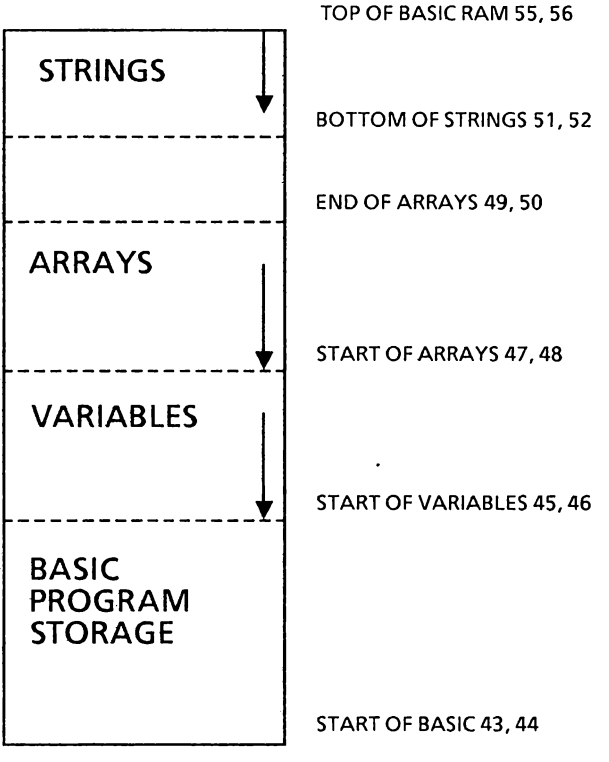
To add a number of commands to BASIC would involve storing them in a table, and preceding each with a #, then adding code between lines 210 and 220 to compare the characters after the # with the commands in the table. If a match was found then the appropriate subroutine would be called.

VARIABLE STORAGE IN BASIC

One of the things the BASIC interpreter must take care of is the handling of variables. It must decide where to store them and have an indication as to what type of variable they are. This section illustrates the various types of variable, the form in which they are stored in the 64 and their location in RAM. The diagram overleaf illustrates the arrangement of memory when a BASIC program is stored and the zero page locations which contain the start address of the various areas of memory.

SIMPLE VARIABLES

Simple variables are stored in memory starting immediately after the BASIC program, at the location pointed to by locations 45 and 46. Each variable occupies seven bytes of memory but these bytes are used differently by the different types of variable.



Memory Allocation for a BASIC program

Integer Variables

The diagram shows how the seven bytes are used by an integer variable.

VARIABLE NAME		VARIABLE VALUE		NULL	NULL	NULL
1st + 128	2nd + 128	High Byte	Low Byte	0	0	0

Format of an integer variable

Note that three bytes are unused, and so the use of integer variables doesn't result in a memory saving over floating point variables.

Floating Point Variables

These are stored as an exponent and a mantissa, so the number is stored in the form mantissa * 2↑ exponent.

The mantissa of a floating point variable is stored in packed Binary Coded Decimal form which gives 8 bit precision with the four bytes allocated. The first bit of the first byte of the mantissa is a sign bit.

VARIABLE NAME		VARIABLE VALUE				
1st	2nd	Expone--nt	Mantiss--a	Mantiss--a	Mantiss--a	Mantiss--a

Format of a floating point variable

String Variables

A string variable is stored in two parts: as an entry in the variable table as illustrated in the diagram below, and as the characters comprising the string, which are stored in a separate area of memory, pointed to by the entry in the variable table.

VARIABLE NAME		POINTER TO CHARACTERS			NULL	NULL
1st	2nd + 128	No of Chars	Low Byte	High Byte	0	0

Format of a string variable

Because the number of characters in the string is contained in one byte in the variable table entry

the maximum number of characters in a string is 255.

String variables may also be defined in a program by such statements as:

```
A$="HOW LONG IS A PIECE OF STRING"
```

In this case the entry for A$ in the variable table would point to a location within the BASIC program text.

Type in the following program exactly as it appears to see the various variable types and their storage mechanisms.

```
10    PRINT "START OF VARIABLES
      BEFORE=";:PRINT PEEK(45)+256*
      PEEK(46)
20    PRINT "START OF ARRAYS
      BEFORE=";:PRINT PEEK(47)+256*
      PEEK(48)
30    A% = 100
40    B = 1000
50    C$ = "CBM-64"
60    D$ = C$+C$
70    PRINT"START OF ARRAYS
      AFTER=";:PRINT PEEK(47)+256*
      PEEK(48)
```

If you run the program you will see that before the variables are created, the start of variables and the start of arrays are at the same location. However when the variables have been created by the program there is a difference of 28 bytes between the two - since each variable occupies 7 bytes and the program created 4 variables.

Type in the following command to examine the variable area:

```
FOR Z=2265 TO 2292 :PRINTZ,PEEK(Z):
NEXT Z
```

You should get a display like this:

2265	193	A (65 + 128) 1st char of name
2266	128	Blank 2nd character of name
2267	0	High byte of integer variable
2268	100	Low byte of integer variable
2269	0	Null
2270	0	Null
2271	0	Null
2272	66	B (66) 1st char of name
2273	0	Blank 2nd char of name
2274	138	128 + exponent
2275	122	1st byte of mantissa
2276	0	2nd byte of mantissa
2277	0	3rd byte of mantissa
2278	0	4th byte of mantissa
2279	67	C (67) first character of name
2280	128	Blank 2nd character of name
2281	6	Number of characters
2282	142	Low & high bytes of pointer
2283	8	to program area
2284	0	Null
2285	0	Null
2286	68	D(68) first character of name
2287	128	Blank 2nd character of name
2288	12	Number of characters
2289	244	Low & high bytes of pointer
2290	159	to string variable area
2291	0	Null
2292	0	Null

From the comments added to the list you should be able to see that the arrangement of data in the variable area corresponds to that decribed above.

One of the uses to which we can put this knowledge is in a short routine to dump the names of all the variables used in a program. Such a routine can be a valuable aid when developing programs, and could be made more useful by getting it to give the values assigned to each variable at the time of the dump. The following program will print a list of all variables used in a program.

```
1      REM VARIABLE DUMP
10000 Z1=PEEK(45)+256*PEEK(46)
10010 Z2=PEEK(47)+256*PEEK(48)-21
10020 FOR Z3=Z1 TO Z2 STEP 7
10030 Z4=PEEK(Z3):Z5=PEEK(Z3+1)
10040 IF Z4>127 AND Z5>127 THEN
      PRINT CHR$(Z4-128)CHR$(Z5-
      128);"%":GOTO 10070
10050 IF Z4<127 AND Z5<127 THEN
      PRINT CHR$(Z4)CHR$(Z5):GOTO
      10070
10060 PRINT CHR$(Z4)CHR$(Z5-128);"$"
10070 NEXT
```

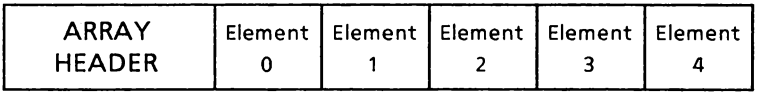
To use the program simply append it to your own program and type in direct mode `GOTO 10000` when you need a variable dump.

For optimum efficiency it should be written in machine code, and could then be called with a **SYS** whenever required during the debugging stage of your BASIC program.

Array Variables

All three types of variable may be stored in array form, in which case different methods are used for allocating memory. Arrays are stored immediately above simple variables, starting from the address pointed to by locations 47 and 48. The array type (floating point, integer or string) is indicated by the way in which the array name is stored – in exactly

the same way as with simple variables. The arrangement of an array in memory is illustrated below.

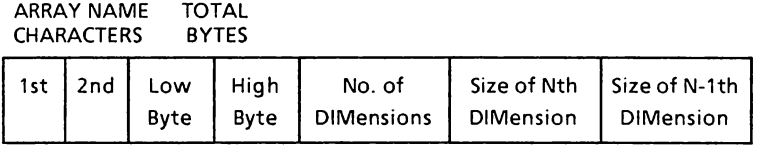


Format of an array

NOTE: Elements are stored in reverse order in string arrays.

The Array Header

The header format is the same regardless of the array type, and occupies seven bytes plus an extra two bytes for each dimension beyond 1.



Format of an array header

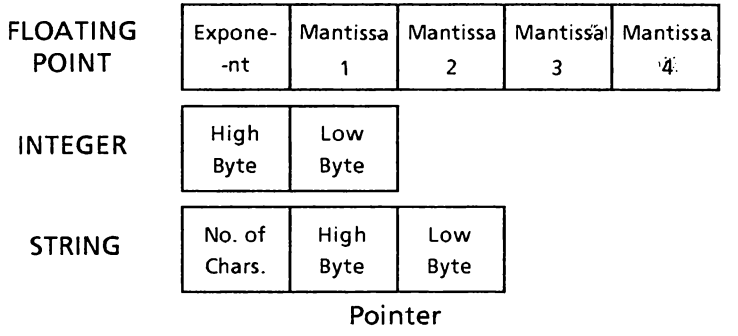
Bytes one and two contain the array name and bytes three and four store the amount of memory occupied by the array in low byte, high byte order. The fifth byte contains the number of dimensions in the array.

In a one dimensional array bytes six and seven contain the size of the array as specified in the **DIM** statement that created it. If no **DIM** statement exists then the default number is 10. For arrays with more than one dimension, the header is larger by two bytes per additional dimension and these two bytes contain the size of the extra dimension. The dimension bytes are stored in reverse order in

the header to that in which they appear in the **DIM** statement that created them.

Array Elements

The way in which array elements are arranged in memory is illustrated in the following diagram:



Format of array elements

Floating point array elements occupy 5 bytes, string elements occupy 3 bytes and integer array elements need only two bytes

The next program creates an array and the immediate mode commands following it display the area of memory in which it is stored.

```
10    DIM AB(10,20)

FOR Z=2067 TO 2085:PRINT Z,PEEK(Z):
NEXT Z
```

You should get a list like this:

2067	90	These seven bytes
2068	0	are the loop counter, Z,
2069	140	used to display
2070	1	the list.
2071	112	
2072	0	

2073	0	
2074	65	A The array name
2075	66	B
2076	140	The number of bytes used, low
2077	4	and high. 4*256+140=1164
2078	2	The number of dimensions
2079	0	Size of second dimension, high
2080	21	and low bytes. 0*256+21=21
2081	0	Size of the first dimension, high
2082	11	and low bytes. 0*256+11=11
2083	0	This is the start of the area of
2084	0	memory in which the array
2085	0	elements are stored

CHAPTER 2

MACHINE CODE ON THE 64

Although the 64 runs programs in BASIC, this is not the 'natural' language of the microprocessor (a 6510) at the heart of the computer. When you switch on the 64 it automatically begins running a very sophisticated set of programs which are written in the machine language of the 6510. It is these programs which interpret the BASIC commands, provide the screen editing facility and handle the cassette, disk drive and printer.

The machine language of a microprocessor is a set of instructions which can be interpreted directly by the electronics within the microchip. Each type of processor has its own instruction set – the 6510 is a variant of the well known 6502 processor and shares the same instructions.

If the processor can interpret machine language programs directly, why go to the trouble of providing a second language, BASIC, for the 64? The reason is that machine language instructions are less powerful than BASIC instructions, and programs are therefore less easy to write, as it may take several machine code instructions to perform the equivalent of one BASIC command.

When computers were first developed all programs for them were written in machine code, but it was soon realised that this made programming very tedious. "High-level" languages such as BASIC were developed to make programming easier by allowing the programmer to concentrate on the

problem to be solved without getting bogged down in the details of writing the code.

THE 6510 MICROPROCESSOR

The electronics of a microprocessor are extremely complicated, but from the machine code programmer's point of view the 6510 is a fairly simple device. The 6510 has three 8-bit data registers, the *accumulator*, and the *X* and *Y index registers*; and three control registers, the *processor status register*, the *stack pointer* and the *program counter* (which unlike the others is a 16-bit register). These registers are similar to storage locations in memory but are built into the microprocessor chip, and are used to hold the data being processed by the 6510.

Accumulator

This is the most used of all the registers. Almost all the data processed by the computer passes through this register. The accumulator is used in all calculations – addition, subtraction and logical operations.

X and Y Index Registers

The index registers are so called because they may be used in a variety of ways to index tables of data in the memory. They also play a useful role as loop counters and temporary storage registers.

Processor Status Register P

This is not so much a register as a collection of single bits which act as 'flags' to indicate various conditions of the processor. These flags are:

C — The Carry flag. Used in addition and subtraction operations.

Z — The Zero flag. Indicates that the result of the last operation was zero.

N — Indicates a negative result.

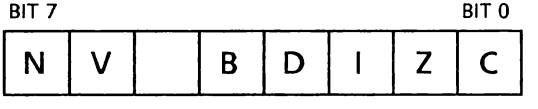
V — Indicates an overflow or underflow in signed arithmetic.

D — Sets Decimal mode.

I — Interrupt disable.

B — Break instruction just performed.

These seven flags form the processor status register, which is an 8-bit register thus:



Bit 5 is not used.

The 6510 has a number of instructions which test the states of these flags and cause the program to branch if a flag is in a certain state. These instructions correspond loosely to the **IF ... THEN ... GOTO** of BASIC, and play an important part in all machine code programs.

Program Counter PC

This 16-bit register stores the address in memory of the next instruction to be executed.

Stack Pointer S

This is an eight bit register used to index the memory area in which subroutine return addresses are stored.

THE 6510 INSTRUCTION SET

The instructions in the machine code instruction set include commands for moving data between the 64's memory and the three data registers, for performing arithmetic and logical operations on the contents of the accumulator, and for testing the results of these operations.

All 6510 instructions take the form of single byte numbers, but may be followed by one or two further bytes as an operand. For example,

```
169   1
```

means 'load the accumulator register with the number 1'. Instructions are usually expressed in hexadecimal as this is more convenient. The instruction given above would look like this in hexadecimal:

```
$A9   $01
```

The $ prefix indicates that the numbers are in hex.

Instructions may be entered in a number of ways. The simplest method is to **POKE** the numbers into store using a BASIC program. This method is unbelievably tedious and error-prone for all but the shortest of programs. For more complex programs an *assembler* should be used. This is a special program which allows you to enter and amend a sequence of machine code instructions using mnemonics to represent the instructions, and then translates these into the numbers which the

microprocessor can read. For example, the instruction above could be entered in the form:

```
LDA   #$01
```

where **LDA** means LoaD the Accumulator, and #$01 is the number to be loaded. The # is used to distinguish a number from an address in assembly language, while the $ symbol again indicates a hexadecimal number.

The instructions of the 6510 are described below.

Transferring Data

The first group of instructions move data between the 6510 registers and the memory.

LDA This instruction copies a byte of data from memory into the accumulator.

STA Copies the accumulator to memory.

For example:

```
LDA $0401

STA $0402
```

copies data from location $0401 (which in decimal is 1025) into the accumulator and then stores it in location $0402 (1026).

There are a number of different forms of the **LDA** and **STA** instructions, which differ in the way the memory is used. These different forms are called *addressing modes.*

Absolute Addressing Mode

Instructions using this addressing mode give the address of the memory location as a two byte number following the instruction code. For example:

```
LDA $0401
```

The code for the **LDA** instruction is $AD (or 173), so the instruction is coded as:

```
AD 01 04
```

(The low byte of the address is always given first.)

Similarly:

```
STA $0402
```

STA has the code $8D, so the instruction is:

```
8D 02 04
```

Immediate Addressing Mode

In this mode the data is read not from a specified memory location, but from the location after the program instruction. This is used to load the accumulator with known values. The # symbol is used to signify that the number following the instruction, called the *operand*, is to be interpreted as a number:

```
LDA #1
```

means load the accumulator with the number 1. The code for the instruction in $A9, so the full instruction would be:

```
A9 01
```

There is no corresponding **STA** instruction - it would be meaningless!

Zero Page Addressing Mode

Here the memory location indicated is in page zero of the memory (the first 256 bytes, $0000 to $00FF, for which the high byte of the address is $00).

```
LDA $7D
```

loads data from $007D. The code is $A5, so the full instruction is:

```
A5 7D
```

This is equivalent to the absolute addressed:

```
LDA $007D        AD 7D 00
```

but is faster in operation, and takes less storage space. Because zero-page addressing is faster it is useful to store frequently used data in page zero.

Other data transfer instructions, **LDX, LDY, STX** and **STY** transfer data to and from the X and Y index registers. These instructions have the following codes:

MODE	LDX	LDY	STX	STY
ABSOLUTE	AE	AC	8E	8C
IMMEDIATE	A2	A0	-	-
ZERO PAGE	A6	A4	86	84

Further instructions move data between the accumulator and the X and Y registers. These are single byte instructions with no operands:

	CODE	FUNCTION
TAX	AA	Copies Accumulator to X register
TAY	A8	Copies A to Y
TXA	8A	Copies X to A
TYA	9A	Copies Y to A

Arithmetic

The 6510 can perform addition and subtraction operations. Multiply and divide must be calculated by program, as instructions are not provided.

Addition is performed by the command **ADC**. This adds a number in memory to the number in the accumulator, storing the result in the accumulator. If the carry flag in the processor status register was set before the operation, 1 is added to the result. If the result is greater then 255, the least significant eight bits remain in the accumulator and the carry flag is set; otherwise it is cleared.

The use of the carry flag allows numbers of two bytes or more to be added together, with the carry operating in the same way as when you add two numbers with pencil and paper.

To add two single byte numbers stored in, say, $1000 and $1001:

```
CLC          Clear the carry flag.
LDA $1000    Load first number.
```

```
ADC $1001       Add second number.

STA $1002       Store result.
```

The result is stored in $1002. The two original numbers are unaffected by the operation.

Notice that the carry flag was cleared before the addition. This should always be done before an addition, as the flag may have been set by a previous operation, and this would give the wrong result for the addition. The code for **CLC** is $18.

Like **LDA** and **STA**, **ADC** can be used in several addressing modes:

Absolute Mode	ADC<address>	6D
Immediate	ADC#<number>	69
Zero Page	ADC<zero page address>	65

Larger numbers can be added in a similar way to that shown in the example above. Suppose there are two 3 byte numbers stored in locations $A1, $A2, $A3 and $A4, $A5, $A6, and the sum of these is to be stored in $B1, $B2, $B3. The method is as follows:

```
CLC         Clear carry flag
LDA $A1     Load lowest byte of first number
ADC $A4     Add lowest byte of second number
STA $B1     Store lowest byte of sum
LDA $A2     Load second byte of first number
ADC $A5     Add second byte of second number
STA $B2     Store second byte of sum
LDA $A3     Load highest byte of first number
ADC $A6     Add highest byte of second number
STA $B3     Store highest byte of sum
```

Just as in normal arithmetic, the lowest parts of the numbers (the least significant bytes) are added first.

Subtraction

ADC is complemented by a subtraction command **SBC**, which subtracts a number from that in the accumulator. This is used in a similar fashion to the **ADC** command, and again the carry flag is used when handling large numbers. There is an important difference – the carry flag must be *set* before the subtraction, and will be cleared to indicate a 'borrow' when the result of the subtraction is less than zero. If the flag is clear before subtraction, 1 is subtracted from the result. The different addressing modes are again available:

Absolute	SBC <address>	ED
Immediate	SBC# <number>	E9
Zero Page	SBC<zero page address>	E5

This example subtracts 1 from a two byte number stored in $0401, $0402:

```
SEC          Set carry = clear borrow
LDA $0401    Load low byte
SBC #1       Subtract 1
STA $0401    Store new low byte
LDA $0402    Load high byte
SBC #0       Subtract 0
STA $0402    Store result
```

The last 3 instructions are not as pointless as they may seem. If the low byte was zero to start with, subtracting 1 gives 255 ($FF) and clears the carry flag (sets 'borrow'). The last three instructions load the high byte of the original number and subtract

0, but if the carry flag has been cleared, 1 will be subtracted from the result. So, if locations $0401 and$0402 had originally contained the number $2900, the result would be $28FF.

Addition and subtraction can be performed only on numbers in the accumulator. There are no corresponding commands for numbers held in the X and Y registers.

Incrementing and Decrementing

You can add or subtract 1 to a number in memory or in the X or Y register using Increment and Decrement instructions:

	CODE	FUNCTION
INX	E8	Adds 1 to X register
DEX	CA	Subtracts 1 from X register
INY	C8	Adds 1 to Y
DEY	88	Subtracts 1 from Y

These are single byte instructions with no operand. The two instructions **INC** and **DEC** perform similar operations on numbers in the memory.

`INC <address>` Adds 1 to the contents of that address

`DEC <address>` Subtracts 1 from the contents of the address

INC and **DEC** do not alter the accumulator or the X and Y registers.

None of these increment and decrement instructions has any effect on the carry flag. The Z flag will be set if the result of any of these

operations is zero, and the N flag will be set if Bit 7 of the result is set. (Z and N are similarly affected by **ADC** and **SBC**.)

Branches

The 6510 has a number of conditional branch instructions which test a flag in the P register and cause the program to branch if the tested flag is set or clear. For example, the instruction **BEQ** causes a branch if the Z flag is set, that is if the result of the last operation performed was zero. **BNE** does the opposite: the program branches if Z is clear, after a non-zero result.

The instructions have single byte operands which give the branch destination relative to the current program position. If the operand is between 0 and 127 the jump is forwards; if the operand is between 128 and 255 the jump is backwards. An operand of 128 causes a jump to the instruction whose address is 128 less than the current position in the program, and the size of the jump gets less as the number increases to 255. So,

```
BNE 20
```

causes a jump forward of 20 bytes if the zero flag is set and:

```
BNE 230
```

causes a jump back of 26 bytes.

A vital point to remember is that the jump is not measured from the address of the branch instruction but from that of the next instruction. This is because the program counter is increased to point to the next instruction before the branch instruction is processed.

Branch instructions are indispensable in programs of any length, as they are the machine code equivalents of BASIC commands like **IF ... THEN ... GOTO**, allowing programs to make decisions and act accordingly. The instructions also provide a convenient way of creating loops. Consider the following:

```
    . . .
    . . .
    LDX #10        A2 0A
    DEX            CA
    BNE 253        D0 FD
    LDA $ABCD      AD CD AB
    . . .
    . . . etc.
```

This loop is repeated 10 times, until the Z flag is set as the value in the X register becomes zero, at which point the program continues. The loop in the example is literally a waste of time, doing nothing more than delay the program slightly. There could however be a number of instructions between the beginning of the loop and the **DEX** instruction.

An important point to remember is that the branch instruction must follow immediately after the instruction which creates the condition under test. Any instruction which modifies any of the 6510 data registers (and quite a few which don't) will modify the flags in the processor status register, so the flags are constantly changing. If there are any instructions between the one whose effect you wish to test and the test itself, the flags may change again before the test is performed. For example, in a loop ending:

```
. . .
. . .
DEX
```

```
LDA $1234
BNE . . .
```

the flags will be set by the **DEX** instruction, but immediately changed by the **LDA** instruction to indicate the nature of the number loaded into the accumulator.

The full list of branch instructions is:

	CODE	OPERATION
BEQ	F0	Branch on result = 0 (Z set)
BNE	D0	Branch on result<>0
BCS	90	Branch if carry set
BCC	B0	Branch if carry clear
BMI	30	Branch if negative (N set)
BPL	10	Branch if positive (N clear)
BVS	70	Branch if V set (overflow)
BVC	50	Branch if V clear

These instructions test the four flags Z, C, N and V.

The Z or zero flag indicates a result of zero, either in the accumulator, or the X or Y register, or if the **INC** or **DEC** instruction was used, in the memory location concerned.

The carry flag may be set or cleared by program. It is also altered by addition and subtraction instructions, and by the register shift instructions which are described later.

The N flag is a copy of Bit 7 of the register last altered. In signed arithmetic this bit indicates the

sign of the number – set for negative, clear for positive.

The V flag indicates an overflow in twos complement arithmetic. The flag may be cleared by the **CLV** instruction.

Comparisons

The three instructions **CMP**, **CPX** and **CPY** are used to compare registers with numbers in memory. These instructions do not alter either memory or the register, but the Z, C and N flags are altered to indicate the result of the comparison.

CMP compares the accumulator with another number. For example:

```
CMP #1
```

compares the contents of the accumulator with the number 1. The comparison leaves the flags in the state they would be in after setting the C flag and subtracting the number from the accumulator. That is:

C is set if A >= Number, otherwise C is clear.

Z is set if A = Number, otherwise Z is clear.

N is set if the operation (A–Number) would leave a 1 in Bit 7.

The instructions **CPX** and **CPY** are similar, except that the X or Y register is tested instead of the accumulator. All these instructions may test memory or numbers using the addressing modes described for **LDA**.

MODE	CMP	CPX	CPY
ABSOLUTE	CD	EC	CC
IMMEDIATE	C9	E0	C0
ZERO PAGE	C5	E4	C4

ADDRESSING MODES

Many of the 6510 instructions can take several forms, providing different ways for specifying the memory location to be used by the instruction. These different forms of the instructions are said to use different addressing modes, because they address the memory in different ways. We have already introduced three addressing modes: absolute addressing, zero page addressing and immediate addressing. These and the other modes are explained below.

Absolute Addressing

This mode uses a two byte operand after the instruction code. The two bytes are the low and high bytes of the address of the data in memory. Absolute addressing is indicated by the full address after the instruction:

```
LDA $ABCD
```

Zero Page Addressing

This mode uses a one-byte operand following the instruction code. This byte is the low part of an address in page zero ($0000 to $00FF). Instructions in this mode are written thus:

```
LDA $C7
```

Indexed Addressing

Indexed addressing uses the X or Y index register to modify the address given after the instructon, so that the address of the data is Operand+X or Operand+Y. For example:

Absolute:

```
LDA $ABCD
```

loads the contents of $ABCD

Absolute Indexed:

```
LDA $ABCD,X
```

loads the contents of $ABCD+X.

The Y index register can also be used in a similar manner.

As well as absolute addressing there are also zero page indexed addressing modes. For example:

```
LDA $AB,X
```

loads the accumulator with the contents of $00AB+X.

Indirect Addressing

In the indirect addressing modes the indicated memory location is not used to store the data, but holds the low byte of the address of another location where the data is to be placed. The high byte of the address is held in the next location. Only one 6510 instruction, the jump instruction **JMP**, can use simple indirect addressing, but many instructions use indexed forms of indirect addressing.

Indirect Indexed & Indexed Indirect Addressing

In indirect indexed addressing the single operand byte following the instruction indicates the first of a pair of zero page locations whose contents form a pointer to the target location. The contents of the Y index register are added to the pointer to find the final address.

Indexed indirect addressing uses the X register to index the zero page address in which the pointer is to be found. So:

Indirect Indexed:

`LDA (TABLE),Y`	Reads the zero-page locations TABLE and TABLE+1 and adds Y to the contents to produce the address of the data.

Indexed Indirect:

`LDA (TABLE,X)`	Adds X to the address TABLE to find the zero page location where the data pointer is held.

OTHER ADDRESSING MODES

Implied addressing

No address at all is specified, as in **CLC**, **RTS** etc.

Relative Addressing

The address is given as an offset from the current program address. This is the addressing mode used by the relative branch instructions. For example:

```
BCS 35
```

means branch to the program instruction at the address 35 bytes above the current address in the program counter.

Accumulator Addressing

The instruction acts only on the accumulator. Again no operand is used. This addressing mode is used only by the register shift instructions.

JUMPS AND SUBROUTINES

In addition to the relative branch instructions described above, there is an absolute jump instruction, **JMP**. This causes the program to jump to another location. There are two addressing modes:

Absolute:

`JMP $ABCD`	The program continues at the location specified in the next two bytes, in this case $ABCD.

Indirect:

`JMP ($ABCD)`	The program jumps to the address whose low byte is contained in location $ABCD, and whose high byte is contained in location $ABCE.

A similar instruction, **JSR**, is used to call subroutines. This may be used only in the absolute addressing mode, for which the instruction code is $20.

`JSR $C000`	Jumps to the subroutine at $C000.

Before jumping to the subroutine, the 6510 stores the current value of the program counter in a special area of memory called the *stack*, so that it may be restored on completion of the subroutine. The command **RTS** causes the return from subroutine; the 6510 reads the first two bytes on the stack and resets the program counter to that address.

The stack is held in page 1 of the 64's memory, locations $0100 to $01FF. The stack is designed to be used as a last-in first-out store, and the *stack pointer* register is used to indicate the first free location in the stack. When the 6510 is reset, the pointer is set to $FF (the high byte is always $01 and can be disregarded), and the pointer is decremented as the data is added by subroutine calls, and is incremented as data is removed by **RTS** commands, so that it always indicates the first free location, to which the next byte to be added will be written. This allows the nesting of subroutines; as subroutines are called the return addresses are added to the stack, which fills down from $01FF. Data is removed in reverse order, and the stack empties up to $01FF.

If the stack pointer should reach zero and a further subroutine call is made, the pointer would return to $1FF, and the oldest items in the stack would be overwritten, which would cause havoc! This is unlikely to happen except in very complicated programs; most of the time you can forget about the working of the stack and let it look after itself.

Data may be written to the stack by a program, using the instruction **PHA**, which 'pushes' the contents of the accumulator onto the stack. This can be useful if you want to put a number on one side for a moment while performing another

calculation. Numbers are pulled from the stack by the instruction **PLA.**

It is also possible to store the processor status register P on the stack and recall it, using **PHP** and **PLP.**

	CODE	OPERATION
PHA	48	A to STACK
PLA	68	STACK to A
PHP	08	P to STACK
PLP	28	STACK to P

The stack pointer is automatically adjusted by the 6510 when these instructions are used. Don't forget to take things off the stack in the reverse order to that in which you put them on.

Be careful using these instructions. Remember that they all use the same stack as is used to store subroutine return addresses. This means that you must be careful not to execute a **RTS** instruction between writing data to the stack and reading it back, or the program will return to the wrong place. You can of course call a subroutine after writing the data; it will return without any problem. What you must not do is return from a subroutine which has pushed data onto the stack before taking the data back off the stack.

Two further instructions, **TSX** and **TXS**, allow you to modify the stack pointer.

TSX BA Copies the stack pointer to the X register

TXS	9A	Copies the X register to the stack pointer

This means you could, if you felt confident, maintain two or more stacks, but this is not advisable. There would be a slight speed advantage in using an area of page 1 as a table, indexed by the stack pointer, but it would be both easier and less hazardous to use indirect indexed addressing in another part of the memory.

INTERRUPTS

A microprocessor in a computer such as the 64 has several jobs to do. As well as running the BASIC interpreter program, the screen must be managed, the keyboard checked for keypresses, and many other routine operations performed. The ideal microprocessor would be able to run many separate programs at the same time, but such a processor does not exist. Instead, the 6510 has a facility which allows programs to be interrupted while another program is run and then restarted at the point at which they were stopped.

Two of the 40 pins on the 6510 chip are *interrupt request* pins. Peripheral devices applying signals to one or other of these pins will stop the 6510 in the middle of whatever it is doing, and divert it to another piece of program. The 6510 will be returned to the original program by an instruction at the end of the interrupt program. The two pins are similar in use, the difference being that the IRQ (interrupt request) pin is ignored if the I bit of the processor status register has been set by an SEI instruction, whereas the NMI (non-maskable interrupt) pin can not be ignored.

When a suitable signal is applied to the IRQ pin the 6510 finishes the current instruction, stores the P register and the program counter on the stack, sets

the interrupt disable flag (I) in the P register and jumps to the program at the address held in locations $FFFE and $FFFF. The processor continues to execute the program from this point until a **RTI** instruction is reached. The **RTI** (return from interrupt) acts in a similar way to **RTS**, but restores the P register from the stack as well as the program counter.

Interrupts in response to the NMI pin are similar, except that the start address for the interrupt program is held in locations $FFFA and $FFFB.

The interrupt feature allows the effects of several programs running at once. The 64 has a *clock* circuit which interrupts the 6510 every 1/60 second to call the keyboard scanning routine. This checks the keyboard and puts the ASCII code of any key held down into the keyboard buffer before **RTI**ing. The effect of this is that the rest of the BASIC interpreting routines do not need any complex subroutines to read the keys; they just look in the keyboard buffer (this is what the BASIC **GET** command does).

Interrupts are discussed further in Chapter 5, which shows how interrupt programs may be used to modify the display.

LOGIC

The two logical operations **AND** and **OR**, familiar in BASIC, are also available in machine code, as is a third, the Exclusive OR. The Exclusive OR differs from OR in that 1 OR 1 gives 1, but 1 EOR 1 gives 0. The mnemonics for the three instructions are **AND**, **ORA** and **EOR**.

The truth tables for these operations are:

A	B	A OR B	A EOR B	A AND B
0	0	0	0	0
0	1	1	1	0
1	0	1	1	0
1	1	1	0	1

The instructions all act on the accumulator.

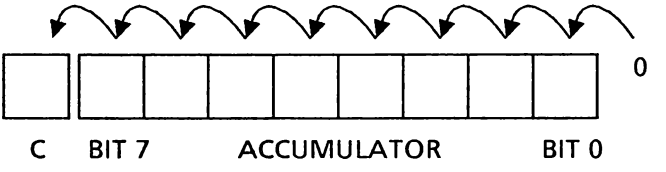
Logical instructions are useful for testing and modifying selected bits within a byte without altering the rest. For example the **AND** instruction may be used to *mask* a part of a byte. To inspect the first four bits of location $1234, the data would be loaded into the accumulator and ANDed with the number 15 (15 in binary is 00001111). As the **AND** operation only sets the bits which are set in both the numbers in the operation, the top four bits of the result will be zero, and the lower four bits will be a copy of the lower four bits of location $1234. The instruction sequence would be:

```
LDA $1234
AND #15
```

SHIFTS

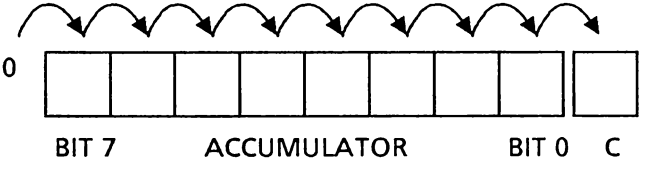
There are four shift instructions available which shift or rotate the bits in the accumulator or memory.

ASL Shifts the accumulator to the left. A zero is placed in Bit 0 and Bit 7 is moved to the carry flag. This is equivalent to multiplying the accumulator by two.



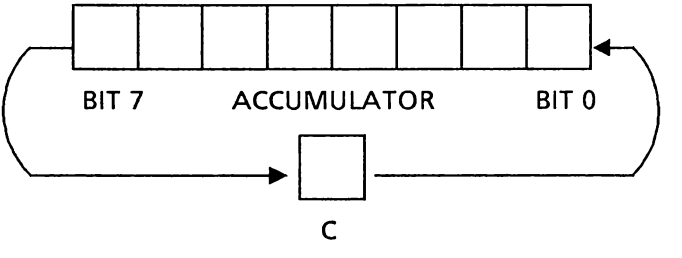
The ASL operation

LSR Shifts the accumulator to the right. A zero is placed in Bit 7, and Bit 0 it moved to the carry flag. This is equivalent to dividing the number in the accumulator by 2.



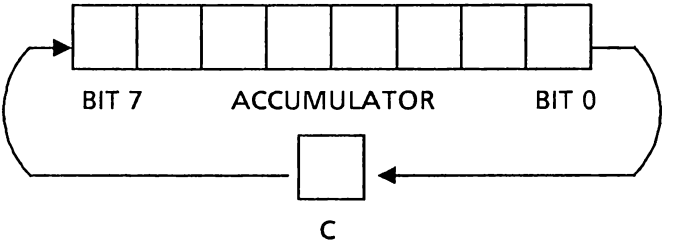
The LSR operation

ROL Rotates the number in the accumulator to the left. The carry flag is copied to Bit 0, and Bit 7 is moved to the carry flag.



The ROL operation

ROR Rotates the accumulator to the right.



The ROR operation

USING MACHINE CODE WITH BASIC

Machine code programs are called from BASIC by the **SYS** command. This is similar in effect to the **GOSUB** command except that the subroutine called is a machine code routine at the address specified in the **SYS** command. Control is returned to BASIC when the routine is completed by the machine code instruction **RTS**. The **SYS** command is followed by the address of the start of the machine code program. For example:

```
SYS 64738
```

calls the routine in ROM which resets the 64 when it is switched on. *Don't* try this if you have a valuable program loaded – it will be erased.

Storing Machine Code Programs

The most convenient place to store most machine code programs is in the 4k block of memory between 49152 ($C000) and 52247 ($CFFF) which is not used by the 64 in the execution of BASIC programs.

If your program is longer than 4k it may be placed lower down in memory in the area usually reserved for BASIC programs. This area normally extends

from 2048 to 40759 but you can reserve a part of it for machine code programs by altering the 'top of memory' pointer in locations 55 and 56.

To use this method, work out the new top of memory you need by subtracting the length of your machine code program from 40759, and **POKE** the low byte of that address into location 55 and the high byte into location 56. Finally reset the pointers by typing:

```
CLR
```

The memory above the new 'top of memory' is now protected from being overwritten by BASIC variables.

The Kernal Routines

Many of the subroutines present in the 64's ROM can be used in your machine code programs. Some of the most accessible and well documented form part of the operating system called the kernal.

To enable you to make use of these routines, a table of their start addresses is kept in memory - a feature shared by all Commodore machines. Calling each routine involves a **JSR** to the appropriate part of the table. By arranging the table in this way, programs written on one Commodore machine may be more easily translated to run on another. An example of the use of the kernal routines to send output to a printer may be found in Chapter 10, and a list of the routines and their functions is given in Appendix 6.

Learning to Write Machine Code

This chapter is not intended to teach you all there is to know about machine code programming,

rather it introduces the fundamentals of the language. If you wish to study the subject further, there are a number of books aboout the 6502/6510 microprocessors which you may find helpful.

We have written a number of machine code programs for this book, and if you study these you will soon get a feel for the language, and realise that despite first appearances, machine code programming is not really very difficult.

CHAPTER 3

BIT-MAPPED GRAPHICS - PART 1

As well as the normal text display, the 64 can produce a high resolution bit-mapped display in which the pixels correspond to individual bits in a defined area of memory. There are two alternative bit-mapped display modes: standard bit-mapped mode, which has a screen resolution of 320 pixels by 200 pixels in two colours; and multicolour bit-mapped mode which has a lower resolution of 160 by 200 pixels, but in four colours.

The graphics modes are controlled by the VIC-II chip. Bit 5 of the control register at location 53265 selects bit-map mode: if the bit is set, bit-mapped mode is selected; if the bit is clear the 64 display is in text mode. To select multicolour bit-mapped mode, Bit 4 of the second VIC control register at location 53270 must also be set.

Display Memory

When you enable bit-mapped mode, you must also decide where in memory the screen information will be stored. Bit mapped displays need nearly 9k of memory, 8000 bytes for the picture data and 1000 for the text screen memory, which in the bit-mapped modes is used to hold colour information. The VIC chip can only address 16k of memory at a time, so the two blocks of memory needed for bit-mapped displays must lie within one of the four 16k *banks* or sections which make up the 64k of total memory space in the 64.

Within the 16k bank, the 1k of text screen memory may be set to begin at any location whose address is

a multiple of 1024, but the 8k bit-map display memory area must begin either at the beginning of the bank, or at the mid point, 8k above the start. These limitations mean that the bit-map screen memory can not be put in the obvious place at the top of the user RAM (from 32k to 40k), as there is no more RAM in this bank into which the text screen memory could be placed. The best place for the bit-map display memory is therefore at the top of the second bank, from 24576 to 32575, with the text screen memory beginning at 23k (23552). This leaves about 21k of memory free for BASIC programs, which should be enough for most purposes.

The bank which the VIC chip addresses is set by two **POKE** commands. First the lowest two bits of the data direction register of the CIA interface chip must be set to 1 by:

```
POKE 56578, PEEK(56578) OR 3
```

and the bank number must be **POKE**d to the lowest two bits of the interface port at 56576:

```
POKE 56576, (PEEK(56576)AND252) OR B
```

where B is the bank number given by the table:

B	BANK	Starting Location
3	0 to 16k	0
2	16k to 32k	16384
1	32k to 48k	32768
0	48k to 64k	49152

Screen Memory Positioning

The position within the 16k bank of the bit-mapped screen is controlled by Bit 3 of location 53272. If the bit is zero the bit-mapped display is placed at the beginning of the bank. If the bit is 1 the screen memory begins at the mid-point of the bank.

The position of the text screen within the bank is controlled by the four highest bits of location 53272. The value of these bits is the number of 1k units by which the start of the screen memory is offset from the beginning of the 16k bank.

To set the bit-map display to begin at 24k, which is the mid-point of the 16k-32k bank, Bit 3 of 53272 must be set. To place the text screen memory at 23k, the four most significant bytes must be set to 23-16, which is 7. The value to be **POKE**d into location 53272 is therefore 7*16 + 8, which is 120.

So, to set the 64 to the bit-mapped display mode with the display at 24k, we need the following short program:

```
5     REM SELECT BANK 16-32K
10    POKE 56578, PEEK(56578) OR 3
20    POKE 56576, (PEEK(56576) AND
      252) OR 2
30    POKE 53265, PEEK(53265) OR
      32:REM SET BIT MAPPED MODE ON
40    POKE 53272, 120:REM SET MEMORY
      POINTERS
```

Restoring the Text Display

To restore the normal display manually, you can hold down RUN/STOP and tap RESTORE. Restoring the text display by program is a matter of returning all the registers to their normal values – clearing Bit 5 in location 53265 and resetting the

address pointers. Add the following lines to the previous program:

```
100   GET K$: IF K$="" THEN 100
200   POKE 53265, PEEK(53265) AND 23
210   POKE 53272, 21
220   POKE 56578, PEEK(56578) OR 3
230   POKE 56576, PEEK(56576) OR 3
```

When run, the program will set up a high-resolution display and wait until you press a key, after which the normal display will be restored.

Clearing the Screen

The display you see when you run the program will be filled with random blocks of colour. These correspond to the data with which the memory happened to be filled when you set bit-mapped mode. To clear the screen, you must clear the memory by **POKE**ing it with zeros.

Colour

The colour of a bit-mapped display is controlled by the numbers in the text screen memory. Each number in this memory area controls the colour of an 8x8 block of pixels on the high resolution display. The 'foreground' colour – the colour displayed for a bit set to 1 in display memory – is controlled by the four highest bits of the screen memory, while the background colour depends on the four lowest bits. For example, to set a foreground colour of red (colour 2) and a white background (colour 1), each location in the text memory must be set to 16*2 + 1, which is 33.

Add the following two lines to the program. Line 50 sets the colours, and line 60 clears the screen memory.

```
50      FOR L=23552 TO 24551: POKE L,
        33: NEXT L
60      FOR M=24576 TO 32575: POKE M,
        0: NEXT
```

The process of setting the colours and clearing the screen takes around 35 seconds - far from instantaneous! The only way of speeding up the screen clearing is to use a machine code program. A program to set the screen mode and clear the screen is described later, but first a few words about multicolour bit-mapped mode.

MULTICOLOUR BIT-MAPPED MODE

Multicolour bit-mapped mode is similar to standard bit-mapped mode, except that the screen resolution is reduced, and that each pixel may take one of four colours. The two extra colours are controlled by the colour memory (always located from 55296 to 56295) and the background colour stored in location 53281. Each pixel is controlled by two bits of data in the display memory as follows:

COLOUR	BIT PATTERN	COLOUR DISPLAYED
0	00	Background - location 53281
1	01	Upper four bits of screen memory
2	10	Lower four bits of screen memory
3	11	Colour memory

To change from standard bit-mapped mode to multicolour bit-mapped mode, Bit 4 of location 53270 must be set, by the command:

```
POKE 53270, PEEK(53270) OR 16
```

The multicolour bit must be cleared on returning to text mode by:

```
POKE 53270, PEEK(53270) AND 239
```

USING MACHINE CODE WITH GRAPHICS

Because of the speed problems using BASIC to control the graphics, machine code can be very useful. Included in this book are a number of machine code programs to speed up graphics functions, and the first of these sets bit-mapped mode (BMM) or multicolour bit-mapped mode (MCBMM) and clears the bit-mapped screen.

If you have an assembler and are familiar with machine code you can enter the machine code as it is given in this chapter and Chapter 4. Alternatively, you can type in the BASIC loader programs in Appendix 7 which you can use to load the machine code into your 64. These programs comprise a series of **DATA** statements which form the machine code program and a short routine to **READ** the **DATA** and **POKE** it into memory.

```
 10 !  **********************
 20 !  *** BIT-MAP MODES ***
 30 !  **********************
 40 !
 50 * = $C000              Starts at 49152
 60 !
 70 !
 80 COL1 = 686             Colour 1
 90 COL2 = 687             Colour 2 (Background
                           in Standard BMM)
100 COL3 = 688             Colour 3 (MC BMM
                           only)
110 MCBM = 689             1 = BMM, 0 = MC
                           BMM
120 !
130 !
```

```
140 !
160 BMM     LDA #1       Start for BMM
170         STA MCBM
180         BNE CLEAR
190 !
210 MCBMM   LDA #0       Start for MCBMM
220         STA MCBM
230 !
240 ! START BY CLEARING BMM SCREEN
245 !
250 CLEAR   LDA #$60     Wipe 64x256 bytes
                         starting at $6000
260         STA 252
270         LDA #0
280         STA 251
290         LDX #64
300         JSR WIPE
310 ! SET COLOURS
320 COLOUR LDA #92       Put COL1 and COL2
                         into screen mem
330         STA 252
340         LDA #0
350         STA 251
360         LDA COL1     Multiply COL1 by 16
370         ASL A
380         ASL A
390         ASL A
400         ASL A
410         ORA COL2     Add COL2
420         LDX #8
430         JSR WIPE
440 !
450         LDA MCBM
460         BNE HIRES
470         LDA #$D8     If MC, put COL3 into
                         colour memory
480         STA 252
490         LDA #0
500         STA 251
510         LDX #8
520         LDA COL3
```

```
530         JSR WIPE
540 ! NOW SET POINTERS FOR BM MODE
545 !
550 HIRES   LDA 56578   Set bank 2
560         ORA #3
570         STA 56578
580         LDA 56576
590         AND #252
600         ORA #2
610         STA 56576
620         LDA 53265   Set BMM on
630         ORA #32
640         STA 53265
650         LDA #120    Set screen position
660         STA 53272
670 !
680         LDA MCBM
690         BNE BMMEND
700         LDA 53270   If MC, set MC mode
710         ORA #16
720         STA 53270
730 !
740 !
750         BMMEND RTS
760 !
770 !
780 !
790 ! SWITCH TO TEXT MODE
795 !
800 LORES   LDA 56578   Reset all pointers for
                        text
810         ORA #3      Restore Bank 3
820         STA 56578
830         LDA 56576
840         AND #252
850         ORA #3
860         STA 56576
870         LDA 53265   Clear BMM bit
880         AND #223
890         STA 53265
900         LDA 53270   Clear MC bit
```

```
 910          AND #239
 920          STA 53270
 930          LDA #21      Reset pointers
 940          STA 53272
 950          RTS
 955 !
 960 ! WIPE OVER BLOCK WITH
     CHARACTER IN ACCUMULATOR
 965 !
 970 WIPE     LDY #127     Subroutine to fill
                           memory with
                           character in
                           accumulator
 980          F1 STA (251),Y
 990          DEY
1000          BPL F1
1010          PHA
1020          CLC
1030          LDA 251
1040          ADC #128
1050          STA 251
1060          LDA #0
1070          ADC 252
1080          STA 252
1090          PLA
1100          DEX
1110          BNE WIPE
1120          RTS
1130          END
```

The exclamation marks in the listing are used by this assembler to indicate comments, and do not form part of the program.

To use the machine code routine to set bit-mapped mode, POKE the foreground colour into location 686 and the background colour into location 687, and call the routine with the command:

```
SYS 49152
```

To return to text mode, use:

```
SYS 49266
```

To set multicolour bit-mapped mode, store the number of the third colour in location 688, and start the routine at line 210 of the listing using the command:

```
SYS 49159
```

Other call addresses allow you to switch from text to bit-mapped mode without clearing the screen, to clear the screen, or to reset the colours. The full list of entry points is:

BMM	49152	Clear, set BMM and colours
MCBMM	49159	Clear, set MCBMM and colours
CLEAR	49164	Clear BM screen and reset colours
COLOUR	49177	Set colours
HIRES	49221	Set to BM or MCBMM
LORES	49266	Return to text mode

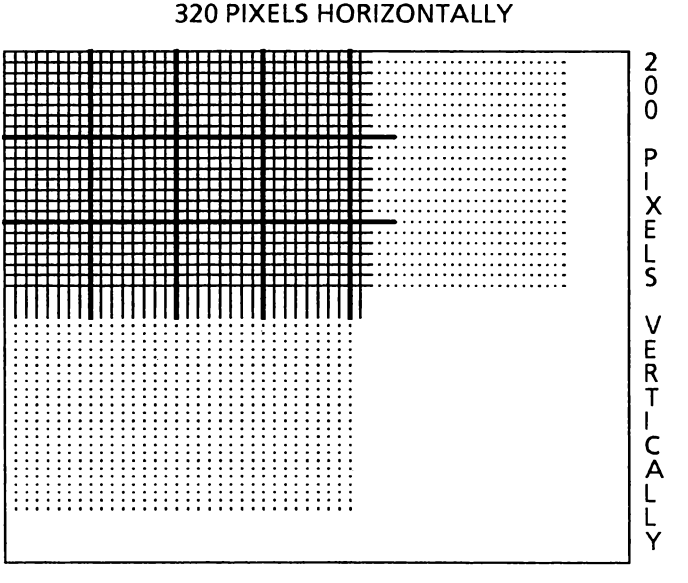
NOTE: Because these routines place the bit-mapped screen in the middle of the BASIC program area, at 24k, there is a risk that the display may be corrupted by the BASIC or by program variables. The bit-mapped screen can be protected from BASIC by resetting the top of BASIC memory thus:

```
POKE 55,0: POKE 56, 92: CLR
```

Which sets the top of memory to 23732 and thus prevents any BASIC programs or variables overwriting the colour memory. A line such as this should be the first in every program which uses bit-mapped graphics.

PLOTTING POINTS

The bit-mapped screen is arranged as a grid of 200 rows of 320 pixels (or 160 pixels in multicolour mode). Each byte in the screen memory represents a row of 8 pixels.



However, the layout of the memory is a little complicated. The top line (Row 0) of the display is mapped onto the display memory like this:

0	8	16	24	.	.	.	....	312
1	9	17	25	.	.	.	....	313
2	10	18	26	.	.	.	....	314

```
3   .   .   .   .   .   .   ....  315
4   .   .   .   .   .   .   ....  316
5   .   .   .   .   .   .   ....  317
6   .   .   .   .   .   .   ....  318
7   .   .   .   .   .   .   ....  319
```

The bytes are laid out in blocks of eight, corresponding to the characters of a text display. This layout means that plotting points on the bit-mapped screen requires some calculation to convert X and Y co-ordinates to byte addresses. The following formulae can be used to set and clear pixels (the value BASE represents the address of the start of the bit-mapped screen memory).

```
ROW  = INT(Y/8)
CHAR = INT(X/8)
LINE = Y AND 7
BYTE = ROW*320 + CHAR*8 + LINE + BASE
BIT  = 7-(X AND 7)
```

The pixel is set by:

```
POKE BYTE, PEEK(BYTE) OR 2↑BIT
```

and cleared by:

```
POKE BYTE, PEEK(BYTE)AND(255-2↑BIT)
```

The second program in our machine code graphics package plots points on the bit-mapped screen.

```
 10 !  ************
 20 !  *** PLOT ***
 30 !  ************
 40 !
 50 !
 60 * = $C0B8          Starts at 49366
 80 !
 90 !
100 BYTELO = 251
```

```
110 BYTEHI = 252
120 T1     = 253
130 T2     = 254
140 XLO    = 679
150 XHI    = 680
160 Y      = 681
170 !
180 COLNO  = 685
190 MCBM   = 689
200 BASE   = 24576
220 !
230 !
240 !
250 PLOT   LDA Y        Find ROW = Y/8
260        LSR A
270        LSR A
280        LSR A
290        STA ROW
300 !
310        LDA XHI      Find CHAR
320        LSR A
330        LDA XLO
340        ROR A
350        LSR A
360        LDX MCBM     If MC mode then
                        CHAR = X/4
370        BEQ PLOT1
380        LSR A        If not MC mode CHAR
                        = X/8
390 PLOT1  STA CHAR
400
410        LDA Y        LINE = Y AND 7
420        AND #7
430        STA LINE
440 !
450        LDA ROW      ROW = ROW*64
460        STA T1
470        LDA #0
480        STA T2
490        LDX #6
500 P1     JSR TIMES2
```

```
510         DEX
520         BNE P1
530         LDA T2
540         STA BYTEHI
550         LDA T1
560         STA BYTELO
570 !
580         JSR TIMES2  Add ROW * 256
590         JSR TIMES2
600         CLC
610         LDA T1
620         ADC BYTELO
630         STA BYTELO
640         LDA T2
650         ADC BYTEHI
660         STA BYTEHI  BYTE now holds
                        ROW*320
670 !
680         LDA #0      Find CHAR*8
690         STA T2
700         LDA CHAR
710         STA T1
720         JSR TIMES2
730         JSR TIMES2
740         JSR TIMES2
750         CLC
760         LDA T1      Add CHAR*8 to
                        BYTE
770         ADC BYTELO
780         STA BYTELO
790         LDA T2
800         ADC BYTEHI
810         STA BYTEHI  BYTE now holds
                        ROW*320 + CHAR*8
820 !
830         CLC         Add LINE to BYTE
840         LDA LINE
850         ADC BYTELO
860         STA BYTELO
870         LDA #0
880         ADC BYTEHI
```

```
 890          STA BYTEHI  BYTE holds
                          ROW*320 + CHAR*8
                          + LINE
 900 !
 910          CLC         Add screen start
                          address
 920          LDA #<BASE  Low byte of BASE
 930          ADC BYTELO
 940          STA BYTELO
 950          LDA #>BASE  High byte of BASE
 960          ADC BYTEHI
 970          STA BYTEHI
 975
 980 ! BYTE = BASE + ROW*320 +
     CHAR*8 + LINE
 985
 990          LDA MCBM    Branch if MC mode
                          selected
1000          BEQ MCPLOT
1010 !
1020 !
1040 !
1050 BMPLOT   LDA XLO     Normal mode PLOT
1060          AND #7
1070          STA BITT    BITT = X AND 7
1080          SEC
1090          LDA #7
1100          SBC BITT
1110          STA BITT    BITT=7-(X AND 7)
1120 !
1130          CLC         If BITT < > 0 then
                          find 2↑BITT
1140          LDA #1
1150          LDX BITT
1160          BEQ P3
1170 P2       ASL A
1180          DEX
1190          BNE P2      Accumulator now
                          holds 2↑BITT
1200 !
1210 P3       LDY #0
```

```
1220 !
1230          LDX COLNO    Test for Plot or Unplot
1240          BEQ UNPLOT   Branch if COLNO=0
1250          ORA (BYTELO),Y
1260          STA (BYTELO),Y  Plot the pixel
1270 !
1280          RTS
1290 !
1310 UNPLOT   EOR #$FF     Clear the pixel
1320          AND (BYTELO),Y
1330          STA (BYTELO),Y
1340          RTS          Standard mode plot
                           ends here
1350 !
1360 !
1380 !
1390 MCPLOT   LDA XLO      Plot in MC mode
1400          AND #3
1410          STA BITT
1420          SEC
1430          LDA #3
1440          SBC BITT
1450          ASL A
1460          STA BITT     BITT=2*(3-(X AND
                           3)
1470
1480          LDY #0       Move colour number to
1490          LDA COLNO    correct position in byte
1500          AND #3
1510          LDX BITT
1515          BEQ MCP2
1520 MCP1     ASL A
1530          DEX
1540          BNE MCP1
1550 MCP2     STA COLS     COLS = COLNO *
                           2↑BITT
1560 !
1570          LDA #%11111100  Set up mask
                              for pixel
1580          LDX BITT
1585          BEQ MCP4
```

```
1590          SEC
1600 MCP3     ROL A          Rotate mask to fit reqd
                             pixel
1610          DEX
1620          BNE MCP3
1630 !
1640 MCP4     AND (BYTELO),Y  Mask out bits
                              of pixel
1650          ORA COLS        Set new pixel
                              state
1660          STA (BYTELO),Y  Store the byte
1665 !
1670          RTS             End of MC plot
1680 !
1690 !
1700 !  MULTIPLY (T1,T2) BY 2
1710 !
1720 TIMES2 LDA #0
1730          ASL T2
1740          ASL T1
1750          ADC T2
1760          STA T2
1770          RTS
1780 !
1790 !
1800 ROW      NOP
1810 CHAR     NOP
1820 LINE     NOP
1830 BITT     NOP            Called BITT because
1840 COLS     NOP            BIT is a 6510
1850 !                       instruction
1860 !
1870 END
```

To use the program, **POKE** the low byte of the X co-ordinate into location 679, the high byte into location 680, and Y into location 681 (these locations are labelled XLO, XHI and Y in the program above) using the commands:

```
POKE 679, X AND 255
```

```
POKE 680, X/256
POKE 681, Y
```

Note that the program does not perform any checks to make sure that the co-ordinates are within the screen limits. Using overlarge values of X or Y is unlikely to have any ill effects other than spoiling the appearance of the display, but you should design your BASIC programs to avoid using incorrect values.

Location 685 (called COLNO in the program) is used to select the colour in which the pixel is plotted. In standard two colour mode, **POKE** 685 with 1 to plot in the foreground colour, or with zero to plot in the background colour (to erase a pixel). In multicolour mode, the colours are numbered from 0 to 3, as shown in the table on page 56.

The PLOT routine begins at location 49336, and so may be run by the command:

```
SYS 49336.
```

CHAPTER 4

BIT MAPPED GRAPHICS - PART 2

In this chapter we look further at the uses and applications of bit-mapped graphics, and introduce line drawing and block fill routines.

DRAWING LINES

Straight lines may be drawn using the equation:

```
Y=M*X+C
```

in which M represents the gradient of the line, and C is a constant. To find the equation of the straight line between the two points X1,Y1 and X2,Y2, we can perform the following calculations:

```
M = DY/DX = (Y2-Y1)/(X2-X1)
C = Y1 - M*X1
```

and then plot the line with:

```
FOR X=X1 TO X2: Y=M*X+C: JSR (PLOT)
```

with a suitable subroutine to plot the pixels.

If you try plotting a few lines by this method you will find that the lines appear broken if Y2-Y1 is greater than X2-X1. If this is so, reverse the algorithm to loop from Y1 to Y2 and calculate the values of X.

The third program in the machine code graphics package draws lines in this manner between two points on the screen:

```
100 !  ************
110 !  *** DRAW ***
120 !  ************
130 !
140 !
150 * = $C1C8              Starts at 49608
160 !
170 !
190 !
200 DRAW   LDA X2L         Take copy of line end
                           co-ordinates
210        STA XTL
220        LDA X2H
230        STA XTH
240        LDA Y
250        STA YT
260 !
270        LDA #0
280        STA NEG
290        SEC             DY = ABS(Y2-Y)
300        LDA Y2
310        SBC Y
320        BCS DRAW1
330        LDA NEG         If Y2 <Y then set NEG
340        EOR #1
350        STA NEG
360        SEC
370        LDA Y
380        SBC Y2
390 DRAW1  STA DY
400 !
410        SEC             DX = ABS (X2-X)
420        LDA X2L
430        SBC XL
440        STA DXL
```

```
450          LDA X2H
460          SBC XH
470          BCS DRAW2
480          LDA NEG        If X2<X then switch
                            NEG
490          EOR #1
500          STA NEG
510          SEC
520          LDA XL
530          SBC X2L
540          STA DXL
550          LDA XH
560          SBC X2H
570 DRAW2    STA DXH
580 !
590          BNE DRAW2A
600          LDA DXL
610          BNE DRAW2A
620          LDA #1         If DX=0 then set
                            NEG=1
630          STA NEG
640 !
650 DRAW2A   LDA DY         If DY=0 then set
                            NEG=1
660          BNE DRAW2B
670          LDA #1
680          STA NEG
690 !
700 DRAW2B   LDA DXH        If DX>=DY then
                            SHALLOW else
                            STEEP
710          BNE SHALLOW
720          LDA DXL
730          CMP DY
740          BCS SHALLOW
750          JMP STEEP
760 !
770 SHALLOW  SEC            Draw shallow lines
                            with DY/DX<=1
780          LDA X2L        If X2<X1 then swap
                            co-ordinates
```

```
 790          SBC XL
 800          LDA X2H
 810          SBC XH
 820          BCS SHAL1
 830          JSR SWAP
 835 !
 840 SHAL1    LDA DY        Find gradient
                            M=DY/DX
 850          STA DIVLO
 860          LDA #0
 870          STA DIVHI
 880          LDA DXL
 890          STA D1
 900          LDA DXH
 910          STA D2
 920          JSR DIVIDE
 930 !
 940          LDA XL        Calculate C
 950          STA MXL       First find X*M
 960          LDA XH
 970          STA MXH
 980          JSR MULT
 990          STA C
1000          LDA P4
1010          STA CH
1020          LDX NEG
1030          BEQ DRAW3     If M +ve then branch
1040          CLC           C=M*X+Y
1050          LDA C
1060          ADC Y
1070          STA C
1080          LDA CH
1090          ADC #0
1100          STA CH
1110          JMP SHLOOP
1120 DRAW3    LDA Y         C=Y-(M*X)
1130          SBC C
1140          STA C
1150          LDA #0
1160          SBC CH
1170          STA CH
```

```
1180 !
1190 SHLOOP LDA XH        Loop from X to X2
1200        STA MXH
1210        LDA XL
1220        STA MXL
1230        JSR MULT      Find M*X
1240 !
1250        LDX NEG
1260        BNE SHL1
1270        CLC           If M +ve then Y =
                          M*X + C
1280        ADC C
1290        STA Y
1300        JMP SHL2
1310 SHL1   SEC           If M -ve then Y = C -
                          M*X
1320        LDA C
1330        SBC P3
1340        STA Y
1350 SHL2   JSR PLOT      Plot the point
1355 !
1360        CLC           Increment X
1370        LDA XL
1380        ADC #1
1390        STA XL
1400        LDA XH
1410        ADC #0
1420        STA XH
1425 !
1430        SEC           See if X=X2
1440        LDA X2L
1450        SBC XL
1460        LDA X2H
1470        SBC XH
1480        BCS SHLOOP    If X2>=X then repeat
1490        JMP TIDY      Finish off
1500 !
1510 !
1520 STEEP  SEC           Draw steep lines with
                          DY/DX>1
1530        LDA Y2
```

```
1540          CMP  Y
1550          BCS  STEEP1
1560          JSR  SWAP      If Y2<Y then swap
                             coordinates
1565 !
1570 STEEP1   LDA  DXL       Find gradient DX/DY
1580          STA  DIVLO
1590          LDA  DXH
1600          STA  DIVHI
1610          LDA  DY
1620          STA  D1
1630          LDA  #0
1640          STA  D2
1650          JSR  DIVIDE
1655 !
1660          LDA  Y         Find Y*M
1670          STA  MXL
1680          LDA  #0
1690          STA  MXH
1700          JSR  MULT
1710          STA  C
1720          LDX  NEG
1730          BEQ  ST1
1740          CLC            If M -ve then C= M*Y
                             + X
1750          ADC  XL
1760          STA  C
1770          LDA  #0
1780          ADC  XH
1790          STA  CH
1800          JMP  STLOOP
1810 ST1      LDA  XL        If M +ve then C = X -
                             M*Y
1820          SBC  C
1830          STA  C
1840          LDA  XH
1850          SBC  #0
1860          STA  CH
1865 !
1870 STLOOP   LDA  Y         Loop from Y to Y2
1880          STA  MXL
```

```
1890          LDA #0
1900          STA MXH
1910          JSR MULT      Find M*Y
1920          LDX NEG
1930          BNE STL1      If slope -ve then
                            branch
1940          CLC           X = M*Y+C
1950          ADC C
1960          STA XL
1970          LDA CH
1980          ADC P4
1990          STA XH
2000          JMP STL2
2005 !
2010 STL1     SEC           X = C-M*Y
2020          LDA C
2030          SBC P3
2040          STA XL
2050          LDA CH
2060          SBC P4
2070          STA XH
2075 !
2080 STL2     JSR PLOT      Plot pixel
2085 !
2090          CLC           If Y<=Y2 then repeat
2100          LDA Y
2110          ADC #1
2120          STA Y
2130          CMP Y2
2140          BCC STLOOP
2150          BEQ STLOOP
2155 !
2156 !
2160 TIDY     LDA XTH       Set X and Y to new
                            line end coordinates
2170          STA XH
2180          LDA XTL
2190          STA XL
2200          LDA YT
2210          STA Y
2220          RTS
```

```
2230 !
2235 !
2240 MULT   LDA Q1        Multiply number in
                          MX by gradient in Q1-
                          Q3
2250        STA M1        Copy gradient to M1-
                          M3
2260        LDA Q2
2270        STA M2
2280        LDA Q3
2290        STA M3
2300        LDA #0        Initialise product P to
                          zero
2310        STA P1
2320        STA P2
2330        STA P3
2340        STA P4
2350        STA MX3
2360        STA MX4
2365 !
2370        LDY #24       Set Y to repeat 24
                          times
2375 !
2380 MULT1  LSR M3        Shift M to the right
2390        ROR M2
2400        ROR M1
2410        BCC MULT2     If nothing in carry
                          then branch
2420        CLC           Add MX to P
2430        LDA MXL
2440        ADC P1
2450        STA P1
2460        LDA MXH
2470        ADC P2
2480        STA P2
2490        LDA MX3
2500        ADC P3
2510        STA P3
2520        LDA MX4
2530        ADC P4
2540        STA P4
```

```
2545 !
2550 MULT2  ASL MXL      Multiply MX by 2
2560        ROL MXH
2570        ROL MX3
2580        ROL MX4
2585 !
2590        DEY          Repeat if Y <0
2600        BNE MULT1
2605 !
2610        LDA P3       Store result in
                         accumulator
2620        RTS
2630 !
2635 !
2640 DIVIDE LDA #0       Divides DIV by D
2650        STA Q3       First set result Q to
                         zero
2660        STA Q2
2670        STA Q1
2680        STA DIV3
2685 !
2690        LDY #24      Set to repeat 24 times
2695 !
2700        SEC          Subtract D from high
                         bytes of DIV
2710        LDA DIVHI
2720        SBC D1
2730        STA DIVHI
2740        LDA DIV3
2750        SBC D2
2760        STA DIV3
2765 !
2770 DVD1   PHP          Save P register for
                         later
2775 !
2780        ROL Q1       Rotate Q: mult by 2
                         and add carry
2790        ROL Q2
2800        ROL Q3
2810        ASL DIVLO    Multiply Div by 2
2820        ROL DIVHI
```

```
2830          ROL DIV3
2835 !
2840          PLP           Reload P register
2850          BCC DVD2      If DIV-D was <0 then
                            branch
2855 !
2860          LDA DIVHI     Subtract D from new
                            value of DIV
2870          SBC D1
2880          STA DIVHI
2890          LDA DIV3
2900          SBC D2
2910          STA DIV3
2920          CLV
2930          BVC DVD3      Jump on to DVD3
2935 !
2940 DVD2     LDA DIVHI     Add D to new value of
                            DIV
2950          ADC D1
2960          STA DIVHI
2970          LDA DIV3
2980          ADC D2
2990          STA DIV3
2885 !
3000 DVD3     DEY
3010          BNE DVD1      Repeat loop 24 times
3015 !
3020          ROL Q1        Multiply result by 2
3030          ROL Q2
3040          ROL Q3
3050          RTS
3060 !
3070 !
3080 SWAP LDA X2L           Swap (X,Y) with
                            (X2,Y2)
3090          LDY XL
3100          STA XL
3110          STY X2L
3120          LDA X2H
3130          LDY XH
3140          STA XH
```

```
3150          STY X2H
3160          LDA Y2
3170          LDY Y
3180          STA Y
3190          STY Y2
3200          RTS
3210 !
3220 ! Label definitions
3225 !
3230 PLOT     = $C0B8
3240 XL       = 679
3250 XH       = 680
3260 Y        = 681
3270 X2L      = 682
3280 X2H      = 683
3290 Y2       = 684
3300 XTL      NOP
3310 XTH      NOP
3320 YT       NOP
3330 DXL      NOP
3340 DXH      NOP
3350 DY       NOP
3360 D1       NOP
3370 D2       NOP
3380 DIVLO    NOP
3390 DIVHI    NOP
3400 DIV3     NOP
3410 MXL      NOP
3420 MXH      NOP
3430 MX3      NOP
3440 MX4      NOP
3450 C        NOP
3460 CH       NOP
3470 Q1       NOP
3480 Q2       NOP
3490 Q3       NOP
3500 M1       NOP
3510 M2       NOP
3520 M3       NOP
3530 P1       NOP
3540 P2       NOP
```

```
3550 P3      NOP
3560 P4      NOP
3570 NEG     NOP
3580 END
```

To use the program to draw from (X,Y) to (X2,Y2), **POKE** the low byte of X into 679, the high byte of X into 680, and Y into 681. Similarly **POKE** X2 into 682 and 683, and Y2 into 684. Set the colour in which the line is to be drawn by **POKE**ing a suitable number into 685, in the same way as with the PLOT routine. The program works equally well in either screen mode. The program begins at location 49608, and so is run by:

```
SYS 49608
```

Like the PLOT routine, this program does not include any checks to ensure that the lines do not run off the screen. You will find that if a line runs off one side of the screen it will reappear at the other, and it is unlikely that any memory other than the display memory will be corrupted. You should however include suitable checks in your BASIC programs to avoid spoiling the displays.

The program works by first checking the gradient of the line. If the line is shallow (the gradient is less than or equal to 1), the line is drawn by the SHALLOW routine which scans from X to X2, finding Y for each value of X by the formula Y=M*X+C. If the line is steep, the STEEP routine is used to draw the line by scanning from Y to Y2 and finding X by each point with the formula X=M*Y+C.

The subroutine DIVIDE is used to calculate the gradient. The number stored in DIV (DIVLO, DIVHI and DIV3) is divided by the number in D (D1 and D2), and the result (the quotient) is stored in Q (Q1 to Q3). The lower two bytes of the result

are the fractional part of the gradient: as the program is designed always to draw lines with a gradient less than 1 the fractional part is very important.

The MULT subroutine multiplies the gradient stored in Q by the value of X or Y stored in MX (MXLO, MXHI, MX3 and MX4) and stores the product in P (P1 to P4). Again the lower two bytes of the result are the fractional part, so the X or Y coordinate to be plotted is found in P3, with the highest bit in P4 if the product represents the X co-ordinate of a steep line.

FILLING BLOCKS

The fourth machine code program in the graphics set is a FILL routine, to plot all the pixels in a given rectangle in one colour. The routine is very short, the machine code equivalent of a BASIC program such as:

```
10    FOR X = X1 TO X2
20    POKE 679, X AND 255: POKE 680,
      X/256
30    FOR Y = Y1 TO Y2
40    POKE 681, Y
50    SYS 49336
60    NEXT Y
70    NEXT X
```

The machine code routine is:

```
10 ! ************
20 ! *** FILL ***
30 ! ************
40 !
50 !
60 * = $C4E2          Starts at 50402
70 !
```

```
 90 !
100 !
110 XL      = 679
120 XH      = 680
130 Y       = 681
140 !
150 X2L     = 682
160 X2H     = 683
170 Y2      = 684
180 !
190 PLOT    = $C0B8
200 !
210 !
230 !
240 FILL    LDA Y         Save Y to reset for
                          each loop
250         STA YT
260 !
280 FILL1   JSR PLOT      Plot point
290         CLC
300         LDA Y         Increment Y
310         ADC #1
320         STA Y
330 !
340         LDA Y2
350         CMP Y         Repeat loop
360         BCS FILL1     until Y>Y2
370 !
380         LDA YT        Reset Y
390         STA Y
400         CLC
410         LDA XL        Increment X
420         ADC #1
430         STA XL
440         LDA XH
450         ADC #0
460         STA XH
470 !
480         SEC           Find X2-X
490         LDA X2H
500         SBC XH
```

```
510          LDA X2L
520          SBC XL
530          BCS FILL1   Repeat loop if
                         X<=X2
540          RTS
550 !
560 !
570 YT       NOP
```

The FILL routine uses the same locations as the Draw routine. **POKE** the X co-ordinate of the top left-hand corner of the block to be filled into locations 679 and 680, and the Y co-ordinate into location 681. The co-ordinates of the bottom right-hand corner of the block should be **POKE**d into location 682, 683 and 684. Location 685 is again used to select the colour.

Using the Graphics Routines

To finish off, here are some short programs which show the use of all the machine code routines we have introduced in the last two chapters. Before running them you must load the graphics machine code into the 64 - see Appendix 7 for details of how to do this. The graphics routines are split into four separate BASIC loader programs which should each be run in accordance with the instructions in Appendix 7. In addition to the four graphics routines, the first program below requires the mix mode machine code described in Chapter 5 to run – a loader program for this is given in Appendix 7.

Graphics Demonstration Program

The first program shows the routines in use and also uses the mixed mode display routine described in Chapter 5.

```
100   REM **********************
110   REM *** GRAPHICS DEMO ***
```

```
120   REM **********************
130   REM
140   REM GRAPHICS PACKAGE MUST BE
      LOADED
150   REM BEFORE THIS PROGRAM IS RUN
160   REM
170   REM
1000  REM SET MULTICOLOUR MODE
1010  POKE 53281,15:POKE 686,2:POKE
      687,6:POKE 688,0
1020  SYS 49159
1030  REM SET SPLIT SCREEN
1040  POKE 50518,225:SYS 50468
1050  PRINT"{CLS}{23 * CD}"
1060  PRINT"   {RVS}MULTI COLOUR BIT
      MAPPED DISPLAY{ROF}"
1070  FOR T=1 TO 1000: NEXT T
2000  REM PLOT POINTS
2005  PRINT"{CLS}{23 * CD}"
2010  PRINT"         {RVS}PLOTTING
      POINTS "
2020  FOR A=0 TO 2*π STEP 0.1
2030  X=100+50*SIN(A)
2040  Y=60-50*COS(A)
2050  GOSUB 2500
2060  NEXT A
2090  FOR T = 1 TO 1000: NEXT T
2100  GOTO 3000
2500  POKE 679,X
2510  POKE 681,Y
2520  POKE 685,1
2530  SYS 49336
2540  RETURN
3000  REM DRAW LINES
3010  PRINT"{CLS}{23 * CD}"
3020  PRINT"         {RVS}DRAWING
      LINES"
3030  X1=10:Y2=150: POKE 685,2
3040  FOR Y1=10 TO 150 STEP 5
3050  X2=Y1
```

```
3060  POKE 679,X1:POKE 681,Y1:POKE
      682,X2:POKE 684,Y2
3070  SYS 49608
3080  NEXT Y1
3090  FOR T = 1 TO 1000: NEXT T
4000  REM FILL BLOCK
4010  PRINT"{CLS}{23 * CD}"
4020  PRINT"          {RVS}FILLING
      BLOCKS"
4030  POKE 685,1
4040  POKE 679,70: POKE681,30
4050  POKE 682,130: POKE684,90
4060  SYS 50402
4070  FOR T = 1 TO 1000: NEXT:END
```

The program will end with the computer in mixed display mode. Use RUN/STOP and RESTORE to return to normal text mode.

Plotting Points

This program uses the PLOT routine to draw a pattern on the screen.

```
10    REM PLOT PATTERN
20    POKE 685,1:POKE 686,2:POKE
      687,7
30    SYS 49152:REM SET UP BMM
35    J=0:K=80
40    FOR N=0 TO 2*Π STEP .05
50    X = 160+J*SIN(N)
60    Y = 95+K*COS(N)
70    POKE 679,X:POKE 681,Y
80    SYS 49336:REM PLOT IT!
90    NEXT N
100   J = J+10:K = K-10
110   IF K >-10 THEN 40
120   GET K$: IF K$="" THEN 120
130   SYS 49266:REM BACK TO TEXT
```

Headache!

This program uses the DRAW routine to create a pattern on the multi-colour bit-mapped mode screen and demonstrates the effect of changing the colours using the COLOUR routine.

```
4     H=180:W=140:XC=80:YC=100:S=10
5     POKE1020,0:POKE1021,96
6     POKE685,1
10    POKE 686,2:POKE 687,1:POKE
      688,6:POKE 53281,12: SYS 49159
15    FOR Y=YC-H/2 TO YC+H/2 STEP S
20    POKE 679,(XC-W/2)AND 255:POKE
      680,(XC-W/2)/256:POKE 681,Y
30    POKE 682,XC AND 255:POKE 683,
      XC/256:POKE 684,YC
40    SYS 49608:POKE 685,(PEEK(685)
      +1-(PEEK(685)=3))AND 3
45    NEXT
50    FOR X=XC-W/2+S TO XC+W/2 STEP
      S
60    POKE 679,X AND 255:POKE 680,
      X/256:POKE 681,YC+H/2
70    POKE 682,XC AND 255:POKE 683,
      XC/256:POKE 684,YC
80    SYS 49608:POKE 685,(PEEK(685)+
      1-(PEEK(685)=3))AND 3
90    NEXT
115   FOR Y=YC+H/2-S TO YC-H/2 STEP-
      S
120   POKE 679,(XC+W/2) AND 255:POKE
      680,(XC+W/2)/256:POKE 681,Y
130   POKE 682,XC AND 255:POKE 683,
      XC/256:POKE 684,YC
140   SYS 49608:POKE 685,(PEEK(685)+
      1-(PEEK(685)=3))AND 3
145   NEXT
150   FOR X=XC+W/2-S TO XC-W/2 STEP-
      S
```

```
160   POKE 679,X AND 255:POKE 680,
      X/256:POKE 681,YC-H/2
170   POKE 682,XC AND 255:POKE 683,
      XC/256:POKE 684,YC
180   SYS 49608:POKE 685,(PEEK(685)+
      1-(PEEK(685)=3))AND 3
185   NEXT
190   FOR CC=1 TO 96
191   POKE 53280,(PEEK(53280)+1) AND
      15
192   POKE 686,(PEEK(686)+1) AND 15
193   POKE 687,(PEEK(687)+1) AND 15
194   POKE 688,(PEEK(688)+1)AND 15
195   POKE 53281,(PEEK(53281)+1) AND
      15
196   SYS 49177
197   FOR T=1 TO 200:NEXT:NEXT
200   GET A$:IFA$="" THEN 200
210   SYS 49266
220   END
```

Lace

This program uses the DRAW routine to create a lace pattern on the standard bit-mapped screen.

```
10    DIM A(36), B(36)
20    POKE 686,6: POKE 687,3 : SYS
      49152
30    L=120: J=80
40    FOR H=1 TO 5
50    FOR N=1 TO 36
60    K = N/18*π
70    A(N)= 128+L*SIN(K):
      B(N)=88+J*COS(K)
100   NEXT N
110   FOR N=1 TO 36
120   M = N+12
130   IF M>36 THEN M=M-36
160   POKE 679,A(N) AND 255: POKE
      680,A(N)/256: POKE 681,B(N)
```

```
170   POKE 682,A(M) AND 255: POKE
      683,A(M)/256: POKE684,B(M):
      POKE 685,1
180   SYS 49608
190   NEXT N
200   L=L/2: J=J/2
210   NEXT H
500   GET K$: IF K$=""THEN 500
510   SYS 49266:END
```

J R's Hat

This program uses the DRAW routine to draw a three dimensional graph – it takes quite a while to run.

```
100   DIM UB(424), LB(424)
110   XC=320:YC=115:XR=175:ZR= 120
120   H=40:W=0.043:XA=107
200   FOR S=1 TO 424
210   UB(S)=0:LB(S)=1000
220   NEXT S
300   POKE 686,2: POKE 687,1:SYS
      49152
500   FOR Z=-ZR+1 TO ZR-1 STEP 5
510   XL=INT(XR*SQR(1-(Z*Z)/
      (ZR*ZR))+.5)
520   X=-XL
530   Y=H*SIN(W*SQR(X*X+Z*Z))
540   X1=X+XC+Z
550   Y1=INT(199-(YC+Y+Z/2)+.5)
600   FOR X=-XL+1 TO XL-1
610   Y=H*SIN(W*SQR(X*X+Z*Z))
620   X2=XC+X+Z
630   Y2=INT(199-(YC+Y+Z/2)+.5)
640   IF Y2>=LB(X2-XA) THEN 680
650   LB(X2-XA)=Y2
660   IF UB(X2-XA)=0 THEN UB(X2-XA)
      = Y2
670   GOTO 700
680   IF Y2<=UB(X2-XA) THEN 730
```

```
690   UB(X2-XA)=Y2
700   POKE 679,(X1/2) AND 255:POKE
      680,(X1/2)/256:POKE 681,Y1
710   POKE 682,(X2/2) AND 255:POKE
      683,(X2/2)/256:POKE 684,Y2
720   POKE 685,1:SYS 49608
730   X1=X2
740   Y1=Y2
750   NEXT X
760   NEXT Z
1000  GET K$:IF K$="" THEN 1000
1010  SYS 49266
```

CHAPTER 5

DISPLAY INTERRUPTS

The VIC-II chip is a very powerful device and allows many different graphics displays to be created. It is possible to extend these capabilities to enable the creation of displays using several different character sets, or more than eight sprites for example – you can even have mixed text and graphics.

This chapter deals with the techniques used in creating such displays, but to understand fully you must know how the VIC-II chip generates its displays, and how a TV or monitor works.

TV Pictures

The pictures on a TV screen are created by an electron beam which is directed at the phosphor coated inner surface of the screen. Where this beam strikes the screen the phosphor glows. To create a full picture the electron beam scans across the screen in rows, varying in intensity as it goes. This variation in intensity is dictated by information from the TV transmitter and produces a corresponding variation in the intensity with which the phosphor glows. When the electron beam reaches the edge of the screen it is turned off and the process starts again from a position just below the last starting position. This process, called *raster scanning*, continues until the bottom of the screen is reached, at which point it starts all over again at the top. The process must happen

many times a second to create a picture that doesn't flicker.

To create a colour picture, the screen is coated with three different types of phosphor which emit the colours red, blue and green when struck by the electron beam. The different phosphors are distributed in tiny dots or blocks over the screen and the colour TV signal must contain information about which of these points the electron beam should strike, as well as luminance information.

In generating its displays the VIC-II chip takes data from video memory and uses it to create a signal which controls the electron beam in the TV in much the same way as a transmitted TV signal does.

The Raster Register

One of the VIC-II registers, the *raster register* (location 53266), is concerned with the current raster position (position down the screen) of the scaning electron beam. It has two functions depending on whether data is written to it or read from it.

If you read the raster register the number returned is the bottom eight bits of the current raster position, as shown in this program:

```
10    PRINT"{CLS}";PEEK(53266):
      GOTO 10
```

Obviously this number is changing very rapidly and incidentally is a good source of random numbers. Since the raster position can be greater than 255, a ninth bit is required and this is Bit 7 of the control register at location 53265.

If data is written to the raster register, it is stored within the VIC chip and used in a raster compare operation – the current raster position is compared with the stored value, and when the two are equal this fact is indicated in the *interrupt register.*

The Interrupt Status Register

When the current raster position equals the stored value, Bit 0 of the interrupt status register is set (Bit 7 is also set for any VIC interrupt). Bit 0 will remain set until you clear it by writing a 1 to that bit.

So far we have seen how we can get an indication of when the scanning electron beam reaches a given point on the screen, but to use this knowledge we must make use of another VIC register – the *interrupt enable register.*

The Interrupt Enable Register

If Bit 0 of this register is set when a raster interrupt is indicated in the interrupt status register, an interrupt will be generated and the 6510 will begin to execute a machine code program whose start address is stored in locations $FFFE and $FFFF (65534 and 65535). Before returning to whatever program was being executed at the time of the interrupt, a program whose start address is stored in locations 788 and 789 is run. If we change the contents of 788 and 789 (the Interrupt ReQuest vectors) to point to a routine of our own, we can alter the display midway through a scan, and so create the effects mentioned at the beginning of the chapter.

To illustrate the technique, here is a short machine code program which changes the background colour of the display half way down the screen. If you don't have an assembler, type in the BASIC

loader program which follows the assembly language listing.

```
100     IRQLO=788          IRQ vectors
110     IRQHI=789
120     RASREG=53266       Raster register
130     IENREG=53274       Interrupt enable
140     SCREEN=53281       Screen colour reg
150     INTREG=53273       Interrupt status
160     IRQVEC=59953       default IRQ vector
170     !
180     SETUP SEI          disable interrupts
190     LDA #<PROG         load IRQ vectors with
200     STA IRQLO          address of new program
210     LDA #>PROG
220     STA IRQHI
230     LDA #1             set bit 0 of interrupt enable
240     STA IENREG          register
242     LDA #0             initialise raster register
244     STA RASREG
245     LDA 53265          including bit 8!
246     AND #127
247     STA 53265
250     CLI                enable interrupts
260     RTS                back to BASIC
270  !
280  !if we get here an interrupt has occurred.
290  !
300  !
PROG    LDA INTREG         examine interrupt register
350     AND #1             for bit 0=1?
360     BEQ NORMAL         if not, exit
370     STA INTREG         raster interrupt
380     LDA RASREG         - clear RASREG
382     BNE RR             if<>0 then branch
384     LDA #145           next interrupt is half way
386     STA RASREG         down the screen
388     STA SCREEN         change screen colour
390     JMP RR2            exit
392     RR LDA #0          next interrupt is at the top
394     STA RASREG         of the screen
```

```
396     STA SCREEN      change colour
420     RR2 PLA         restore values in registers
421     TAY             before interrupt
422     PLA
423     TAX
424     PLA
425     RTI             end
NORMAL JMP IRQVEC       handle other interrupts
```

Here is the BASIC loader for the program, which loads the code into the cassette buffer, runs it and clears itself from memory.

```
5     REM BASIC LOADER FOR INTERRUPT
      DEMO
10    FOR Z=828 TO 901
20    READ D
30    POKE Z,D
40    NEXT Z
50    SYS 828:NEW
20000 DATA 120,169,91,141,20,3,169,
      3,141,21,3,169,1,141,26,208
20010 DATA 169,0,141,18,208,173,17,
      208,41,127,141,17,208,88,96,17
      3
20020 DATA 25,208,41,1,240,33,141,
      25,208,173,18,208,208,11,169,1
      45
20030 DATA 141,18,208,141,33,208,76,
      125,3,169,0,141,18,208,141,33
20040 DATA 208,104,168,104,170,104,
      64,76,49,234
```

How the Program works

Lines 180 to 260 load the IRQ vectors with the start address of the new routine and initialise the registers. Notice that the I flag is set to prevent any interrupts occurring while the program is running (and cleared afterwards!).

The interrupt program itself starts at line 340 by checking for Bit 0 of the interrupt register being set. If it is not then the interrupt wasn't generated by a raster compare routine and control is passed to the normal interrupt routines by line 440.

If a raster interrupt has occurred, Bit 0 of the interrupt register is cleared and the contents of the raster register are examined.

If the interrupt occurred at the top of the screen then the raster register will contain zero. In this case the raster register is loaded with 145 (corresponding to a position half way down the screen – the position where we want the next interrupt to occur) and the screen colour is changed to white.

If the interrupt was at the middle of the screen (i.e. the raster register contained 145), the raster register is cleared so that the next interrupt happens at the top of the screen, and the screen colour is set to 0 which is black.

Before returning from the routine, the accumulator, X and Y registers are retrieved from the stack where they were placed by the 64's interrupt handling routines.

If you run the program you will see that the top half of the screen will be white while the bottom is black. Once running it will have no effect on your BASIC programs, and can be disabled by pressing RUN/STOP and RESTORE.

You will notice that the screen flickers slightly and that the rate of flicker increases when you press a key. This is because each time you press a key an interrupt is generated, calling the new interrupt routine before control is passed to the 64's routine at 59953. If the running of this routine coincides

with the processing of a key press then a raster interrupt will be 'missed' and the screen will flicker. This effect is particularly noticeable with this example, but can be minimised as you will see in later examples.

As we mentioned earlier raster interrupts can be used to create effects not possible by any other means. For example you could display text in different fonts on different parts of the screen by having several character sets in RAM and arranging your interrupt routine to change the character set pointer when a raster interrupt occurs. This technique is used in the next program to display 8 standard sprites and 8 multicolour sprites on the screen at the same time!

An Abundance of Sprites!

```
10      ;**************
15      ;*              *
20      ;* 16 sprites  *
25      ;*              *
30      ;**************
40      ;
50      POINTER=2040
60      YPOS=53249
70      SMCREG=53276
100     IRQLO=788
110     IRQHI=789
120     RASREG=53266
130     IENREG=53274
150     INTREG=53273
160     IRQVEC=59953
170     *=$C000
175 ;
180 SET SEI
190     LDA #<PROG
200     STA IRQLO
210     LDA #>PROG
220     STA IRQHI
```

```
230       LDA #1
240       STA IENREG
242       LDA #0
244       STA RASREG
245       LDA 53265
246       AND #127
247       STA 53265
250       CLI
260       RTS
270 !
300 PROG  LDA INTREG
310       AND #1
320       BEQ NOR
330       STA INTREG
340       LDA RASREG
350       BNE MULT
360       LDA #145
370       STA RASREG
380       LDX #7
390 STD   LDA #14
400       STA POINTER,X
410       DEX
420       BPL STD
421       LDX #14
422       LDA #70
424 Y1    STA YPOS,X
425       DEX
426       DEX
427       BPL Y1
430       LDA #0
440       STA SMCREG
450       JMP EXIT
500 MULT  LDA #0
510       STA RASREG
520       LDX #7
540       LDA #13
550 LOOP  STA POINTER,X
570       DEX
580       BPL LOOP
581       LDA #200
582       LDX #14
```

```
584 Y2   STA YPOS,X
585      DEX
586      DEX
587      BPL Y2
590      LDA #255
600      STA SMCREG
610 EXIT PLA
620      TAY
630      PLA
640      TAX
650      PLA
660      RTI
670 NOR  JMP IRQVEC
```

The next program is a BASIC loader for the machine code.

NOTE: The code occupies the same area of memory as the graphics routines, which will be overwritten.

```
10     REM LOADER FOR 16 SPRITE DEMO
20     FOR Z=49152 TO 49271
30     READ D:POKE Z,D
40     NEXT Z
50     REM
20000 DATA 120,169,31,141,20,3,169,
       192,141,21,3,169,1,141,26,208
20010 DATA 169,0,141,18,208,173,
       17,208,41,127,141,17,208,88,96
       ,173
20020 DATA 25,208,41,1,240,79,
       141,25,208,173,18,208,208,34,
       169,145
20030 DATA 141,18,208,162,7,169,
       14,157,248,7,202,16,248,162,14
       ,169
20040 DATA 70,157,1,208,202,202,16,
       249,169,0,141,28,208,76,111,
       192
```

```
20050 DATA 169,0,141,18,208,162,7,
      169,13,157,248,7,202,16,250,
      169
20060 DATA 200,162,14,157,1,208,202,
      202,16,249,169,255,141,28,208,
      104
20070 DATA 168,104,170,104,64,76,
      49,234
```

You will see that the program is very similar to the last one, the only difference is that when the scan is half-way down the screen the sprite Y position registers are changed to 200, the sprite data pointers are switched to use another set of data and multicolour mode is selected. To see the effect of the interrupt program, enter it followed by this BASIC program:

```
1     REM 16 SPRITES ON ONE SCREEN
      DEMO
5     POKE 53281,0
10    FOR I=0 TO 127:READ D
20    POKE 832+I,D:NEXT
30    FOR I=2040 TO 2047
40    POKE I,14:NEXT
60    FOR I=0 TO 14 STEP 2
70    POKE 53248+I,25+(15*I)
75    POKE 53248+I+1,70
80    NEXT I
90    FOR I=0 TO 7
100   POKE 53287+I,25+I
110   NEXT:POKE 53294,2
120   POKE 53285,1
130   POKE 53286,2
140   POKE 53269,255
150   PRINT"{CLS}{YEL}HERE ARE 8
      STANDARD SPRITES"
160   FOR DE=0 TO 1000:NEXT
170   SYS 49152
```

```
180   PRINT"{15 * CD}HERE ARE 8
      M{WHT}U{RED}L{CYN}T{PUR}I{GRN}
      C{BLU}O{YEL}L{ORG}O{BRN}U{YEL}
      R SPRITES"
190   PRINT"{CD}{CD}{CD}{CD}{CD}
      {CD}{WHT}IT'S ALL DONE WITH
      INTERRUPTS!!"
200   GOTO 200
998   REM
999   REM DATA FOR MULTI COLOR
      SPRITES
1000  DATA 8,136,136,42,170
1010  DATA 168,42,170,168,37
1020  DATA 170,88,148,105,22
1030  DATA 165,170,90,170,170
1040  DATA 170,170,170,170,42
1050  DATA 40,168,42,40,168
1060  DATA 170,170,170,175,255
1070  DATA 250,181,85,94,181
1080  DATA 85,94,173,85,122,43
1090  DATA 85,232,42,255,168
1100  DATA 10,170,160,2,170
1110  DATA 128,0,170,0,0,40,0,0
1120  REM
1999  REM DATA FOR STANDARD SPRITES
2000  DATA 24,24,24,24,60,24,24,126
2010  DATA 24,12,255,48,15,255,240,3
2020  DATA 255,192,7,255,224,15,255
2030  DATA 40,24,126,24,48,126,12,96
2040  DATA 126,6,255,255,255,255,255
2050  DATA 255,255,255,255,255,255,
      255
2060  DATA 127,255,254,56,60,28,28,
      24
2070  DATA 56,14,0,112,28,0,56,56,0,
      28,0
```

You could use this method to get even more sprites on one display, but you would need a machine code program to move them around satisfactorily.

MIXED MODE DISPLAYS

One of the disadvantages of the 64's display compared with some other micros is that you cannot display text at the same time as high resolution graphics. Raster interrupts can be put to use here by switching between bit-mapped mode and standard text mode at the appropriate place on the screen. That is how the split screen in the program demonstrating the graphics routines in Chapter 4 was created.

Here is a listing of the split screen interrupt program and a BASIC loader:

```
100       IRQLO = 788
110       IRQHI = 789
120       RASREG = 53266
130       IENREG = 53274
140       INTREG = 53273
150       IRQVEC = 59953
160       LORES = 49266
170       HIRES = 49221
200 * = $C512
210 SETUP SEI
220         LDA  #<PROG
225         STA  IRQLO
230         LDA  #>PROG
240         STA  IRQHI
250         LDA  #1
255         STA  IENREG
260         LDA  #0
265         STA  RASREG
270         LDA  53265
280         AND  #127
290         STA  53265
300         CLI
310         RTS
500 PROG    LDA  INTREG
510         AND  #1
```

```
520         BEQ NORM
530         STA INTREG
540         LDA RASREG
550         BNE TEXT
560         JSR HIRES
565         LDA #200
566         STA RASREG
570         JMP EXIT
600 TEXT    JSR LORES
610         LDA #0
620         STA RASREG
700 EXIT    PLA
710         TAY
720         PLA
730         TAX
740         PLA
750         RTI
800 NORM    JMP IRQVEC
```

The routine is loaded by the following BASIC program and is located immediately after the graphics routines, which obviously must be in situ for it to work.

```
10     REM MIX MODE LOADER
20     FOR Z=50468 TO 50541
30     READ D:POKE Z,D
40     NEXT Z
20000 DATA 120,169,67,141,20,3,169,
       197,141,21,3,169,1,141,26,208
20010 DATA 169,0,141,18,208,173,17,
       208,41,127,141,17,208,88,96,
       173
20020 DATA 25,208,41,1,240,33,141,
       25,208,173,18,208,208,11,32,69
20030 DATA 192,169,200,141,18,208,
       76,83,197,32,114,192,169,0,141
       ,18
```

```
20040 DATA 208,104,168,104,170,104,
      64,76,49,234
```

To initialise the routine type:

```
SYS 50450
```

The display will now comprise a graphics screen with a six line text window at the foot of the screen.

We have mentioned just a few of the ways raster interrupts can be used to create more interesting displays – it's up to you to adapt the principles to suit your own applications.

OTHER INTERRUPTS

In addition to setting Bit 0 when the current raster position equals the stored raster position, the interrupt status register provides an indication of sprite collisions.

Bit 1 is set when a sprite to data collision occurs and will remain set until cleared. If the corresponding bit in the interrupt enable register is set, an interrupt will be generated. Similarly, Bit 2 will be set when a sprite to sprite collision occurs.

No information is provided as to which sprite(s) are involved – that must be done by your program. But using interrupts means your program does not have to be constantly checking the sprite collision registers so it will run more quickly.

Interrupts are used in many ways by computers – another example is in the program in Chapter 6, where movement of the joystick generates an interrupt. This allows very smooth motion of the cursor, and means that other programs can run at the same time.

CHAPTER 6

PROGRAMS AND PEOPLE

Writing programs to be used by other people is a very different art from the writing of programs which only you will use. Most people who use programs written by others are not skilled in the use of computers, and may not be familiar with the typewriter keyboard. In this chapter we will look at a few ideas for making programs easier to use – making them 'user friendly' as the jargon puts it.

USER FRIENDLINESS

There are two sides to making programs user-friendly. They must allow the user to select options easily and quickly, and they must also tell the user what options are available at all times. A program may have supporting documentation, but the screen display should present enough information for the program to be usable without referring to the notes once the basics have been learnt.

Menus

One of the simplest and most effective ways of assisting the user to control the program is the *menu*. This is a list of the options available, with some means of indicating how to select an option. The simplest form presents the list with a key legend next to each item, indicating that the key should be pressed to select that feature. The function keys of the 64 work well with this type of menu; other possibilities are to use the initial letter

of each option as the selecting key, or to use numbers.

A second form of menu displays the list as before, but indicates the selected item by displaying it in a different colour, or in inverse video. One or two keys are then used to move the selection up and down the list, and a third key acts as the "go" button, indicating that the selected item is to be acted on. The cursor keys and the RETURN key are probaby best in this application.

Mice and Pointers

The typewriter keyboard is far from the ideal device for communicating with a computer for the newcomer. Many people have no experience of using a keyboard, and find using programs very difficult if they are constantly searching for the right key to press. In addition, the keyboard is not always the best way of controlling a computer, even for an expert. To get round these problems, there are a number of different devices now being used to allow the operator to make inputs by pointing at symbols on the screen.

Perhaps the best known of these devices at the moment is the *mouse*, a small box with a ball-bearing underneath which moves a cursor around the screen as the mouse is rolled about on the desk. The mouse usually has one or two buttons on the top which are used to make selections when the cursor reaches the right point on the screen.

Other pointing devices which perform a similar function are the *light-pen*, which detects the light emitted by the screen and so may be used to point directly at the screen, and the *tracker ball*, which rolls a cursor round the screen. The latest innovation is the touch sensitive screen, which

allows the humble human finger to select objects on the display.

The type of display used with these devices may be very different from that used in ordinary textual menus. If the user is making selections by pointing at the screen, the display describing what is available need not use text, but can use small pictorial symbols to indicate options. These symbols are called *icons*, and are used in several of the most advanced business micros now on the market.

SKETCHPAD PROGRAM

This program demonstrates the use of icons on the 64, using a joystick to draw pictures in multicolour bit-mapped mode.

```
90    PRINT "{CLS}{WHT}{CD}" TAB(12)
      "{RVS} 64 SKETCH PAD {ROF}":
      PRINT: PRINT: PRINT
100   C0=6:C1=10:C2=1:C3=0:IC=1
105   POKE 53281,C0: POKE 686,C1:
      POKE 687,C2: POKE 688,C3
110   FB=2
120   REM READ SPRITE DATA
130   FOR I=0 TO 383: READ X: POKE
      23168+I,X: NEXT
140   REM READ MACHINE CODE DATA
150   FOR I=0 TO 202:READ X: POKE
      50542+I,X: NEXT
160   REM SET SPRITE Y COORDS
170   FOR I=53251 TO 53263 STEP 2:
      POKE I,225: NEXT
180   FOR I=0 TO 12 STEP 2: POKE
      53250+I,(24+24*I) AND 255:
      NEXT
990   REM MAIN MENU
```

```
1000  PRINT "{CLS}{WHT}{CD}" TAB(12)
      "{RVS} 64 SKETCH PAD {ROF}":
      PRINT: PRINT: PRINT
1010  PRINT TAB(5) "SHOW CURRENT
      PICTURE   ... {RVS} F1 {ROF}":
      PRINT: PRINT
1020  PRINT TAB(5) "NEW PICTURE
            ... {LTRED}{RVS} F2
      {ROF}{WHT}": PRINT: PRINT
1030  PRINT TAB(5) "LOAD PICTURE
      FROM DISK ... {RVS} F3 {ROF}":
      PRINT: PRINT
1040  PRINT TAB(5) "SAVE PICTURE TO
      DISK   ... {RVS} F5 {ROF}":
      PRINT: PRINT
1050  PRINT TAB(5) "CHANGE COLOURS
            ... {RVS} F7 {ROF}":
      PRINT: PRINT
1060  PRINT TAB(5) "STOP PROGRAM
            ... {LTRED}{RVS} F8
      {ROF}{WHT}"
1100  GET K$: IF K$="" THEN 1100
1110  IF K$="{F1}" THEN SYS 49177:
      GOSUB 2010: GOTO 1000
1120  IF K$="{F2}" THEN GOSUB 2000:
      GOTO 1000
1130  IF K$="{F3}" THEN GOSUB 5000:
      SYS 49177: GOSUB 2010: GOTO
      1000
1140  IF K$="{F5}" THEN GOSUB 6000:
      GOTO 1000
1150  IF K$="{F7}" THEN GOSUB 7000:
      GOTO 1000
1160  IF K$="{F8}" THEN END
1170  GOTO 1100
1990  REM BIT-MAPPED DRAWING
2000  SYS 49159: REM TURN ON MC BIT
      MAP DISPLAY
2010  REM NOW SET SPRITE DATA
      POINTERS
2020  POKE 24568,106:POKE 24569,108
```

```
2030  FOR I=24570 TO 24572: POKE I,
      107: NEXT
2040  POKE 24573,109: POKE
      24574,110: POKE 24575,111
2045  POKE 53248,172: POKE 53249,120
2050  POKE 53288,IC: POKE 53289,C1:
      POKE 53290,C2: POKE 53291, C3
2055  POKE 53292,IC: POKE 53293,IC:
      POKE 53294,IC:POKE 53287,IC
2060  POKE 53269, 255: REM TURN
      SPRITES ON
2065  POKE 53264,192: REM SPRITE X
      MSB
2070  SYS 50542: POKE690,1: REM TURN
      ON CURSOR MACHINE CODE
2075  C=PEEK(53278)
2080  POKE 53271,0: REM SPRITE Y
      EXPANSION
2090  POKE 53277,2↑(FB+1): REM
      SPRITE Y EXPANSION
2095  POKE 685,FB: POKE 53281,C0
2099  REM ICON HANDLING
2100  C=PEEK(53278): IF C=0 THEN
      2100
2105  IF (PEEK(56320) AND 16)<>0
      THEN 2100
2110  CC = C AND 254
2120  IF CC=2 THEN FB=0: GOTO 2090
2130  IF CC=4 THEN FB=1: GOTO 2090
2140  IF CC=8 THEN FB=2: GOTO 2090
2150  IF CC=16 THEN FB=3: GOTO 2090
2160  IF CC=32 THEN GOSUB 3000:
      GOTO2090
2170  IF CC=64 THEN GOSUB 4000: GOTO
      2090
2180  IF CC<>128 THEN 2100
2190  REM RETURN TO MENU
2200  POKE 53269,0: REM TURN OFF
      SPRITES
2210  SYS 50732: REM TURN OFF CURSOR
      INTERRUPT PROG
```

```
2220  SYS 49266: REM BACK TO TEXT
      MODE
2230  POKE 53281,6
2240  RETURN
2990  REM DRAW LINE
3000  POKE 690,0:POKE 53277,CC
3005  IF (PEEK(56320)AND16) <> 16
      THEN 3005
3010  IF (PEEK(56320)AND16) <> 0
      THEN 3010
3020  X1=PEEK(691): X2=PEEK(692):
      Y1=PEEK(693)
3030  IF (PEEK(56320)AND16) <> 16
      THEN 3030
3040  IF (PEEK(56320)AND16) <> 0
      THEN 3040
3050  XL=PEEK(691): XH=PEEK(692):
      Y=PEEK(693)
3060  GOSUB 3100
3070  IF(PEEK(56320)AND16) <> 16
      THEN POKE 685,0: GOSUB 3100:
      POKE 685,FB: GOTO 3050
3080  POKE 690,1: RETURN
3100  POKE 682,X1: POKE 683,X2: POKE
      684,Y1
3110  POKE 679,XL: POKE 680,XH: POKE
      681,Y
3120  SYS 49608: RETURN
3990  REM FILL BLOCK
4000  POKE 690,0:POKE 53277,CC
4005  IF (PEEK(56320)AND16) <> 16
      THEN 4005
4010  IF (PEEK(56320)AND16) <>0 THEN
      4010
4020  X1=PEEK(691): X2=PEEK(692):
      Y1=PEEK(693)
4030  IF (PEEK(56320)AND16) <>16
      THEN 4030
4040  IF (PEEK(56320)AND16) <>0 THEN
      4040
```

```
4050  XL=PEEK(691): XH=PEEK(692):
      Y=PEEK(693)
4060  GOSUB 4200:POKE 685,0: GOSUB
      4200: POKE685,FB
4070  IF (PEEK(56320)AND16) <>16
      THEN 4050
4080  GOSUB 4200
4090  IF X1+256*X2>XL+256*XH THEN
      T1=X1: T2=X2: X1=XL: X2=XH:
      XL=T1: XH=T1
4100  IF Y1>Y THEN T=Y: Y=Y1: Y1=T
4110  POKE 679,X1:POKE 680,X2: POKE
      681,Y1
4120  POKE 682,XL: POKE 683,XH: POKE
      684,Y
4130  SYS 50402
4140  POKE 690,1: RETURN
4200  POKE 682,X1: POKE 683,X2: POKE
      684,Y1
4210  POKE 679,XL: POKE 680,XH: POKE
      681,Y1
4220  SYS 49608
4230  POKE 682,XL: POKE 683,XH: POKE
      684,Y1
4240  POKE 679,XL: POKE 680,XH: POKE
      681,Y
4250  SYS 49608
4260  POKE 682,XL: POKE 683,XH: POKE
      684,Y
4270  POKE 679,X1: POKE 680,X2: POKE
      681,Y
4280  SYS 49608
4290  POKE 682,X1: POKE 683,X2: POKE
      684,Y
4300  POKE 679,X1: POKE 680,X2: POKE
      681,Y1
4310  SYS 49608: RETURN
4990  REM LOAD PICTURE
5000  PRINT "{CLS}{WHT}{CD}" TAB(12)
      "{RVS} 64 SKETCH PAD {ROF}":
      PRINT: PRINT: PRINT
```

```
5010  PRINT TAB(5) "{RVS} LOAD
      PICTURE {ROF}": PRINT: PRINT
5020  INPUT "     PICTURE TITLE";F$
5030  OPEN 1,8,2,F$+"S,R"
5040  INPUT#1,C0
5050  INPUT#1,C1: POKE 686,C1
5060  INPUT#1,C2: POKE 687,C2
5070  INPUT#1,C3: POKE 688,C3
5080  INPUT#1,IC
5090  FOR I=24576 TO 32575
5100  GET#1, D$
5110  IF D$="" THEN D$=CHR$(0)
5120  POKE I,ASC(D$)
5130  NEXT
5140  CLOSE 1
5150  RETURN
5990  REM SAVE PICTURE
6000  PRINT "{CLS}{WHT}{CD}" TAB(12)
      "{RVS} 64 SKETCH PAD {ROF}":
      PRINT: PRINT: PRINT
6010  PRINT TAB(5)"{RVS} SAVE
      PICTURE {ROF}":PRINT:PRINT
6020  INPUT "     PICTURE TITLE";F$
6030  OPEN 1,8,2,F$+"S,W"
6040  PRINT#1,C0
6050  PRINT#1,C1
6060  PRINT#1,C2
6070  PRINT#1,C3
6080  PRINT#1,IC
6090  FORI=24576TO32575
6100  PRINT#1,CHR$(PEEK(I));
6110  NEXT
6120  PRINT#1:CLOSE1
6130  RETURN
7000  PRINT "{CLS}{CD}": INPUT
      "BACKGROUND";C0
7010  PRINT: INPUT "COLOUR
      1";C1:POKE 686,C1
7020  PRINT: INPUT "COLOUR
      2";C2:POKE 687,C2
```

```
7030  PRINT: INPUT "COLOUR
      3";C3:POKE 688,C3
7040  PRINT: INPUT "ICON COLOUR";IC
7050  RETURN
19990 REM DATA FOR SPRITES
20000 DATA 0,0,0,0,0,0,0,0,0,0
20010 DATA 24,0,15,255,240,12,24,48,
      12,24,48,12,0,48,12,0,48,12,0,
      48,31,0,248
20020 DATA 12,0,48,12,0,48,12,0,48,
      12,24,48,12,24,48,15,255,240,0
      ,24
20030 DATA 0,0,0,0,0,0,0,0,0,0,0
20100 DATA 0,0,0,31,192,60,127,252,
      254,127,255,254,127,255,254,
      255,255
20110 DATA 255,255,255,255,63,255,
      255,63,255,255,127,255,254,
      127,255,254,127
20120 DATA 255,252,127,255,248,63,
      255,252,31,255,252,31,255,254,
      31,255,255
20130 DATA 15,255,255,3,255,231,0,
      63,224,0,7,192,0,0,0,0,31
20140 DATA 192,60,63,252,254,112,
      127,230,96,7,134,224,0,7,
      240,0,3
20150 DATA 48,0,3,48,0,7,112,0,6,96,
      0,14,96,0,28,112,0
20160 DATA 24,48,0,12,24,0,12,24,0,
      6,30,0,63,15,224,127,3
20170 DATA 252,103,0,63,224,0,7,192,
      0,224,0,0,112,0,0,56,0
20180 DATA 0,28,0,0,14,0,0,7,0,0,3,
      128,0,1,192,0,0
20190 DATA 224,0,0,112,0,0,56,0,0,
      28,0,0,14,0,0,7,0
20200 DATA 0,3,128,0,1,192,0,0,224,
      0,0,112,0,0,56,0,0
20210 DATA 28,0,0,14,0,0,0,0,0,0,0,
      63,255,252,63,255,252
```

```
20220 DATA 63,255,252,63,0,252,63,
      63,252,63,63,252,63,63,
      252,63,3
20230 DATA 252,63,63,252,63,63,252,
      63,63,252,63,63,252,63,
      63,252,63
20240 DATA 63,252,63,255,252,63,255,
      252,63,255,252,0,0,0,0,0,0
20250 DATA 0,0,0,0,0,0,0,63,255,252,
      48,0,12,48,255,12,48
20260 DATA 0,12,48,0,12,51,252,204,
      48,0,12,48,0,12,51,252,204
20270 DATA 48,0,12,48,0,12,51,252,
      204,48,0,12,48,0,12,51,252
20280 DATA 204,48,0,12,63,255,252,
      0,0,0,0,0,0,0
24990 REM DATA FOR MACHINE CODE
20500 DATA 120,169,123,141,20,3,169,
      197,141,21,3,88,96,173,0,
      220,74
20510 DATA 176,12,72,173,1,208,201,
      41,144,3,206,1,208,104,
      74,176,12
20520 DATA 72,173,1,208,201,239,176,
      3,238,1,208,104,74,176,35,
      72,173
20530 DATA 16,208,41,1,208,7,173,0,
      208,201,14,144,19,56,173,0,208
20540 DATA 233,1,141,0,208,176,8,
      173,16,208,41,254,141,16,
      208,104,74
20550 DATA 176,29,72,173,16,208,41,
      1,240,7,173,0,208,201,75,
      176,13
20560 DATA 238,0,208,208,8,173,16,
      208,9,1,141,16,208,104,74,
      176,68
20570 DATA 56,173,1,208,233,40,141,
      181,2,56,173,0,208,233,
      12,141,179
```

```
20580 DATA 2,173,16,208,233,0,41,1,
      141,180,2,173,180,2,74,141,180
20590 DATA 2,173,179,2,106,141,179,
      2,173,178,2,240,21,173,
      179,2,141
20600 DATA 167,2,173,180,2,141,168,
      2,173,181,2,141,169,2,32,
      184,192
20610 DATA 76,49,234,120,169,49,141,
      20,3,169,234,141,21,3,88,96
```

How to use the program

This program uses the machine code graphics routines introduced in Chapters 3 and 4. The machine code **must** be loaded into the 64 before this program is run. Appendix 7 gives full instructions on how to do this.

The program requires a joystick, which must be plugged into control port 2 at the right hand side of the 64. The joystick is used to move a cursor around the multicolour bit-mapped display, and the fire button of the joystick plots points on the screen.

When you run the program, after a short pause while the data is read, a menu is displayed offering you the following options, which are selected by the function keys:

Show Current Picture: This will change the display to bit-mapped mode without clearing the bit-mapped screen.

New Picture: This clears and displays the bit-mapped screen.

Load Picture:	Loads previously saved pictures from the disk.
Save Picture:	Saves the current bit-mapped picture onto the disk.
Change Colours:	Allows you to change the colours of the display and the icons.
Stop Program:	Stops the program!

Selecting either of the first two options displays a multicolour bit-mapped screen with seven icons at the bottom and a cross-hair cursor in the middle. Moving the joystick will move the cursor, and holding down the fire button will cause points to be plotted under the cursor.

The icons are used to select different line colours, and to use the line drawing and block filling routines. The four left-hand icons represent blobs of paint, and are used to select the line colour. Simply fire at the appropriate icon to change the colour. The selected icon will expand to twice its normal width.

The fifth icon (the line) is used to select line drawing mode. Once this is selected, the next point at which you fire will be the start of the line, and pressing fire again will mark the line end, and the line will be drawn. If you hold down the fire button while marking the line end a line will be flashed on and off, and will follow the cursor around until you release the button, when the line will be drawn in the currently selected colour.

The sixth icon (the F) is used in a similar way to fill blocks on the screen. The two fire pushes after

selecting the fill icon will mark diagonally opposite corners of the rectangle to be filled.

The seventh icon represents the main menu, and is used to return there.

The 64's sprites are used as the icons and the cursor. The collisions detection facility of the VIC chip is used to check whether the cursor sprite is over one of the icon sprites.

The program uses an additional machine code routine to move the cursor and to plot on the bit-mapped screen when the joystick fire button is pressed. This machine code is included in the **DATA** statements in lines 20500 to 20610, and is loaded into memory when you run the program. This program is listed below.

```
100 IRQLO   = 788        64 interrupt vector
110 IRQHI   = 789        stored here
120 !
130 XL      = 679        Co-ordinates for plot
140 XH      = 680
150 Y       = 681
160 !
170 CURSORXL = 691       Cursor sprite co-
                         ordinates
180 CURSORXH = 692
190 CURSORY  = 693
200 !
210 XMIN    = 14         Max and min
220 XMAX    = 75         cursor co-ordinates
230 YMIN    = 41
240 YMAX    = 239
250 !
260 IRQVEC = 59953       Normal interrupt
270 !                    address
280 PLOT    = $C0B8      Plot routine
290 PLOTOK = 690         Plot on/off
300 !
```

```
310 JOYREG = 56320      Joystick port
320 !
330 SPRXL  = 53248      Cursor (sprite 0)
340 SPRXH  = 53264      position registers
350 SPRY   = 53249
360 !
370 !
380 * = $C56E           Starts at 50542
390 !
400 !
410 !
420 SETUP  SEI          Reset interrupt vector
430        LDA #<PROG   to point at this
440        STA IRQLO    program
450        LDA #>PROG
460        STA IRQHI
470        CLI
480        RTS
490 !
500 !
510 PROG   LDA JOYREG   Read joystick port
520        LSR A
530        BCS J2       Branch if bit 0 set
540        PHA
550        LDA SPRY     Stick held up
560        CMP #YMIN    If not at top
570        BCC J1B      move cursor up
580        DEC SPRY
590 J1B    PLA
600 !
610 J2     LSR A        Branch if bit 1 of
620        BCS J3       joystick port set
630        PHA
640        LDA SPRY     Stick held down
650        CMP #YMAX    If not at bottom
660        BCS J2B      move cursor down
670        INC SPRY
680 J2B    PLA
690 !
700 J3     LSR A        Branch if bit 2
710        BCS J4       of joystick port set
```

```
 720          PHA
 730          LDA SPRXH    Stick held left
 740          AND #1       If not at left-hand
 750          BNE J3A      edge of screen
 760          LDA SPRXL    move cursor left
 770          CMP #XMIN
 780          BCC J3B
 790 J3A      SEC
 800          LDA SPRXL
 810          SBC #1
 820          STA SPRXL
 830          BCS J3B
 840          LDA SPRXH
 850          AND #254
 860          STA SPRXH
 870 J3B      PLA
 880 !
 890 J4       LSR A        Branch if bit 3 of
 900          BCS J5       joystick port set
 910          PHA
 920          LDA SPRXH    Stick held right
 930          AND #1       If not at right-hand
 940          BEQ J4A      edge of screen
 950          LDA SPRXL    move cursor right
 960          CMP #XMAX
 970          BCS J4B
 980 J4A      INC SPRXL
 990          BNE J4B
1000          LDA SPRXH
1010          ORA #1
1020          STA SPRXH
1030 J4B      PLA
1040 !
1050 J5       LSR A        Branch if bit 4 of
1060          BCS EXIT     joystick port set
1070          SEC
1080          LDA SPRY     Fire button down
1090          SBC #40      reset plot co-ordinates
1100          STA CURSORY
1110          SEC
1120          LDA SPRXL
```

```
1130          SBC #12
1140          STA CURSORXL
1150          LDA SPRXH
1160          SBC #0
1170          AND #1
1180          STA CURSORXH
1190          LDA CURSORXH
1200          LSR A
1210          STA CURSORXH
1220          LDA CURSORXL
1230          ROR A
1240          STA CURSORXL
1250          LDA PLOTOK  Check if plot allowed
1260          BEQ EXIT    If not branch
1270          LDA CURSORXL
1280          STA XL
1290          LDA CURSORXH
1300          STA XH
1310          LDA CURSORY
1320          STA Y
1330          JSR PLOT    Plot pixel
1340 !
1350 !
1360 EXIT     JMP IRQVEC  Jump to normal
1370 !                    interrupt routine
1380 !
1390 !
1400 RESET    SEI               Disable routine
1410          LDA #<IRQVEC      by resetting
1420          STA IRQLO         interrupt vector
1430          LDA #>IRQVEC      to point to
1440          STA IRQHI         normal interrupt
1450          CLI               handler
1460          RTS
```

CHAPTER 7

MAKING MUSIC

Sound adds another dimension to programs – making them more interesting and exciting or more user friendly. This chapter covers some of the ways in which the 64 can be used to play tunes and melodies, by exploiting the capabilites of its built-in synthesiser – the SID chip.

THE SID CHIP

The SID chip provides the 64 with three voices, each with a range of eight octaves, which can be independently programmed to produce an almost infinite variety of waveforms. The sound so produced may be heard either through the TV speaker or fed to a hi-fi system.

Playing Tunes

To get the 64 to play tunes, you must 'feed' the SID chip with the appropriate frequency data for the notes comprising the tune, in the correct order and at the correct speed.

One way of doing this is shown in this example:

```
1     GOSUB 1000
10    LO=INT(RND(1)*256)
20    HI=INT(RND(1)*256)
30    GOSUB 500
40    FOR D=0 TO 50:NEXT D
50    GOTO 10
500   REM CHANGE NOTE
```

```
510   POKE 54272,LO
520   POKE 54273,HI
530   RETURN
999   REM SET UP SID CHIP
1000  POKE 54296,15:REM VOL
1030  POKE 54277,54:REM ATTACK/DECAY
1040  POKE 54278,168:REM SUST/REL
1050  POKE 54276,33:REM WAVEFORM
1060  RETURN
```

This idea can be extended to cover all three channels like this:

```
1     GOSUB 1000:GOSUB 2000:GOSUB
      3000
10    LO=INT(RND(1)*256)
20    HI=INT(RND(1)*256)
30    POKE L1,LO:POKE L2,LO/2:POKE
      L3,LO/4
40    POKE H1,HI:POKE H2,HI/2:POKE
      H3,HI/4
50    FOR D=0TO50:NEXT D
60    GOTO 10
999   REM SET UP SID CHIP REG 1
1000  POKE 54296,15:REM VOL
1010  L1=54272:REM LOW
1020  H1=54273:REM HIGH
1030  POKE 54277,54:REM ATTACK/DECAY
1040  POKE 54278,168:REM SUST/REL
1050  POKE 54276,33:REM WAVEFORM
1060  RETURN
1999  REM SET UP SID CHIP REG 2
2000  L2=54279:REM LOW
2010  H2=54280:REM HIGH
2020  POKE 54284,54:REM ATTACK/DECAY
2030  POKE 54285,168:REM SUST/REL
2040  POKE 54283,33:REM CONTROL REG
2050  RETURN
2099  REM SET UP SID CHIP REG 3
3000  L3=54286:REM LOW
3010  H3=54287:REM HIGH
```

```
3020  POKE 54291,54:REM ATTACK/DECAY
3030  POKE 54292,168:REM SUST/REL
3040  POKE 54290,33:REM WAVEFORM
3050  RETURN
```

Neither of these examples could be described as music – for that we need a more controlled way of calculating the note frequencies.

The table on page 152 of the Commodore 64 User Guide gives a list of frequency values corresponding to musical notes (if you are unfamiliar with musical notation refer to Appendix 5). Using this table you can build up a series of numbers corresponding to the high and low bytes of the frequency of the notes, but can they be stored? One way is to keep them in a string, reading them one at a time and **POKE**ing them into the appropriate SID registers. This technique is illustrated in the following program:

```
1     GOSUB 1000
10    L$="%%%%?**%%/%"
20    H$="{CD}{CD}{CD}{CD}{HOM}
      {RVS}{RVS}{CD}{CD}"+CHR$(16)+"
      {CD}"
30    FOR Q=1 TO LEN(L$)
40    LO=ASC(MID$(L$,Q,1))
50    HI=ASC(MID$(H$,Q,1))
60    GOSUB 500
70    FOR X=0 TO 100:NEXT X
80    HI=0:LO=0:GOSUB 500
90    NEXT Q
100   STOP
500   REM CHANGE NOTE
510   POKE 54272,LO
520   POKE 54273,HI
530   RETURN
999   REM SET UP SID CHIP
1000  POKE 54296,15:REM VOL
1020  POKE 54277,54:REM ATTACK/DECAY
```

```
1030  POKE 54278,168:REM SUST/REL
1040  POKE 54276,33:REM WAVEFORM
1050  RETURN
```

The main limitation of this method is that the notes are all the same length, which renders it impractical for all but the simplest of tunes.

There are several ways of overcoming this. For example, a second string could be used to hold the data for the length of notes. Add these lines to the last program to hear this technique in action.

```
25    L$="32132121214"
70    FOR D=0 TO 100*VAL(MID$(L$,Q,
      1)):NEXT D
```

This method is adequate for simple tunes, but a better way is to hold information concerning notes and note lengths in **DATA** statements. You can see an example of this technique in the program at the end of the chapter, in which data for high and low frequencies is held in **DATA** statements.

Multi Channel Music

You can easily expand the **DATA** statement method to cater for tunes using all three voices, perhaps with different Attack/Decay and Sustain/Release characteristics to simulate different musical instruments. An extra problem occurs when a note on one voice is of a different length to that on another. To overcome this you could use three duration loop counters, one for each voice, or, more simply, reduce each note to the shortest common length and play a note several times if it is longer than this. This technique is illustrated in the program at the end of this chapter.

MUSIC FROM MACHINE CODE PROGRAMS

Using the SID chip from within machine code programs is very easy, since all you have to do is duplicate the BASIC **POKE**s to achieve the same effect, as the next program illustrates.

```
1      *=$C000
100    VOICE1LOW=54272
110    VOICE1HIGH=54273
120    ATTDEC=54277
130    SUSREL=54278
140    CONTROL=54296
150    NOTES=679
160    !
1000   INIT   LDX #0
1001          LDA #13
1002          STA NOTES
1003          LDA #15
1004          STA CONTROL
1005          LDA #54
1006          STA ATTDEC
1007          LDA #168
1008          STA SUSREL
1009   PLAY   LDA #33
1010          STA 54276
1015          LDA TABLE,X
1020          STA VOICE1LOW
1030          INX
1040          LDA TABLE,X
1050          STA VOICE1HIGH
1060          INX
1070          LDA TABLE,X
1075          INX
1080          TAY
1090          JSR DELAY
1092          LDA #32
1093          STA 54276
1094          LDY #255
1096   LOOP   DEY
1097          BNE LOOP
```

```
1100          DEC NOTES
1110          BNE PLAY
1120  QUIET   LDA #32
1130          STA 54276
1140          LDA #0
1150          STA 54296
1160          RTS
2000  DELAY   LDA #100
2010          STA 162
2015  TIME    LDA 162
2020          BPL TIME
2030  DEY
2040  BNE DELAY
2050  RTS
5000  TABLE   BYT 177,25,2,177,25,2,
              227,22,1,177,25,1,214,28
              ,2,154,21,4
5010          BYT
              63,19,4,37,17,2,37,17,
              2,47,16,1,37,17,1,63,19,
              2,107,14,4
```

However since machine code is so much faster than BASIC you could introduce such tricks as altering the volume of a note or changing the waveform, dynamically. Another possibility is to use interrupts to allow tunes to be played while other actions are being performed – you could even have the 64 play soothing music as you type in a program!!

Calculating Note Frequencies

The frequencies of musical notes are governed by mathematical rules - for example any note is exactly twice the frequency of the same note in the octave below. This means you don't have to rely on a table to calculate the frequencies you need for a particular tune: you can get the 64 to do it for you.

The next program is a simple music editor which allows you to enter a note and its octave in the form C3, which is note C from the third octave, and displays the appropriate frequency numbers for the SID chip to play that note. The note is played and the numbers and notes stored in the array TD%. Sharps and flats can be specified by adding the # or b symbol to the note when you type it in - C4#. The strings describing the notes are stored in the array N$.

```
10    DIM TD%(2,199),N$(199):HI=0:
      LO=0:GOTO 15000
992   REM
993   REM ***************************
994   REM *                         *
995   REM * CALCULATE NOTE, OCTAVE *
996   REM *                         *
997   REM ***************************
998   REM
1000  N=ASC(LEFT$(Z$,1))-65:IF N<0
      OR N>6 THEN EF=1:GOTO 1040
1010  O=ASC(MID$(Z$,2,1))-48:IF O<0
      OR O>7 THEN EF=1:GOTO 1040
1020  IF LEN(Z$)=2 THEN P=0:GOTO
      1040
1030  P=ASC(RIGHT$(Z$,1)):IF P<>35
      AND P<>45 THEN EF=1
1040  RETURN
1992  REM
1993  REM **************
1994  REM *              *
1995  REM * CALC PITCH *
1996  REM *              *
1997  REM **************
1998  REM
2000  IFN>1 THEN N=N-2:GOTO 2020
2010  N=N+5
2020  N=N*2
2030  IF N>4 THEN N=N-1
2040  IF P=35 THEN N=N+1
```

```
2050  IF P=45 THEN N=N-1
2060  RETURN
2992  REM
2993  REM ********************
2994  REM *                  *
2995  REM * CALC FREQ VALUES *
2996  REM *                  *
2997  REM ********************
2998  REM
3000  F=(274*(2↑O))*2↑(N/12)
3010  HI=INT(F/256)
3020  LO=INT(F-HI*256)
3030  RETURN
9992  REM
9993  REM ********
9994  REM *      *
9995  REM * PLAY *
9996  REM *      *
9997  REM ********
9998  REM
10000 POKE 54277,54:REM ATTACK/DECAY
10010 POKE 54278,168:REM SUST/REL
10020 POKE 54276,33:REM WAVEFORM
10030 POKE 54296,15
10040 POKE 54272,LO
10050 POKE 54273,HI
10060 RETURN
14992 REM
14993 REM ***************
14994 REM *             *
14995 REM * ENTER NOTES *
14996 REM *             *
14997 REM ***************
14998 REM
15000 GOSUB 10000:REM SET UP SID
15010 PRINT"s": INPUT "NOTE";Z$
15020 L=LEN(Z$):IF L<2 OR L>3 THEN
      15000
15030 GOSUB 1000
15040 IF EF=1 THEN EF=0:GOTO 15000
15050 GOSUB 2000
```

```
15060 GOSUB 3000
15070 PRINT"HIGH FREQUENCY=";HI
15080 PRINT"LOW FREQUENCY=";LO
15085 TD%(0,Z)=HI:TD%(1,Z)=LO:
      N$(Z)=Z$
15090 GOSUB 10040: Z=Z+1
15095 PRINT "PRESS ANY KEY TO
      CONTINUE"
15100 GET K$:IF K$="" THEN 15100
15110 HI=0:LO=0:GOSUB 10040:GOTO
      15010
```

PICTURES AT AN EXHIBITION

This program plays a piece of classical music in three part harmony to demonstrate some of the techniques covered in this chapter. A lot of typing is involved but the end result is worth it!

```
1     GOSUB 1000:GOSUB 2000:GOSUB
      3000
10    READ H1,L1,H2,L2,H3,L3
15    IF H1=999 THEN END
20    POKE 54273,H1:POKE 54272,L1
30    POKE 54280,H2:POKE 54279,L2
40    POKE 54287,H3:POKE 54286,L3
50    FOR D=0 TO 120:NEXT D
60    GOTO 10
999   REM SET UP SID CHIP REG 1
1000  POKE 54296,15:REM VOL
1030  POKE 54277,54:REM ATTACK/DECAY
1040  POKE 54278,191:REM SUST/REL
1050  POKE 54276,33:REM WAVEFORM
1060  RETURN
1999  REM SET UP SID CHIP REG 2
2000  POKE 54284,54:REM ATTACK/DECAY
2010  POKE 54285,168:REM SUST/REL
2020  POKE 54283,33:REM CONTROL REG
2030  RETURN
2099  REM SET UP SID CHIP REG 3
3000  POKE 54291,54:REM ATTACK/DECAY
```

```
3010  POKE 54292,168:REM SUST/REL
3020  POKE 54290,33:REM WAVEFORM
3050  RETURN
10000 DATA 12,216,0,0,0,0,12,216,0,
      0,0,0,11,114,0,0,0,0,11,114,0,
      0,0,0
10010 DATA 15,70,0,0,0,0,15,70,0,0,
      0,0,17,37,0,0,0,0,22,227,
      0,0,0,0
10020 DATA 19,63,0,0,0,0,19,63,
      0,0,0,0
10025 REM ************************
10030 DATA 17,37,0,0,0,0,22,227,
      0,0,0,0,19,63,0,0,0,0,19,63,0,
      0,0,0,15,70,0,0,0,0
10040 DATA 15,70,0,0,0,0,17,37,0,0,
      0,0,17,37,0,0,0,0,12,216,
      0,0,0,0
10050 DATA 12,216,0,0,0,0,11,114,
      0,0,0,0,11,114,0,0,0,0
10055 REM ************************
10060 DATA 12,216,9,159,7,163,12,
      216,9,159,7,163,11,114,8,
      147,7,53
10070 DATA 11,114,8,147,7,53,15,70,
      9,159,7,163,15,70,9,159,7,163
10080 DATA 17,37,14,107,8,147,22,
      227,14,107,8,147,19,63,14,107,
      11,114
10090 DATA 19,63,14,107,11,114
10095 REM ************************
10100 DATA 17,37,14,107,8,147,22,
      227,14,107,8,147,19,63,15,
      70,11,114
10110 DATA 19,63,15,70,11,114,15,70,
      12,216,9,159,15,70,12,216,
      9,159
10120 DATA 17,37,12,216,10,205,17,
      37,12,216,10,205,12,216,8,
      147,6,108
```

```
10130 DATA 12,216,8,147,6,108,11,
      114,8,147,7,53,11,114,8,
      147,7,53
10135 REM ************************
10140 DATA 11,114,0,0,0,0,11,114,0,
      0,0,0,12,216,0,0,0,0,12,216,0,
      0,0,0
10150 DATA 9,159,0,0,0,0,9,159,0,0,
      0,0,11,114,0,0,0,0,12,216,
      0,0,0,0
10160 DATA 8,147,0,0,0,0,8,147,0,
      0,0,0
10165 REM ************************
10170 DATA 12,216,0,0,0,0,14,107,0,
      0,0,0,11,114,0,0,0,0,11,114,
      0,0,0,0
10180 DATA 22,227,11,114,5,185,22,
      227,11,114,5,185,19,63,11,
      114,7,163
10190 DATA 19,63,11,114,7,163,17,37,
      11,114,6,108,15,70,11,114,
      6,108
10200 DATA 11,114,0,0,5,185,11,114,
      0,0,5,185
10205 REM ************************
10210 DATA 11,114,0,0,0,0,11,114,0,
      0,0,0,12,216,0,0,0,0,12,216,
      0,0,0,0
10220 DATA 9,159,0,0,0,0,9,159,0,0,
      0,0,11,114,0,0,0,0,12,216,
      0,0,0,0
10230 DATA 10,60,0,0,0,0,10,60,0,
      0,0,0
10235 REM ************************
10240 DATA 15,70,0,0,0,0,17,37,0,0,
      0,0,13,156,0,0,0,0,13,156,
      0,0,0,0
10250 DATA 27,56,13,156,6,206,27,56,
      13,156,6,206,22,227,13,156,
      9,21
```

```
10260 DATA 22,227,13,156,9,21,20,
      100,13,156,7,163,18,42,13,156,
      7,163
10270 DATA 13,156,0,0,6,206,13,156,
      0,0,6,206
10275 REM ************************
10280 DATA 13,156,10,60,3,8,13,156,
      10,60,3,8,15,70,10,60,3,8,15,
      70,10,60,3,8
10290 DATA 13,156,10,60,2,220,13,
      156,10,60,2,220,15,70,10,
      60,3,54
10300 DATA 17,37,10,60,3,54,20,100,
      10,60,3,54,15,70,10,60,3,54
10310 DATA 13,156,10,60,6,16,13,156,
      10,60,6,16
10315 REM ************************
10320 DATA 18,42,13,156,11,114,20,
      100,17,37,13,156,22,227,18,42,
      13,156
10330 DATA 27,56,13,156,0,0,24,63,
      20,100,15,70,22,227,18,42,
      13,156
10340 DATA 20,100,17,37,13,156,24,
      63,0,0,12,32,22,227,18,
      42,15,70
10350 DATA 18,42,0,0,9,21,20,100,17,
      37,13,156,20,100,17,37,13,156
10355 REM ************************
10360 DATA 13,156,10,60,3,8,13,156,
      10,60,3,8,15,70,10,60,3,8,15,
      70,10,60,3,8
10370 DATA 13,156,10,60,2,220,13,
      156,10,60,2,220,15,70,10,
      60,3,8
10380 DATA 17,37,10,60,3,8,20,100,
      10,60,6,16,15,70,7,163,0,0
10385 REM ************************
10390 DATA 17,37,12,216,7,163,17,
      37,12,216,7,163,19,63,12,216,
      7,163
```

```
10400 DATA 19,63,12,216,7,163,17,37,
      12,216,7,53,17,37,12,216,7,53
10410 DATA  19,63,12,216,7,163,22,
      227,12,216,7,163,25,170,12,
      216,7,163
10420 DATA 19,63,12,216,7,163,17,
      37,12,216,15,70,17,37,12,
      216,15,70
10425 REM ************************
10430 DATA 22,227,17,37,14,107,25,
      177,21,154,17,37,28,214,22,
      227,17,37
10440 DATA 34,75,0,0,17,37,30,141,
      25,177,3,54,28,214,22,227,7,53
10450 DATA 25,177,21,154,17,37,30,
      141,0,0,15,70,28,214,22,227,
      19,63
10460 DATA 22,227,0,0,11,114,25,
      177,21,154,17,37,25,177,21,
      154,17,37
10465 REM ************************
10470 DATA 28,214,17,37,14,107,21,
      154,17,37,14,107,22,227,19,
      63,7,163
10480 DATA 22,227,19,63,7,163,28,
      214,17,37,2,220,28,214,17,
      37,2,220
10490 DATA 19,63,15,70,10,205,19,63,
      15,70,10,205,28,214,17,37,
      2,220
10500 DATA 28,214,17,37,2,220,19,63,
      15,70,10,60,19,63,15,70,10,60
10505 REM ************************
10510 DATA 22,227,14,107,11,114,17,
      37,14,107,11,114,19,63,15,
      70,11,114
10520 DATA 19,63,15,70,11,114,22,
      227,14,107,4,208,22,227,14,
      107,4,208
```

```
10530 DATA 19,63,15,70,11,114,19,63,
      15,70,11,114,22,227,14,107,
      4,208
10540 DATA 17,37,14,107,4,208,19,
      63,15,70,11,114,19,63,15,
      70,11,114
10545 REM ************************
10550 DATA 17,37,12,216,10,60,
      17,37,12,216,10,205,14,107,11,
      114,8,147
10560 DATA 14,107,11,114,8,147,15,
      70,11,114,9,159,15,70,11,114,
      9,159
10570 DATA 17,37,12,216,10,60,17,
      37,12,216,10,205,14,107,11,
      114,8,147
10580 DATA 14,107,11,104,8,147,15,
      70,11,114,9,159,19,63,11,114,
      9,159
10585 REM ************************
10590 DATA 17,37,12,216,10,205,17,
      37,12,216,10,205,14,107,11,114
      ,9,159
10600 DATA 14,107,11,114,9,159,17,
      37,12,216,10,205,17,37,12,216,
      10,205
10700 DATA 22,227,11,114,0,0,22,227,
      11,114,0,0,20,100,15,70,12,216
10710 DATA 19,63,0,0,9,159,17,37,14,
      107,11,114,15,70,11,114,9,159
10715 REM ************************
10720 DATA 17,37,11,114,3,155,17,37,
      11,114,3,155,19,63,11,114,4,12
10730 DATA 19,63,11,114,4,12,22,227,
      17,37,14,107,22,227,17,37,
      14,107
10740 DATA 25,177,20,100,15,70,30,
      141,0,0,15,70,22,227,11,114,
      2,220
```

```
10750 DATA 22,227,11,114,2,220,25,
      177,12,216,3,54,25,177,12,216,
      3,54
10755 REM ************************
10760 DATA 22,227,11,114,2,220,22,
      227,11,114,2,220,20,100,15,70,
      12,216
10770 DATA 19,63,0,0,9,159,17,37,14,
      107,11,114,15,70,11,114,9,159
10775 DATA 17,37,11,114,8,147,17,37,
      11,114,8,147,19,63,11,114,
      9,159
10776 DATA 19,63,11,114,9,159,22,
      227,17,37,14,107,22,227,17,37,
      14,107
10777 REM ************************
10780 DATA 25,177,20,100,15,70,30,
      140,15,70,0,0,22,227,11,114,
      2,220
10790 DATA 22,227,11,114,2,220,25,
      177,12,216,3,54,25,177,12,
      216,3,54
10800 DATA 22,227,11,114,2,220,22,
      227,11,114,2,220,12,216,0,0,6,
      108,12,216,0,0
10810 DATA 6,108,11,114,0,0,5,185,
      11,114,0,0,5,185
10815 REM ************************
10820 DATA 25,177,21,154,15,70,30,
      141,0,0,15,70,22,227,11,114,
      2,220
10830 DATA 22,227,11,114,2,220,25,
      177,12,216,3,54,25,177,12,216,
      3,54
10840 DATA 22,227,11,114,2,220,22,
      227,11,114,2,220,12,216,10,60,
      7,163
10850 DATA 12,216,10,60,7,163,11,
      114,8,147,7,53,11,114,8,147,
      7,53
10855 REM ************************
```

```
10860 DATA 15,70,11,114,7,163,15,70,
      11,114,7,163,17,37,14,107,
      11,114
10870 DATA 22,227,0,0,11,114,19,63,
      15,70,11,114,19,63,15,70,
      11,114
10880 DATA 17,37,14,107,11,114,22,
      227,0,0,11,114,19,63,15,70,
      11,114
10890 DATA 19,63,15,70,11,114,15,70,
      11,114,9,159,15,70,11,114,
      9,159
10895 REM ************************
10900 DATA 17,37,12,216,10,60,17,37,
      12,216,10,60,12,216,10,205,
      8,147
10910 DATA 12,216,10,205,8,147,11,
      114,8,147,7,53,11,114,8,147,
      7,53
10920 DATA 12,216,9,159,6,108,12,
      216,9,159,6,108,11,114,8,147,
      7,53
10930 DATA 11,114,8,147,7,53,15,70,
      11,114,7,163,15,70,11,114,
      7,163
10935 REM ************************
10940 DATA 17,37,14,107,11,114,22,
      227,0,0,11,114,19,63,15,70,
      11,114
10950 DATA 19,63,15,70,11,114,15,70,
      12,216,9,159,15,70,12,216,
      9,159
10960 DATA 20,100,17,37,12,216,20,
      100,17,37,12,216,17,37,14,107,
      11,114
10970 DATA 17,37,14,107,11,114,15,
      70,11,114,9,159,15,70,11,114,
      9,159
50000 DATA 0,0,0,0,0,0,999,0,0,0,0,0
```

The frequency data is stored in the order Low Byte, High Byte for voices one, two and three.

Arranging music, especially classical music, for the 64 often involves a good deal of pruning to make it work with only three voices – very little of it was composed with the SID chip in mind!

CHAPTER 8

THE 1541 DISK DRIVE

The 1541 disk drive provides a means of storing and retrieving programs and data in a more flexible way than is possible with the cassette unit. The storage medium is the floppy disk, a flexible disk of plastic coated with magnetic material and enclosed in a protective envelope. The principal advantages of floppy disks over cassette tapes are greatly increased speed of data retrieval, and random access to data.

Connecting the Disk Drive

The disk drive connects to the 64 via the serial bus, which uses the 5 pin socket on the back of the computer. A suitable cable is supplied with the disk drive. Always switch on the disk drive before the 64, to avoid damage to the computer. Never switch the drive on or off with a disk in place.

HOW THE DISK DRIVE OPERATES

The 1541 disk drive unit contains both mechanical and electronic parts. The mechanics are used to hold a disk firmly in place when the door is closed and to rotate the disk at approximately 300 rpm. A special motor called a stepper motor controls the movement of a read/write head similar to the one used in the cassette unit. The stepper motor can place the head in contact with the disk, and move it to and fro along the slot cut in the disk envelope. Because the disk is rotating, this arrangement means that any portion of the disk can be reached

by the head. The head can read data from and write it to the disk.

The 1541 contains its own microprocessor, a 6502, which is very similar to the 6510 in the 64. The processor has some RAM memory and a program known as a *disk operating system* stored in ROM. The 2k of RAM is arranged into 256 byte buffers, and any data transfers between computer and disk drive occur via these buffers. There are also some chips to control the interface between the disk drive and the 64.

The Disk Operating System

The disk operating system or DOS is contained in 16k of ROM in the disk drive. This controls all the mechanical functions of the drive and the interface with the computer. One advantage of having the DOS resident within the disk drive is that for many operations the transmission of the appropriate command to the DOS is all the computer needs to do. The DOS carries out the operation leaving the computer free for other tasks.

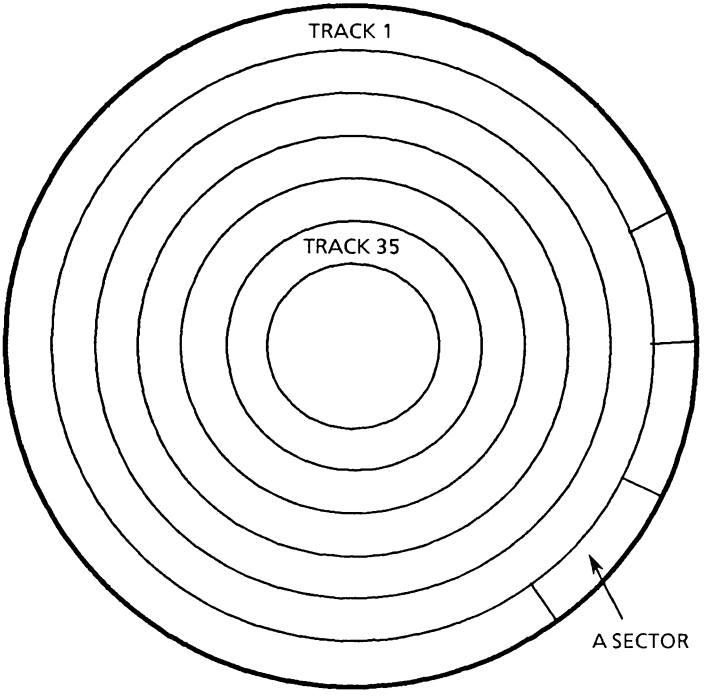
Data Storage on Floppy Disks

Data is stored on a disk in a number of concentric tracks each of which is divided into a number of sectors. The diagram on the next page shows how the tracks are arranged.

A disk for use in a 1541 drive has 35 tracks, track 1 being outermost. Each track is divided into a number of sectors which can store a *block* of data – 256 bytes. The exact number of sectors per track varies, since the tracks are of unequal length – the outermost tracks have 21 sectors while track 35 has only 17 sectors.

TRACK NUMBER	NUMBER OF SECTORS
1 - 17	21
18 - 24	20
25 - 30	18
31 - 35	17

Sector Distribution by Track



The Arrangement of Tracks and Sectors

In addition to the data bytes, each sector contains information used by the DOS for various

housekeeping functions and to indicate on which track and sector the next block of data for a file is located.

Care of Floppy Disks

Floppy disks are rather more delicate than cassette tapes, and to prevent damage great care must be taken when handling and storing them. By following the guidelines in this section you will minimise the risk of losing your valuable programs or data.

a) Always return the disk to its envelope immediately after use, and keep disks in a sturdy box designed for the purpose.

b) Do not store disks near magnetic fields (such as those generated by the TV), since stray magnetic fields can destroy data.

c) Keep disks away from heat or sunlight.

d) Never write on the disk's plastic jacket – fill in sticky labels before applying them to the disk.

e) Do not touch the surface of the disk or try to clean it.

f) Remove disks from the drive before switching it on or off.

With the exception of the danger from magnetic fields, the rules for looking after floppy disks are similar to those for audio records. However, the penalties for not observing them are rather more severe!

Formatting Blank Disks

The 1541 disk drive uses standard 5.25 inch floppy disks. Ask your dealer for *single sided single density* disks.

Before you can use a new disk it must be prepared, by a process called *formatting* to enable the DOS to store data on it. The formatting operation defines the tracks and sectors of the disk, and is carried out by the DOS.

To pass instructions to the drive, you must open a channel, in the same way as you would to send data to the cassette unit or any other peripheral device. The channel number 15 is reserved for communications with the disk unit.

To format the disk type:

```
OPEN 15,8,15

PRINT#15,"NEW:TEST DISK,A1":CLOSE 15
```

These two commands will open the disk command channel and send the NEW command to the disk drive. The NEW command, in common with all other DOS commands, can be abbreviated to its first letter, so:

```
PRINT#15,"N:TEST DISK,A1":CLOSE 15
```

would work equally well.

The disk drive will start to make a noise and the cursor will reappear leaving the 64 free for use while the DOS program formats the disk. The process involves the clearing of any old data from the disk and the setting up of the 683 blocks for data storage. To keep track of what is saved on the disk, some of these blocks are devoted to a

directory, which holds the disk name (in this case TEST DISK, but you could use any name of up to 16 characters) and a list of all the files on the disk. Each block on the disk is labelled with the disk identifier (in this case A1, but any two characters are allowed) which enables the DOS to uniquely identify the disk.

The formatting operation takes about 80 seconds, after which disk activity will cease. You can now examine the directory of the disk by typing:

```
LOAD "$",8

LIST
```

You will see the disk name and identifier in inverse video, and the words '664 blocks free'. The characters '2A' indicate which version of the DOS program was used to format the disk: this is the same for all 1541 drives.

Because nothing has been stored on the disk yet, the maximum number of blocks is available for use (664*256 bytes = 169984 bytes of storage), and there are no entries in the directory.

Clearing the Directory

Once a disk has been formatted, there is no need to do so again. If you have a disk containing programs you no longer require (be certain!!), you can clear the directory by using the NEW command but omitting the disk ID:

```
OPEN 15,8,15

PRINT#15,"NEW:RECLAIMED DISK"
```

This will not clear the disk, but will change the directory header and make all the sectors available for use so that you can write over any existing data.

THE BLOCK AVAILABLITY MAP (BAM)

The DOS needs to know which of the sectors of a disk have not yet been used, so that maximum use can be made of the space available. To keep track of what's left, track 18 sector 0 is reserved for a map of the available sectors of each track on the disk. This map is known as the *block availablity map* or *BAM.*

The arrangement of data in the BAM is shown in the diagram below:

BYTE	CONTENTS	USE
0, 1	18, 01	Pointer to Track & Sector of first Directory Block
2	65	ASCII 'A' indicates disk format
3	0	Not used
4 - 143	BAM	Bit map of blocks used in tracks 1 - 35

The BAM Format

The first four bytes of the BAM are used by the DOS as storage for information about the disk, the remainder is given over to a bit map of the 683 blocks on a disk. Each bit in a byte of the BAM represents a block of data on the disk. If the bit is set then the block is available for use; if not then

that block has been used. The BAM is updated every time a program is saved or when a data file is closed so that the DOS has available an up to date picture of the storage space left on the disk.

Loading and Saving Programs

The most obvious and immediate use of the 1541 drive is as a means of storing programs. The BASIC commands **LOAD** and **SAVE** are used in the same way as they are with the cassette unit, except that you must specify the disk drive as the device you wish to use by adding the device number 8 to a command. For example to load a program from disk, type:

```
LOAD "MUTO ATTACK",8
```

The BASIC program MUTO ATTACK will be loaded from the disk into memory.

To save a BASIC program on disk:

```
SAVE "KILLER DAISIES",8
```

You can save a revised version of a program so that it replaces the old one with the same name on the disk. To do this insert an @ in the command like this:

```
SAVE "@:BEAN SHOOT",8
```

Machine code programs or blocks of data can also be loaded from disk, by adding a secondary address of 1 to the **LOAD** command:

```
LOAD "TANK BATTLE",8,1
```

will load the file into the area of memory from which it was saved.

SAVE always saves a BASIC program, so to save machine code programs without a monitor or assembler involves modifying the zero page locations used to mark the beginning and end of a BASIC program in memory. The locations concerned are:

43 Low byte of start address of BASIC program
44 High byte of start address of BASIC program

45 Low byte of start of BASIC variables
46 High byte of start of BASIC variables

In normal use, the **SAVE** command stores the contents of memory between the addresses pointed to by the contents of these locations. To use **SAVE** to save data from another area of memory, these two pointers must be loaded with the start and end address of that area. For example to save a machine code program stored from $C000 to $C100, you would type:

```
POKE 44,192:POKE 43,0

POKE 46,193:POKE 45,0
```

followed by:

```
SAVE "MACHINE CODE",8
```

These operations will occur much more rapidly than with the cassette unit. Because of the random access nature of disk drives, several programs can be stored on the same disk without increasing the time taken to load a program.

As with cassettes, it is possible to prevent overwriting of data. The small notch in one edge of the disk is called a *write protect notch,* and to

prevent overwriting this must be covered with one of the small sticky tabs supplied with the disk.

The disk drive is a very flexible device, and is useful for more than storage of programs. To exploit it fully you'll need to know more about how it works.

DISK HOUSEKEEPING OPERATIONS

The DOS recognises a group of commands known as housekeeping commands. As the name suggests these commands allow you to maintain the state of the disk by renaming files, deleting files etc.

To use these commands you must **OPEN** a channel to channel 15 of the disk drive (use the secondary address 15). For example:

```
OPEN 15, 8, 15
```

The commands are then sent to the disk drive using the **PRINT#** command. Remember to **CLOSE** the channel after using it.

The first of the commands is a method of reading the error status of the drive – an error is indicated by the red LED flashing on the front of the drive.

The following program reads the error status of the drive from channel 15:

```
10    OPEN 15,8,15
20    INPUT#15,A$,B$,C$,D$
30    PRINT "ERROR CODE "A$
40    PRINT "ERROR TYPE "B$
50    PRINT "      TRACK "C$
60    PRINT "     SECTOR "D$
```

A list of DOS errors and their meanings is given in Appendix 2.

This is a very cumbersome way of obtaining error information, and a more convenient method is described in the section on the DOS support program on page 150.

Initialise

The INITIALISE command (abbreviated to I) returns the disk drive to its normal state by copying the BAM into disk drive RAM. All files are closed and error conditions cancelled. The syntax is:

```
PRINT#15,"I"
```

You will find this command useful when experimenting with some of the more complex disk operations in the next chapter!

Validate

A VALIDATE operation will 'tidy up' and force the DOS to re-organise the files on the disk and thus make optimum use of the available disk space and free any partially used blocks. The command is abbreviated to V, and its format is:

```
PRINT#15,"V"
```

Scratch

You can delete files from the disk with the SCRATCH (abbreviated to S) command - the format is:

```
PRINT#15,"S:FILE NAME"
```

You can delete a group of files using the 'wild card' facility, for example:

```
PRINT#15,"S:TEST*"
```

would delete all files on the disk whose file names began with the characters TEST. Be careful when using this facility, or you may delete files you want to keep.

Copy

The COPY (C) command allows you to duplicate a file under a different name on the disk. The syntax is:

```
PRINT#15,"C:NEW NAME=OLD NAME"
```

Another function of the COPY command is to combine files - this technique is discussed in the section on sequential files in the next chapter.

Rename

It is possible to alter the directory entry for a file using the RENAME (R) command:

```
PRINT#15,"R:NEW ENTRY=OLD ENTRY"
```

THE DOS SUPPORT PROGRAM

One of the programs on the demonstration disk supplied with the 1541 disk drive is a short machine code program which supports the DOS resident in the disk drive. Known as "DOS 5.1", its function is to simplify disk handling commands.

The machine code is loaded into the 64 by a short BASIC program called "C-64 WEDGE", which is

also on the disk. This simply loads DOS 5.1 and initialises it with a **SYS** call to its start address.

Since DOS 5.1 is located in the 4k block of memory from $C000 it has no effect on the amount of memory available for BASIC programs.

Under DOS 5.1 the @ and > keys are used to issue commands to the disk drive, in exactly the same way as **PRINT#** is used to issue commands via channel 15.

Directory

For example to obtain a directory of a disk, use the command:

```
@$
```

The directory will be loaded directly into screen memory and not as a program, so any program in memory will not be overwritten. You can slow down the scrolling of a long directory by holding down the CTRL key, and use the space bar to stop and start the scrolling.

DOS 5.1 allows you to search the directory for a specific program as follows:

```
@$ PROGRAM NAME
```

If the program is on the disk its name will appear on the screen, otherwise the directory will appear blank.

You can search for and display the names of programs in a specified group by adding an asterisk (*) to the program name:

```
@$ PROG*
```

This will display the names of any programs in the directory beginning with the characters PROG.

Similarly the command

```
@$ ????NAME
```

will search for and display the names of any programs whose last four characters are NAME, regardless of the preceding four characters.

Error Status

When a disk error is indicated by the flashing red LED, the status can be read by typing:

```
@ (or >)
```

without the need to run a program.

This will result in a display of the error code and type (as given in Appendix 2), and the track and sector on which the error occurred, in the following format:

XX, ERROR MESSAGE, TT, SS

where

XX is the error code number

TT is the track number

SS is the sector number

Loading BASIC Programs

To load a BASIC program under DOS 5.1, type:

```
/PROGRAM NAME
```

and press RETURN.

Quotation marks are optional.

Loading Machine Code Programs

To load a machine code program, type:

```
%MACH PROG
```

and press RETURN. This has the same effect as the statement:

```
LOAD "MACH PROG",8,1
```

and loads the machine code program or data into memory, starting at the location from which it was saved, rather than at 2048 as with a normal **LOAD** or / command. Again DOS 5.1 does not require quotation marks around the program name.

Auto Loading BASIC Programs

DOS 5.1 allows you to load a BASIC program so that it will run automatically, by typing:

```
↑PROGRAM NAME
```

Saving BASIC Programs

You can save a BASIC program by typing:

```
←PROGRAM NAME
```

All the other operations described earlier in this chapter which involve issuing commands to the disk drive via channel 15 can be duplicated with the @ or > keys. For example, to format a disk, type:

```
@N:DISK NAME,ID
```

Quitting DOS 5.1

If you want to stop using DOS 5.1, it can be de-activated with the command:

```
@Q
```

This will return the 64 to its normal state, but will not delete DOS 5.1 from memory, and it can be reactivated by typing:

```
SYS 52224
```

DOS 5.1 makes the life of the 1541 disk user much easier, but it's not very convenient to have to load it into the computer from the demonstration disk every time you use the machine. It is a good idea to copy it onto your own disks. To do this first load DOS 5.1 into the 64 from the demonstration disk, then type:

```
POKE 44,204:POKE 43,0

POKE 46,207:POKE 45,89
```

Now insert your disk and type:

```
SAVE "DOS 5.1",8
```

The machine code will be saved on your disk for your future use, because the four **POKE**s 'tricked' the 64 into thinking "DOS 5.1" was a BASIC program – the technique is described earlier in this chapter. To reset the pointers afterwards type:

```
NEW
```

CHAPTER 9

ADVANCED DISK OPERATIONS

The disk drive is useful for more than program storage and this chapter covers the various types of data file available with the 1541 drive, and the commands needed to make use of them. A section on the use of machine code with the disk drive is included.

SEQUENTIAL DATA FILES

Most applications for computers require some means of storing and retrieving data. In a simple system, a cassette unit can be used for storage, but this has the limitation that to access any item of data on a tape, the computer must first read through all those that precede it - the data is stored sequentially, one byte after another, along the tape. This method of data storage is acceptable for small amounts of data, but only a very keen enthusiast could make much serious use of such an arrangement for an application such as a database.

Sequential files can also be stored on floppy disks in exactly the same way as with cassettes, but are much faster in use.

As with cassettes, disk sequential files are created with the **OPEN** command. The syntax of the **OPEN** statement when creating a sequential file is:

```
OPEN LFn,Dev,SA,"NAME,S,Type"
```

The logical file number, LFn, can be any number between 0 and 255, and is used to reference that file

throughout the program. A device number, Dev, of 8 specifies the disk drive, and the Secondary Address, SA, may be any number between 2 and 14. Filenames may contain up to sixteen characters, and the character 'S' indicates a sequential file.

The type parameter specifies whether the file is to be read (R) or written (W).

If no type parameter is included, a new file is created ready for a write operation. When the **OPEN** command is executed, the DOS checks to see if the file already exists and if so the file is opened ready for a read or write operation, as specified by the type parameter. If the file doesn't exist and a write operation is specified, a new file is created. Any other use of **OPEN**, such as trying to open a non-existent file for a read operation, will result in an error.

Reading and Writing Sequential Files

The **PRINT#** command is used to write data to a sequential file. Data items can be separated by carriage returns (CHR$(13)), commas or semi-colons. The rules are exactly the same as those for printing to the screen and with cassette files and need not be covered here.

The **GET#** and **INPUT#** commands are used to read data from a sequential file and again their use is exactly the same as with cassette files.

Closing Sequential Files

A file must be closed after use. This will complete the transfer of data from the buffer onto the disk, and free the channel for other uses. If the file is not

closed, some data may be lost. The command to close a file is:

```
CLOSE LFn
```

The next program shows how to create a sequential file on a disk, and how data can be written to and read from it:

```
1     REM SEQUENTIAL DEMO
10    OPEN 6,8,6,"SEQUENTIAL FILE,S,
      W":REM OPEN SEQUENTIAL FILE
      FOR A WRITE OPERATION
20    FOR X=0 TO 25:REM WRITE DATA
30    PRINT#6,CHR$(65+X);
40    NEXT X
50    CLOSE 6
99    REM READ IT BACK!
100   OPEN 6,8,6,"SEQUENTIAL FILE,S,
      R":REM OPEN FOR A READ
      OPERATION
110   FOR X=0 TO 25
120   INPUT#6,A$
130   PRINT A$;
140   NEXT X
150   CLOSE 6
```

You can re-open a sequential file to enable it to be updated by inserting a @ symbol in the command like this:

```
OPEN 3,8,10,"@:UPDATED FILE,S,W"
```

Concatenating Sequential Files

You can add the contents of up to four sequential files together making one large file, with the COPY command. For example the following statement would add the contents of the file called STOCK to the file called PRICES and create a new file called

INVENTORY which would contain all the information.

```
PRINT#15, "C:INVENTORY=0:STOCK,
0:PRICES"
```

Sequential files are useful for storing such things as character sets or sprite data, in which no single item of data is to be retrieved alone. The word processor program which we used to write this book stores its text as a sequential file. Because data is stored one item after another, sequential files make very efficient use of the disk space, but are inflexible when it comes to more sophisticated applications.

There is another type of data file you can use with the 1541 drive which overcomes some of the limitations of sequential files and allows you to access data from any point on the disk without having to read all that precedes it. This is known as the *relative file,* and its use results in a large time advantage over sequential files, at the expense of less efficient use of disk space and more complex programming.

RELATIVE DATA FILES

Relative files allow the programmer ready access to any item of data on the disk by structuring the data into records. The DOS keeps a record of the tracks and sectors used by a relative file by establishing *side sectors* – a list of pointers to the start of each record within a relative file. This procedure is handled completely by the DOS, thereby simplifying the programming task.

Each record in a relative file may contain up to 254 characters – the arrangement is shown in the table on the next page.

BYTE	CONTENTS
0, 1	Pointer to T & S of next data block
2-255	254 bytes of data. Empty records have FF as first character, all the rest are null. Partially filled records are padded with nulls.

Relative File Data Block Format

Each side sector is contained in a single data block and can store pointers, in the form of track and sector numbers, for up to 120 records.

BYTE	CONTENTS
0, 1	Pointer to T & S of next side sector block
2	Side sector Number
3	Record Length
4, 5	Track & sector of side sector number 0
6, 7	Track & sector of side sector number 1
8, 9	Track & sector of side sector number 2
10, 11	Track & sector of side sector number 3
12, 13	Track & sector of side sector number 4
14, 15	Track & sector of side sector number 5
16-255	Track & sector pointers to 120 data blocks

Relative File Side Sector Block Format

Each relative file can have up to 6 side sectors, and so may comprise up to 720 records – more than the capacity of a disk!

Creating a Relative File

When a relative file is created, a side sector is created for that file, containing the data given in the table above, and the first record of that file is

set up. The BASIC **OPEN** command is used to create a relative file:

```
OPEN LFn,Dev,Chan,"NAME,L,"+ CHR$
(Rec.Length)
```

An example of this command in use is:

```
OPEN 2,8,2,"DATABASE,L,"+CHR$(50)
```

where file number 2 has been used to create a relative file called DATABASE, having records which are 50 characters long.

Once a file has been created it can be accessed in the usual way, without specifying file type:

```
OPEN 2,8,2,"DATABASE"
```

Reading and Writing Relative Files

To read or write data in a relative file, the number of the required record must be specified using the POSITION command. Like other DOS commands, POSITION (abbreviated as P) is sent to the drive via channel 15:

```
PRINT#15,"P"CHR$(C)CHR$(Low)CHR$(High)
```

Where C is a channel number and Low and High are the low and high bytes of the record number (two bytes are needed because a record number may be greater than 255).

The POSITION command positions the *file pointer* to the specified record before a read or write operation. If you position the pointer to a record which hasn't been created, the error code 50 is created, which should not be regarded as an error, rather a warning that no **INPUT#** or **GET#** operation should be attempted. It is quite in order

to ignore the message if you intend to write data into the record, since the act of writing will create the record. Data is written to a relative file in the same way as to a sequential file - using **PRINT#** after setting the file pointer to the required position.

If it is envisaged that a large number of records are to be created, it is advisable to create the last record at the start of the program. This will force the DOS to create all the intermediate records, and any extra side sectors as required. Each record so created will contain 255 as the first character, indicating that is unused. By creating these records, all subsequent operations with the relative file will be greatly speeded up.

The use of the POSITION command with relative files is illustrated in the following program:

```
10    OPEN 15,8,15
20    OPEN 2,8,2,"RELATIVE
      TEST,L,"+CHR$(100)
30    PRINT#15,"P"CHR$(2)CHR$(100)
      CHR$(0)
40    INPUT#15,A,B$,C,D
50    PRINT "ERROR"A;B$
60    PRINT#2,"THIS IS THE 100TH
      RECORD"
70    CLOSE 15:CLOSE 2
80    END
```

How the Program Works

The program creates a relative file with a record length of 100 characters, and uses the POSITION command to place the file pointer at record 100. A check of the error channel gives the error message ERROR 50 - RECORD NOT PRESENT, but as explained above, this is interpreted as a warning not to read data. Line 60 ignores the error and

writes some data into record 100. This also causes records 1 to 99 to be created with 255 as the first character, and the two channels are closed in line 70.

The next program will allow you to read the contents of the relative file just created, by using POSITION to move the file pointer to record 100 and read the contents.

```
99    REM READ IN 100 TH RECORD
100   OPEN 15,8,15
110   OPEN 2,8,2,"RELATIVE TEST"
120   PRINT#15,"P"CHR$(2)CHR$(100)
      CHR$(0)
130   INPUT#2,D$
140   PRINT "THE CONTENTS OF RECORD
      100 ARE:"D$
150   CLOSE15:CLOSE2
```

If you modify the program to read any other record, a 'STRING TOO LONG' error will be generated when the program attempts to read the data. This occurs because each empty record contains a CHR$(255) followed by 99 bytes of CHR$(0) and no terminator character, like a carriage return or comma. The 80 character limit on the BASIC **INPUT#** command is therefore exceeded and an error occurs.

You can overcome this problem by using **GET#** to check that the first character of a record isn't 255 before reading the contents of that record. To do this, a further facility of the POSITION command is used - the ability to place the file pointer at any position within a record. This is achieved by adding a further parameter to the P command specifying the position within a record:

```
PRINT#15,"P"CHR$(2)CHR$(15)CHR$(0)CHR$
(10)
```

This command would place the file pointer at the 10th byte of record 15 in the relative file accessed via channel 2.

If you change the following lines in the last program, you will be able to use this technique to test each record in the file:

```
115   INPUT"WHICH RECORD";R
120   PRINT#15,"P"CHR$(2)CHR$(R)
      CHR$(0)CHR$(0)
125   GET#2,T$:IF T$=CHR$(255) THEN
      PRINT "EMPTY RECORD":GOTO 150
```

If you run the modified program using any record number between 0 and 99 you will discover empty records. If you try record 100, the DOS will become confused about the contents of the record because the **GET#** in line 125 has moved several bytes of data into the buffer, so you will get peculiar results! To prevent this occuring in more serious applications you must reset the file pointer after checking for an empty record if you then want to use **INPUT#**.

The ability to position the file pointer to any point within a record means that a record can be divided into fields, allowing more efficient use of the disk space and leading to a more flexible way of creating databases. To make it work, you must keep track of the length of each field in order to be able to read and write into each field. The example given in the 1541 manual of a mailing list shows the principles behind the technique (obvious errors aside!). The program later in this chapter is a more comprehensive example of the use of relative files together with indexing sequential files.

RANDOM DATA FILES

Random data files provide a means of specifying the track and sector of the disk upon which your data is stored. They are harder to program than relative files and offer no advantage over them in most applications. They do offer greater control over what goes where on the disk, but are less efficient in their use of disk space than relative files.

To use random files two channels must be opened to the disk drive, one for sending commands (channel 15) and another for sending data to one of the 256 byte RAM buffers in the disk drive.

Opening a Random File

The syntax of the **OPEN** command for a random file is shown below:

```
OPEN LFn,Dev,Channel No,"#"
```

An example of its use is:

```
OPEN 5,8,5,"#"
```

The # symbol can optionally be followed by a number to specify in which of the buffers you want to store the data, but this is not normally necessary. The DOS selects the next available buffer for you, and the only reason for choosing a specific one is for machine code applications where some code is to be stored in the buffer, and you need to know at what address it starts.

Reading and Writing Random Files

Having **OPEN**ed a random file, another command is sent via channel 15 to instruct the DOS to either read the contents of a specified track and sector

into the selected buffer, or to write the contents of the buffer to a specified track and sector of the disk.

The two commands which do this are BLOCK-READ and BLOCK-WRITE (abbreviated as B-R and B-W).

BLOCK-READ

The syntax of the BLOCK-READ command is:

```
PRINT#15,"B-R:"Chan;Drv;Tk;Sec
```

Note that the parameters are separated by semi-colons (;) and *not* commas as described in some versions of the 1541 disk drive manual. The drive number,Drv, is 0 for a single drive unit like the 1541 and can be omitted.

The following program demonstrates the use of the BLOCK-READ command to display the contents of any track and sector.

```
5     REM BLOCK READ DEMO
10    OPEN 15,8,15
20    OPEN 5,8,5,"#"
30    PRINT"{CLS}": INPUT "WHICH
      TRACK,SECTOR?";T,S
40    IF(T>18 AND S>20)OR (T<25 AND
      S>18)OR(T<31 AND S>17)OR(T<36
      AND S>17) THEN 30
50    PRINT#15,"B-R:"5;0;T;S
60    FOR Z=0 TO 255
70    GET#5,D$
80    IF ST=0 THEN A$=A$+D$:NEXT Z
90    PRINT "{CLS}TRACK "T" SECTOR"
      S:PRINT
100   PRINT A$
110   CLOSE 5:CLOSE 15:END
```

This program will work for any block, but to see something meaningful, try using it to examine the directory (track 18 sector 1).

How the Program Works

Lines 10 and 20 open channel 15 for commands and channel 5 to the buffer.

Line 30 sets the variables T and S to the chosen track and sector and line 40 checks that such a combination of track and sector exists on the disk. If not, another input is requested.

Line 50 sends the Block Read (B-R) command, specifying the buffer associated with channel 5, drive 0 and the track and sector numbers which specify the data block to be read.

The loop in lines 60 to 80 continues to read data one byte at at time from the buffer for as long as the system status word, ST, is zero. When the end of the data in the block is reached ST is set to 64 and control jumps to line 90, which displays the contents of track T, sector S.

BLOCK-WRITE

The BLOCK-WRITE command performs the opposite action to the B-R command and allows you to write data to any block on the disk. Its syntax is:

```
PRINT#15,"B-W:"Chan;Drv;Tk;Sec
```

Again the parameters are separated by semicolons, and the drive number is zero for a single drive.

The following program writes the letters of the alphabet onto track 1; sector 1 of the disk - you could check this by using the B-R program above.

```
10 OPEN 15,8,15
20 OPEN 5,8,5,"#"
30 FOR I=0 TO 25
40 PRINT#5,CHR$(I+65)
50 NEXT I
60 PRINT#15,"B-W:"5;0;1;1
70 CLOSE 15:CLOSE 5
```

Notice that the data is written into the buffer via channel 5, and when this operation is complete the contents of the buffer are written to the block specified in the B-W command.

In these examples, we have not specified which buffer is to be used for B-R and B-W operations, leaving the choice up to the DOS. This is perfectly acceptable under normal circumstances, but in very complex programs you may want to know which buffer has been selected for a particular operation. You can find out by issuing a **GET#** command immediately after opening the channel for data (in our case channel 5). The byte returned is the number of the buffer selected by the DOS. You can only interrogate the disk drive for buffer information *before* any read or write operations are carried out with that buffer.

You can probably imagine that writing data haphazardly all over the disk in this way isn't very useful - you can easily lose track of what data is where and if you inadvertently overwrite another file or the directory you can ruin the disk. For random files to co-exist safely with program and other files on a disk, care must be taken to check the BAM for a free block *before* writing data to the disk, and to update the BAM after writing the data.

The command which allows you to do this is BLOCK-ALLOCATE.

BLOCK-ALLOCATE

The syntax of BLOCK-ALLOCATE (abbreviated as B-A) is:

```
PRINT#15,"B-A:"Drv;Tk;Sec
```

If you precede an attempt to select a block for use in a random file by a B-A command, the DOS will check the BAM to see if that block is available. If it isn't, an error is returned via channel 15, along with the numbers of the next available track and sector. The next program shows B-A in operation.

```
10    PRINT "{CLS}": OPEN 15,8,15
20    T=18:S=1
30    PRINT#15,"B-A:"0;T;S
40    INPUT#15,A,B$,C,D
50    IF A<>65 THEN 80
60    PRINT "TRACK"T" SECTOR"S" IS
      NOT AVAILABLE": PRINT
70    PRINT "THE NEXT FREE BLOCK IS
      AT TRACK "C: PRINT TAB(26)
      "SECTOR"D:END
80    PRINT "TRACK"T" SECTOR"S" MAY
      BE USED"
```

How the Program Works

The track and sector for allocation are set to 18 and 1 which is the start of the directory, and consequently not available. Line 30 attempts to allocate this block, and line 40 reads the results of the attempt from channel 15. If, as in this example, the DOS finds that the requested block is unavailable (i.e. its entry in the BAM is set to 1) then an error code of 65 is returned and the numbers of the next available track and sector are

supplied. If the block is available, the B-A command updates the BAM and a write operation can be performed. The results of the B-A attempt are displayed by lines 60 to 80.

In practice, this test would be carried out before any attempt to write data.

BLOCK-FREE

BLOCK FREE (B-F) is a corresponding command which allows you to release a block for use. No data is destroyed but the BAM entry for the specified block is cleared allowing new data to overwrite what is already there.

The syntax of the B-F command is:

```
PRINT#15,"B-F:"Drv;Tk;Sec
```

To make use of random files, you need to keep a record of the blocks allocated to a random file. The best way of doing this is by maintaining a 'directory' in a sequential file. Each random file would have a corresponding sequential file containing information about which blocks have been used.

The following short program stores data in a random file and keeps an index of the track and sectors as they are used.

```
10    T=1:S=1
20    OPEN 15,8,15
30    OPEN 5,8,5,"#"
40    OPEN 4,8,4,"@0:INDEX,S,W"
50    INPUT"NUMBER OF ENTRIES";N
55    PRINT#4,N
60    FOR X=1 TO N
70    PRINT"{CLS}ENTER DATA FOR
      ENTRY NUMBER";X
```

```
80    INPUT D$
90    PRINT#5,D$
100   PRINT#15,"B-A:"0;T;S
110   INPUT#15,A,B$,C,D
120   IF A=65 THEN T=C:S=D:GOTO 100
125   PRINT#15,"B-W:"5;0;T;S
130   PRINT#4,T","S
140   NEXT X
150   PRINT"BYE!!"
160   CLOSE 4:CLOSE 5:CLOSE 15
170   END
```

How the Program Works

Lines 20 to 40 open the three channels, channel 4 being used for the sequential file which scratches any previous file called "INDEX", and line 50 inputs the number of entries to be made to the file. This number is stored as the first character in the sequential file "INDEX". The loop in lines 60 to 140 inputs file data from the keyboard, allocates a block with the B-A command, writes the data to the block and adds the track and sector number of the block to the sequential file.

Reading data back from such random files involves using the data in the sequential file to find the track and sector at which any data item is located, and using the B-R command to access it. This is demonstrated in the next program:

```
200   OPEN 15,8,15
210   OPEN 5,8,5,"#"
220   OPEN 4,8,4,"INDEX,S,R"
230   INPUT#4,N
240   INPUT "WHICH RECORD TO
      RETRIEVE";NR
250   IF NR>N THEN PRINT"NO SUCH
      RECORD !":GOTO 240
260   FOR X=1 TO NR
270   INPUT#4,T,S
```

```
280   NEXT X
290   PRINT#15,"B-R:"5;0;T;S
300   INPUT#5,A$
310   PRINT A$
330   CLOSE 4:CLOSE 5:CLOSE 15
340   END
```

If you run these two programs, and then examine the directory of the disk, you'll see that there is no directory entry for the random file, but that the number of blocks free is reduced by the number of entries you specified in the first program. After you have experimented you can free the blocks taken up by the random files using the VALIDATE command.

The Buffer Pointer

One drawback of the technique in the programs above is that it is inefficient - an entire block would be used to store only one character. You can increase the efficiency of the method by storing more than one data item in a block. To make this possible, the DOS keeps a count of the number of characters written to the buffer during a random file operation - it is kown as the *buffer pointer.* Each time you create a random file data block, the buffer pointer is stored on the disk with the data.

You can use the buffer pointer to divide blocks into fields and so make more efficient use of the disk space.

The buffer pointer can be set to any position within a block with the BUFFER-POINTER command (abbreviated as B-P), its syntax is:

```
PRINT#15,"B-P:"Chan;Position
```

This program shows how you can use the B-P command to subdivide a block into fields.

```
1     REM B-P WRITE
10    PRINT"{CLS}":OPEN 15,8,15
20    OPEN 5,8,5,"#"
30    OPEN 4,8,4,"B-P INDEX,S,W"
40    FOR P=1 TO 220 STEP 20
45    READ D$
50    PRINT#15,"B-P:"5;P
60    PRINT#5,D$","P
70    NEXT P
80    T=1:S=1
90    PRINT#15,"B-A:"0;T;S
100   INPUT#15,A,B$,C,D
110   IF A=65 THEN T=C:S=D:GOTO 90
120   PRINT#4,T","S
130   PRINT#15,"B-W:"5;0;T;S
140   CLOSE4:CLOSE5:CLOSE15
150   END
160   DATA ZERO,ONE,TWO,THREE,FOUR,
      FIVE,SIX,SEVEN,EIGHT,NINE,TEN
```

How the Program Works

Three channels are opened to the disk and the loop between lines 40 and 70 creates each field of the record in the buffer, the buffer pointer being incremented by 20 for each item in line 50. When all eleven data items are in the buffer in positions specified by line 50, a disk block is allocated in the usual way, the track and sector recorded in the indexing sequential file and the data written to the disk by line 130.

The next program allows you to inspect the random file just created, by using the buffer pointer to select a single data item from the buffer.

```
1     REM B-P READ DEMO
10    PRINT"{CLS}"
```

```
20    OPEN 15,8,15
30    OPEN 5,8,5,"#"
40    OPEN 4,8,4,"B-P INDEX,S,R"
50    INPUT"FIELD TO VIEW (0-10)";R
60    IF R<0 OR R>10 THEN 30
70    BP=R*20+1
80    INPUT#4,T,S
90    PRINT#15,"B-R:"5;0;T;S
100   PRINT#15,"B-P:"5;BP
110   INPUT#5,A$,N
120   PRINT"{CLS}FIELD"R"CONTAINS"A$
130   PRINT "AND STARTS AT BYTE"N
140   CLOSE4:CLOSE5:CLOSE15
150   END
```

How the Program Works

The logic of the program is the reverse of that in the B-P read example. The buffer pointer position is calculated from the entered field number, the track and sector of the block are read from the sequential file by line 80, and the buffer filled with the contents of that block by line 90. The buffer pointer is set to the required data item by line 100 and the contents of the field read from the buffer by line 110.

There are two further commands which are variations of the BLOCK-READ and BLOCK-WRITE commands covered in this section. Called USER 1 and USER 2 these commands allow you to deal with the contents of a complete block regardless of the buffer pointer.

USER 1 and USER 2 (abbreviated as U1 or UA and U2 or UB) are two of a set of commands used in machine code applications which are discussed later in this chapter.

The following program is an example of the use of relative file handling techniques, combined with sequential files.

HOME BASE

```
1     REM *************
2     REM *             *
3     REM * HOME BASE *
4     REM *             *
5     REM *************
6     REM
7     REM REQUIRES 1541 DISK DRIVE
8     REM
9     PRINT"{CLS}"
10    GOSUB 10000
20    GOSUB 240
40    PRINT TAB(13);"{RVS} HOME BASE
      {ROF}{CD}{CD}{CD}{CD}"
50    PRINT TAB(9)"{RVS}1{ROF}
      CREATE NEW FILE{CD}"
60    PRINT TAB(9)"{RVS}2{ROF}
      ENTER RECORD{CD}"
70    PRINT TAB(9)"{RVS}3{ROF}
      SEARCH FOR RECORD{CD}"
80    PRINT TAB(9)"{RVS}4{ROF}
      EXIT{CD}{CD}{CD}{CD}{CD}"
100   PRINT L$
110   PRINT TAB(10);"{DKGRY}ENTER
      CHOICE {RVS}1{ROF} TO
      {RVS}4{ROF}{GRN}";
120   GET A$:IF A$="" THEN 120
130   IF A$<"1" OR A$>"4" THEN 120
135   M$=BK$:GOSUB 300
140   ON ASC(A$)-48 GOSUB 1000,4000,
      5000,6000
150   GOTO 20
199   REM READ ERROR CHANNEL
200   OPEN 15,8,15
210   INPUT#15,A$,B$,C$,D$
220   CLOSE 15
```

```
230   RETURN
239   REM CREATE SCREEN LAYOUT
240   PRINT"{GRN}{CLS}{ROF}CHR$
      (169){14 SPACES}CHR$(223)
      {RVS}{24 SPACES}"
250   PRINT"{16 SPACES}{{RVS}{24
      SPACES}{ROF}";
260   PRINTTAB(39);CHR$(223)
270   RETURN
299   REM DISPLAY M$ ON STATUS LINE
300   PRINT"{HOM}{ 23 * CD}";SPC((
      40-LEN(M$))/2);M$;"{GRN}":IF
      NB=1 THEN 330
310   FOR N=1 TO NU:PRINT"{CU}";:
      NEXT N
320   PRINT BK$"{CU}"
330   NU=0:RETURN
349   REM DRAW HORIZ LINE
350   PRINT"{HOM}{21 * CD}";L$;
      "{HOM}{CD}{CD}"
360   RETURN
993   REM
994   REM **********************
995   REM *                    *
996   REM * CREATE A NEW FILE *
997   REM *                    *
998   REM **********************
999   REM
1000  GOSUB 240
1010  PRINT TAB(10);"{RVS} CREATE A
      NEW FILE {ROF}{CD}{CD}
      {CD}{CD}"
1020  INPUT" FILE NAME (<15 CHARS)";
      FT$
1030  IF LEN(FT$)>14 THEN 1000
1040  IF FT$ = "" THEN RETURN
1050  PRINT:PRINT:INPUT" HOW MANY
      FIELDS/RECORD (<11)";NF
1060  IF NF=0 OR NF>10 THEN
      PRINT"{CU}{CU}{CU}":GOTO 1050
1080  FOR Z=1 TO NF
```

```
1090  GOSUB 240:GOSUB350:PRINT
      TAB(10);"{RVS} CREATE A NEW
      FILE{ROF}{CD}{CD}{CD}{CD}"
1100  PRINT "ENTER NAME OF
      FIELD"Z;:INPUT N$(Z):
      PRINT:PRINT
1105  IF N$(Z)="" THEN RETURN
1110  PRINT"ENTER LENGTH OF
      FIELD"Z;:INPUT FL(Z)
1120  RL=RL+FL(Z)
1130  IF RL>254THEN RL=RL-FL(Z):M$=
      "{RED}RECORD LENGTH EXCEEDED":
      NU=12:GOSUB300:GOTO1110
1140  NEXT Z
1150  GOSUB 240
1160  PRINT "{HOM}{CD}{CR}{DKGRY}
      "FT$"{GRN}"
1170  PRINT TAB(10);"{CD}{CD}{RVS}
      CREATE A NEW FILE {ROF}{CD}
      {CD}{CD}"
1180  PRINT"   #"TAB(6);"NAME";
      TAB(25);"LENGTH{CD}"
1190  FOR Z=1 TO NF
1200  PRINTTAB(2);Z;TAB(6);N$(Z);
      TAB(25);FL(Z)
1210  NEXT Z:GOSUB 350
1220  M$="{DKGRY}   {ROF}PRESS
      {RVS}Y{ROF} TO CONTINUE
      {RVS}N{ROF} TO REJECT":
      NU=5:GOSUB 300
1230  GET YN$:IF YN$="" THEN 1230
1240  IF YN$="N" THEN RETURN
1250  OPEN 2,8,2,FT$+",L,"+CHR$(RL)
1255  CLOSE 2
1260  OPEN 4,8,4,"@0:INDEX,S,W"
1270  PRINT#4,FT$
1275  FR=1
1280  PRINT#4,FR
1290  PRINT#4,NF
1300  PRINT#4,RL
1310  FOR Z=1 TO NF
```

```
1320  PRINT#4,N$(Z)","FL(Z)","
1330  NEXT Z
1340  CLOSE 4
1350  RETURN
3993  REM
3994  REM ********************
3995  REM *                  *
3996  REM * ENTER NEW RECORD *
3997  REM *                  *
3998  REM ********************
3999  REM
4000  IF LEN(FT$)<>0 THEN 4100
4010  OPEN 4,8,4,"INDEX,S,R"
4020  GOSUB 200
4030  IF B$="OK" THEN CLOSE 4:GOTO
      4060
4040  NU=5: M$="{RED}"+B$:GOSUB 300
      :FOR Q=0 TO 1000:NEXT Q
4050  CLOSE 4:RETURN
4060  OPEN 4,8,4,"INDEX,S,R":
      INPUT#4,FT$,FR,NF,RL
4070  FOR Z=1 TO NF
4080  INPUT#4,N$(Z),FL(Z):NEXT Z
4090  CLOSE 4
4100  FOR Z=1 TO NF:GOSUB 240:GOSUB
      350
4110  PRINT "{HOM}{CD}{CR}{DKGRY}"
      FT$"{GRN}"
4120  PRINT
      TAB(11);"{CD}{CD}{RVS}ENTER A
      RECORD {ROF}{CD}{CD}{CD}{CD}"
4130  M$="MAX FIELD LENGTH ="+STR$
      (FL(Z)):NU=16:GOSUB 300
4140  PRINT N$(Z);
4150  INPUT R$(Z)
4160  IF LEN(R$(Z))<FL(Z) THEN 4220
4170  NB=1:M$="{RED}FIELD LENGTH
      EXCEEDED":GOSUB 300:NB=0
4180  FOR T=0 TO 400:NEXT T:GOTO
      4110
4220  NEXT Z
```

```
4230  FOR Z=1 TO NF:GOSUB 240:GOSUB
      350
4240  PRINT "{HOM}{CD}{CR}{DKGRY}
      "FT$"{GRN}"
4260  PRINT TAB(11);"{CD}{CD}{RVS}
      ENTER A RECORD {ROF}{CD}{CD}
      {CD}{CD}"
4270  FOR Z=1 TO NF
4280  PRINT N$(Z)" "R$(Z)
4290  NEXT Z
4300  NB=1:M$="{DKGRY}PRESS {RVS}Y
      {ROF} TO ACCEPT {RVS}N{ROF} TO
      REJECT":GOSUB 300:NB=0
4310  GET YN$:IF YN$="" THEN 4310
4320  IF YN$="N" THEN 4510
4330  IF YN$<>"Y" THEN 4310
4340  OPEN 2,8,2,FT$
4350  OPEN 15,8,15:PO=1
4355  FOR Z=1 TO NF
4360  HI=INT(FR/256):LO=FR-HI*256
4370  PO=PO+FL(Z-1)
4380  PRINT#15,"P"CHR$(2)CHR$(LO)
      CHR$(HI)CHR$(PO)
4390  PRINT#2,R$(Z)
4400  NEXT Z:FR=FR+1
4410  CLOSE 2:CLOSE 15
4420  OPEN 4,8,4,"@0:INDEX,S,W"
4430  PRINT#4,FT$
4440  PRINT#4,FR
4450  PRINT#4,NF
4460  PRINT#4,RL
4470  FOR Z=1 TO NF
4480  PRINT#4,N$(Z)","FL(Z)","
4490  NEXT Z
4500  CLOSE 4
4510  T$="":FOR Z=1 TO NF:R$(Z)="":
      NEXT Z:RETURN
4993  REM
4994  REM **********************
4995  REM *                    *
4996  REM * SEARCH FOR RECORD *
```

```
4997  REM *                    *
4998  REM **********************
4999  REM
5000  SE$="":Y=0:PO=0:IF LEN(FT$)<>0
      THEN 5100
5010  OPEN 4,8,4,"INDEX,S,R"
5020  GOSUB 200
5030  IF B$="OK" THEN CLOSE 4:GOTO
      5060
5040  NU=5: M$="{RED}"+B$:GOSUB 300
      :FOR Q=0 TO 1000:NEXT Q
5050  CLOSE 4:RETURN
5060  OPEN 4,8,4,"INDEX,S,R":
      INPUT#4,FT$,FR,NF,RL
5070  FOR Z=1 TO NF
5080  INPUT#4,N$(Z),FL(Z):NEXT Z
5090  CLOSE 4
5100  GOSUB 240:GOSUB 350
5110  PRINT "{HOM}{CD}{CR}{DKGRY}
      "FT$"{GRN}"
5120  PRINT TAB(10);"{CD}{CD}{RVS}
      SEARCH FOR RECORD {ROF}{CD}
      {CD}{CD}{CD}"
5130  FOR Z=1 TO NF
5140  PRINT "{RVS}"CHR$(Z+64)"{ROF}
      ";N$(Z)
5150  NEXT Z
5160  M$="{DKGRY}   SELECT FIELD FOR
      SEARCH {RVS}A{ROF} TO "+
      "{RVS}"+CHR$(NF+64)+"{ROF}"
5170  NB=1:GOSUB 300:NB=0
5180  GET A$:IF A$="" THEN 5180
5185  IF A$<"A" OR A$>CHR$(NF+64)
      THEN 5180
5190  F=ASC(A$)-64:GOSUB 240:GOSUB
      350:T$=""
5200  PRINT "{HOM}{CD}{CR}{DKGRY}
      "FT$"{GRN}"
5210  PRINT TAB(10);"{CD}{CD}{RVS}
      SEARCH FOR RECORD {ROF}{CD}
      {CD}{CD}{CD}"
```

```
5220  M$=BK$:GOSUB 300: M$="{DKGRY}
      ENTER CONTENTS OF FIELD FOR
      SEARCH":NU=16:GOSUB 300
5225  PRINT N$(F);" ";
5230  INPUT SE$
5260  M$=BK$:GOSUB 300:M$="SEARCHING
      FOR RECORD":NB=1:GOSUB 300:
      NB=0
5300  OPEN 2,8,2,FT$
5310  OPEN 15,8,15
5315  FOR Q=0TOF-1:PO=PO+FL(Q):
      NEXT:PO=PO+1
5320  FOR Z=1 TO FR-1
5330  HI=INT(Z/256):LO=Z-HI*256
5350  PRINT#15,"P"CHR$(2)CHR$(LO)
      CHR$(HI)CHR$(PO)
5360  INPUT#2,T$
5399  REM WILD CARD SEARCH
5400  IF RIGHT$(SE$,1)<>"*" THEN
      5410
5405  IF LEFT$(T$,LEN(SE$)-1)=LEFT$
      (SE$,LEN(SE$)-1)THEN FO(Y)=Z:
      Y=Y+1:GOTO 5415
5410  IF T$=SE$ THEN FO(Y)=Z:Y=Y+1
5415  NEXT Z
5420  M$=BK$:GOSUB 300:M$=STR$(Y)+"
      RECORDS WERE FOUND":NB=1:GOSUB
      300:NB=0
5430  FOR T=0 TO 500:NEXT T
5440  IF Y=0 THEN GOTO 5790
5499  REM DISPLAY FOUND RECORDS
5500  FOR Z=0 TO Y-1:PO=1
5505  GOSUB 240:GOSUB 350
5510  PRINT "{HOM}{CD}{CR}{DKGRY}
      "FT$"{GRN}"
5511  PRINTTAB(10);"{CD}{CD}{RVS}
      SEARCH FOR RECORD {ROF}{CD}
      {CD}{CD}{CD}"
5515  F=FO(Z):HI=INT(F/256):LO=F-
      HI*256
5520  FOR Q=1 TO NF
```

```
5530  PO=PO+FL(Q-1)
5540  PRINT#15,"P"CHR$(2)CHR$(LO)
      CHR$(HI)CHR$(PO)
5550  INPUT#2,R$(Q-1)
5560  PRINT BK$"{CR}{CU}{RVS}";
      CHR$(Q+64)"{ROF} "N$(Q);"
      ";R$(Q-1)
5565  NEXT Q
5570  IF Y=1 THEN 5700
5580  IF Z=Y-1 THEN Z=-1
5600  M$=BK$:GOSUB 300:NB=1:
      M$="{DKGRY}    SHOW NEXT
      RECORD {RVS}Y{ROF} OR
      {RVS}N{ROF}?":GOSUB 300:NB=0
5610  GET YN$:IF YN$="" THEN 5610
5620  IF YN$="N" THEN AR=Z:Z=Y-
      1:GOTO 5700
5630  IF YN$<>"Y" THEN 5610
5640  NEXT Z
5700  M$=BK$:GOSUB300:NB=1:M$=
      "{DKGRY}      {RVS}A{ROF}MEND
      {RVS}DEL{ROF}ETE OR {RVS}
      RETURN{ROF}?":GOSUB 300:NB=0
5710  GET YN$:IF YN$="" THEN 5710
5720  IF YN$="A" THEN 5800
5730  IF YN$=CHR$(13) THEN 5780
5740  IF YN$<>CHR$(20) THEN 5710
5750  M$=BK$:GOSUB300:M$="{DKGRY}
       {RVS}  ARE YOU SURE ?  {ROF}"
      :NB=1:GOSUB300:NB=0
5755  GET YN$:IF YN$="" THEN 5755
5760  IF YN$<>"Y" THEN 5700
5770  PO=1:FOR J=1 TO NF:FOR K=1 TO
      FL(J)-1:DE$=DE$+" ":NEXT K:
      PO=PO+FL(J-1)
5775  PRINT#15,"P"CHR$(2)CHR$(LO)
      CHR$(HI)CHR$(PO):PRINT#2,DE$:D
      E$="":NEXT J
5780  Y=0:PO=0:CLOSE2:CLOSE15:RETURN
5799  REM AMEND RECORD
```

```
5800  PRINT"{HOM}{CD}{CD}{CD}{CD}
      "BK$"{CR}{CU}"TAB(14)"{RVS}AME
      ND RECORD{ROF}"
5805  M$=BK$:GOSUB 300:M$="{DKGRY}
      {RVS}A{ROF}MEND WHICH FIELD?"
      :NB=1:GOSUB 300:NB=0
5810  GET YN$:IF YN$="" THEN 5810
5820  IF YN$<"A" OR YN$> CHR$(NF+64)
      THEN 5810
5830  GOSUB 240:GOSUB 350:PRINT"
      {HOM}{CD}{CR}{DKGRY}"FT$"{GRN}
      "
5840  PRINT TAB(14);"{CD}{CD}{RVS}
      AMEND RECORD {ROF}{CD}{CD}"
5845  Q=ASC(YN$)-64
5850  PRINT N$(Q);SPC(2);R$(Q-1)
5860  M$=BK$:GOSUB 300:M$="MAX FIELD
      LENGTH ="+STR$(FL(Q)):NB=1:
      GOSUB 300:NB=0
5870  PRINT"{HOM}{10 * CD}"N$(Q);
      SPC(2);:INPUT R$(Q-1)
5880  GOSUB 240:GOSUB 350
5890  PRINT "{HOM}{CD}{CR}{DKGRY}
      "FT$"{GRN}"
5900  PRINT TAB(14);"{CD}{CD}{RVS}
      AMEND RECORD {ROF}{CD}{CD}"
5910  FOR Y=1 TO NF
5920  PRINT N$(Y);SPC(1);R$(Y-1):
      NEXT Y
5930  M$="{RVS}{DKGRY}Y{ROF} TO
      ACCEPT OR {RVS}N{ROF} TO
      REJECT":NB=1:GOSUB 300:NB=0
5940  GET YN$:IF YN$="" THEN 5940
5950  IF YN$="N" THEN RETURN
5960  IF YN$<>"Y" THEN 5940
5970  PO=1:FOR Y=0 TO NF-1:PO=PO+
      FL(Y)
5980  PRINT#15,"P"CHR$(2)CHR$(LO)
      CHR$(HI)CHR$(PO)
5990  PRINT#2,R$(Y):NEXT Y:CLOSE2:
      CLOSE15
```

```
5995  RETURN
6000  PRINT"{CLS}BYE!!":END
9993  REM
9994  REM ********************
9995  REM *                    *
9996  REM * SET UP ROUTINES *
9997  REM *                    *
9998  REM ********************
9999  REM
10000 POKE 53280,5:POKE 53281,15
10010 FOR Z=0 TO 39:L$=L$+CHR$(192):
      NEXT Z
10020 FOR Z=0 TO 38:BK$=BK$+
      CHR$(32):NEXT Z
10100 RETURN
```

How to Use HOME BASE

After loading the program, remove the program disk from the drive and insert the file disk. The database can extend over an entire diskette, so it is best to reserve a disk for each file you wish to keep. If you want to create a new file, insert a blank, formatted disk.

Main Menu

The main menu comprises four options:

1 CREATE NEW FILE

2 ENTER A RECORD

3 SEARCH FOR RECORD

4 EXIT

At the foot of the screen is the status line – this is used throughout the program to request inputs (dark grey text), display error messages (red text) and give information about the current operation

(green text). When the program is run the status line will prompt for a choice from the main menu – you will not be able to enter or search for a record until a file has been created on the disk.

1 Create New File

The program will prompt you to enter the name of the file and the number of fields it is to contain. For each field you must enter a name and the maximum number of characters. The total number of characters in a file may not exceed 254.

The file will be created and control will return to the main menu – the file name will be displayed in the top left hand corner of the screen. You can now enter data into the file.

2 Enter a Record

Each field in a record is filled in turn – the status line will display the maximum number of characters allowed in each field. If this number is exceeded an error message will be displayed and you will be asked to re-enter the data. After all the fields have been filled in this way the entire record is redisplayed, at which point you can reject it or accept it. If you accept, the data is stored on the disk, and in both cases the main menu is redisplayed.

3 Search For a Record

The names of the fields will be displayed and you will be asked for the name and contents of the field to be used in the search. A 'wild card' facility is incorporated, so that entering a number of characters followed by an asterisk (*) will locate every record in which the specified field contains those characters. The disk will be searched for the specified field contents and if more than one record

is found, you may display them all by responding 'Y' to the SHOW NEXT FIELD prompt. If you press 'N', or only one record was found, you will be given the option to amend or delete the record or return to the main menu.

Amending a Record

The field to be amended is selected by reference letter and the new data entered. The amended record will be displayed for you to accept or reject. Pressing 'Y' will save the amended record and the main menu will be redisplayed.

Deleting a Record

If you opt to delete a record and press 'Y' in response to the ARE YOU SURE? prompt, the record will be deleted and the main menu redisplayed.

How the Program Works

HOME BASE uses a single relative file to store data, and a sequential file called INDEX which contains information about the names and sizes of the fields in a record, length of a record and the number of the next free record. The arrangement of data in the INDEX file is as follows:

FT$	Name of relative file
FR	Number of the next free record
NF	Number of fields per record
RL	Length of a record
N$(1)	Name of field 1
FL(1)	Length of field 1
N$(2)	Name of field 2
FL(2)	Length of field 2
:	:
etc	etc

INDEX is created at the start of the program when a relative file is set up, and updated each time a new record is added to the relative file.

Lines 40 – 150 display the main menu and input a selection from it.

A number of subroutines in lines 200 – 360 perform frequently used operations such as maintaining the status line, reading the error channel, etc.

Lines 1000 – 1350 create a new relative file, named FT$, containing NF fields. The name and length of each field is entered, and if the format is accepted, a relative file having records of length RL is created, and the INDEX file set up.

Lines 4000 – 4510 handle the entry of a new record. If no file is currently loaded, the disk is checked and data from the INDEX file is loaded. If no INDEX file is found on the disk, an error message is generated and the main menu redisplayed. The data for each field is entered and stored in array R$( ). If the new record is accepted, it is stored on the disk at record FR, the INDEX file is updated and the main menu redisplayed.

Lines 5000 – 6000 provide the search facility. The disk is checked for the existence of an INDEX file if no file is currently loaded, and data is read in from INDEX. The field names contained in array N$( ) are displayed and the field for the search, F, is selected from them. The contents of the field for the search, SE$, are input and the appropriate field of each record on the disk is compared with SE$. If the comparison is successful the record number of that record is stored in array FO( ).

If the last character of SE$ is an asterisk (*) the comparison only covers the number of characters in SE$, providing a 'wild card' search.

At the end of the search, the number of records displayed is Y. If no records were found a message to this effect is displayed and the main menu is redisplayed. If only one record was found, the record is displayed and control passes to line 5700, where amend and delete options are offered. If more than one record was found, pressing 'Y' in response to the 'SHOW NEXT RECORD' prompt will allow each record to be displayed in turn, continuing to cycle until 'N' is pressed, at which point control passes to line 5700.

Lines 5750 – 5799 delete the displayed record, by printing blank strings to the appropriate fields on the disk.

Lines 5800 – 5995 allow amendment of a field in the record and if the amendment is accepted, rewrite the entire record to the disk and redisplay the main menu.

Variable Use

FT$	File name
NF	Number of fields/record
N$( )	Array of field names
FL( )	Array of field lengths
RL	Record length
L$	Horizontal line
M$	Message for status line
BK$	Blank line
FR	Next free record number

R$(I) Contents of field I

Improvements to the Program

In order to make this program as generally applicable as possible (and to allow room for the other chapters in the book!), a number of functions are omitted which would make it much better for specific jobs.

For example, the free format of the display could be replaced by a fixed card format to suit your application.

The restriction of 80 characters per field is due to the BASIC **INPUT** command and could be overcome using **GET** and some form of checking routine.

Additionally, some form of hardcopy option might be useful.

When a record is deleted, it cannnot be re-used and disk space is wasted. One way round this would be to keep another sequential file of deleted record numbers . When a new record is created this would be checked and if there are any entries, the record number of a deleted record would be allocated to the new record.

The program is designed for just one relative file per disk, but if you used yet another sequential file as a directory there's no reason why you shouldn't keep more than one relative file per disk.

With the information provided it would be a useful and instructive exercise to modify and improve HOME BASE to suit your requirements.

MACHINE CODE AND THE 1541 DRIVE

The DOS recognises a group of commands designed to allow you to create machine code programs to run in the disk drive RAM, possibly modifying the operation of the DOS.

To use these commands requires a detailed knowledge of the DOS program and the architecture of the disk drive – information which is not freely available from the manufacturers. However the commands are briefly covered here should you find need to use them.

BLOCK-EXECUTE (B-E)

This command allows you to load machine code routines from the disk into disk drive RAM and execute them. It is similar to the B–R command except that after loading the code, the disk drive's microprocessor begins to execute it. The program must end in an RTS (ReTurn from Subroutine) instruction. The format of the command is:

```
PRINT#15,"B-E:"Ch;Dr;T;S
```

where a block of data is read from track T sector S on drive Dr into the channel Ch buffer, and execution commences at byte 0 of that buffer.

MEMORY WRITE (M-W)

M–W allows you to send data comprising machine code programs from the 64, via channel 15, into disk drive RAM. The format is:

```
PRINT#15,"M-W:"CHR$(Lo)CHR$(Hi)CHR$(N)
CHR$(A)CHR$(B)....etc
```

where a program consisting of bytes A, B etc. up to N characters is sent to RAM starting at address (256*Hi + Lo). Up to 34 bytes may be sent at a time.

MEMORY READ (M-R)

The M-R command provides a means for reading data from disk drive ROM or RAM, one byte at a time, via the error channel into the 64. The format is:

```
PRINT#15,"M-R:"CHR$(Lo)CHR$(Hi)
```

The contents of the location specified by 256*Hi + Lo can be read from channel 15 using **GET#**.

MEMORY EXECUTE (M-E)

Allows you to execute machine code programs in the disk drive memory from the address specified, until an RTS is encountered. The format is:

```
PRINT#15,"M-E:"CHR$(Lo)CHR$(Hi)
```

USER (U)

The USER command makes it possible to link to machine code programs by using a jump table set up in disk drive memory. The command is followed by an ASCII character which forms an index to the table. The characters 1 to 9 or A to J can be used.

The U1 and U2 commands perform the B-R and B-W operations mentioned earlier, but ignore the buffer pointer to operate on an entire data block. The remaining eight point to the locations given in the table opposite, which must be set up to contain the start address of the machine code programs you wish to execute.

USER	OPERATION
U1 or UA	B-R command
U2 or UB	B-W command
U3 or UC	Jump to $0500
U4 or UD	Jump to $0503
U5 or UE	Jump to $0506
U6 or UF	Jump to $0509
U7 or UG	Jump to $050C
U8 or UH	Jump to $050F
U9 or UI	Jump to $FFFA
U; or UJ	Power up Vector

The USER Command Jump Table

The format of the USER command command is :

```
PRINT#15,"UN:"Ch;Dr;T;S
```

USING OTHER DISK DRIVES WITH THE 64

The 1540 Disk Drive

It is possible to use the 1540 disk drive (designed for the VIC 20) with the 64 with one change. The screen must be turned off during the loading of a program with the command:

```
POKE 53265,11
```

When the program is loaded, turn the screen back on with the command:

```
POKE 53265,27
```

Twin 1541 Drives

Many disk based applications are greatly enhanced by having two disk drive units. Although it is possible to link 1541 drives together and change their device number as described in the manual, it seems that some errors in the ROMs cause drives used in this way to 'hang up' at random intervals for no apparent reason. We have tried using two drives in this way for making backup disks, and were plagued with such problems, so be warned!

IEEE Drives

A major disadvantage of the 1541 drive is that data transfers between it and the 64 use a serial data bus. This means that data is transmitted one bit at a time, rather than all 8 bits in a byte being transmitted at once, as in a parallel system. For this reason the 1541 is very slow by disk drive standards as most disk drives adopt the faster (and more expensive) parallel technique.

Another problem with it is that only a relatively small amount of data can be stored on a disk because of the way in which it is formatted.

If you have an application which requires greater storage capacity or more rapid data retrieval, you can use the 64 with some of Commodore's larger (and more expensive) drives.

To do so will involve considerable expense, not only for a dual drive unit like the 8050, but also for an interface adaptor to provide the 64 with the necessary IEEE (parallel) interface.

Copying Tape Software to Disk

A major headache in upgrading from tape to disk is that your collection of programs is still on tape. Where once you would accept a long delay in loading, you soon become spoiled by the speed of disk drives, and need to transfer your software to disk. In a few cases this is simply a matter of loading the program into the 64, and saving it onto disk. However most commercial software is protected aginst copying - unfortunately this means that it is also protected against legitimate copying!

One of the most popular ways of protecting 64 tapes is to save a program in several blocks, and have a short loader program as the first program on the tape. The loader is often written in machine code, to protect against the casual pirate, but is usually quite simple - using kernal routines - to load the program from the tape. The most straightforward way to transfer such a program to disk is to find out how many parts the program is saved in and write a short BASIC loader to do the job of the machine code program on the tape. You can then load each block of program from the tape and save it to disk in the normal way, and use your loader to load the blocks back from the disk when required.

There are other ways of protecting software, and it can be quite satisfying to 'crack' this sort of problem - provided, of course, that you don't sell the results of your efforts!

CHAPTER 10

THE MPS801 PRINTER

A printer is a useful addition to any microcomputer system, providing program listings and hard copy of text and graphics generated by the computer.

There are several different types of printer and many variations of each type are available for the Commodore 64.

The MPS801 printer (formerly the 1525)is the most popular choice for the home user since it is both cheap and versatile – both character and graphic information can be obtained.

How the Printer Works

The MPS801 is a *dot matrix printer* which uses a set of small pins arranged in a matrix to strike the ribbon and make small dots on the paper. Each dot corresponds to one pin.

The printer receives data from the 64 on the serial bus and its microprocessor interprets the data – sorting printing characters from control characters. For each printing character the processor causes the hammer to strike the appropriate pins to create the dot pattern for that character, moves the print head along the carriage and prints the next character. At the end of a line, the print head returns to the lefthand side and the paper is scrolled ready for the next line of data.

Connecting the Printer

Before connecting the MPS801 to the 64, turn off the computer and printer. Plug one end of the serial interface cable into the 64 and the other into the rear of the printer. If you have a 1541 disk drive connect the disk drive to the 64 and the printer to the spare socket on the drive. In cases where more than one peripheral is connected to the bus, problems may arise in addressing either device. This normally requires you to switch off all peripherals, then switch them on again before they can be used. This is due to a fault in the operating system, about which little can be done.

Testing

The MPS801 is provided with a test facility – insert some paper and move the three position switch at the rear of the printer to 'T'. The printer will print the complete 64 character set and continue to do so until you switch off or move the switch away from the 'T' position.

Using the Printer

Like other peripherals, communication between the printer and the 64 is achieved by opening a channel to the device. The channel is opened using the BASIC **OPEN** command:

```
OPEN LFn,Dn,SA
```

where:

LFn is the logical file number (any number from 0 to 255) which is used to reference the channel.

Dn is the device number which is either 4 or 5 depending upon the position of the three position switch at the rear of the printer.

SA is the secondary address, which acts like the CBM and SHIFT keys on the 64 by toggling between upper and lower case mode, and upper case and graphics mode printouts.

To select upper case and graphics mode the secondary address is set to 0. If no secondary address is specified in the OPEN command, it defaults to 0. A secondary address of 7 selects upper and lower case printouts.

Once the channel is opened, characters are transmitted to the printer using the **PRINT#** command as in the following example:

```
10    OPEN 4,4
20    FOR Z=1 TO 26
30    PRINT Z,CHR$(Z+64)
40    PRINT#4,Z,CHR$(Z+64)
50    NEXT Z
60    CLOSE 4
70    END
```

Notice that what is displayed on the screen is mimicked by the printer, and that data can be formatted on the printer in the same way as it can on the screen. A comma moves the print head into the next 'column' before printing and a carriage return is issued at the end of each **PRINT#** command. The carriage return can be suppressed by adding a semicolon at the end of the **PRINT#** command. So, if you change line 20 in the previous program to:

```
20    FOR Z=33 TO 112:PRINT CHR$(Z);
      :NEXT Z
```

and delete lines 30, 40 and 50, the full 80 column capability of the printer will be demonstrated.

As with **PRINT**ing to the display, the **TAB** and **SPC** commands work on the printer.

Another way of obtaining copy on the printer is to use the **CMD** command.

The **CMD** command transfers the output from the screen to the specified channel. Its syntax is:

```
CMD LFn
```

where the logical file number must be the same as the one specified in the **OPEN** command.

After issuing the **CMD** command, all data normally output to the 64's screen is directed to the printer. This is the way to obtain program listings:

```
OPEN 4,4

CMD 4

LIST
```

After a listing has been obtained, the output from the 64 is still directed to the printer, and to return to normal you must type:

```
PRINT# LFn

CLOSE LFn
```

This will clear any data remaining in the printer buffer and close the channel, returning output to the TV screen.

It is possible to use the **OPEN, PRINT#, CLOSE** and **CMD** commands to output data to the printer under program control as this program shows:

```
10    REM PRINTING UNDER PROG
      CONTROL
20    PRINT"{CLS}"
30    INPUT"NAME ";N$
40    OPEN 4,4
50    PRINT#4,"BIG BROTHER IS
      WATCHING YOU "N$"!!!"
60    CLOSE 4
70    END
```

PRINTING MODES

The MPS801 operates in various printing modes selected by control characters in the data sent to the printer. The table below shows the print modes and the control characters used to select them.

The characters in the above table are sent to the printer in **PRINT#** commands and interpreted by the printer as control characters. The mode so selected will continue to be the printing mode until a further control character is detected by the printer.

Cursor Up and Cursor Down

The characters 'cursor up' (CHR$(145)) and 'cursor down' (CHR$(17)) select which of the two 128 character sets is to be used for printing. The two fonts are the same as those selected on the 64 by holding down the SHIFT key and pressing the CBM key. The following program illustrates the difference:

```
10    REM CRSR UP/CRSR DOWN
20    FOR Z=65 TO 90
30    A$=A$+CHR$(Z)
```

CODE	FUNCTION
145	Cursor up mode
17	Cursor down mode
8	Graphics mode
16	Tab print head
18	Reverse on
146	Reverse off
14	Double width mode
10	Line feed
13	Carriage return
27, 16	Specify dot address
26	Repeat graphic data

Printer Control Codes

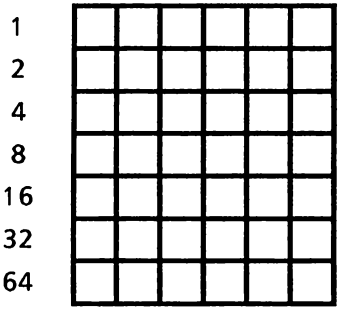
```
40      NEXT Z
50      OPEN 4,4:PRINT#4,A$
60      CLOSE 4
70      OPEN 4,4,7
80      PRINT#4,A$
90      CLOSE 4
100     END
```

NOTE: A secondary address of 7 must be specified in the **OPEN** command for cursor down mode to work.

Graphics Mode

Graphics mode (CHR$(8)) allows the printing of user defined characters.

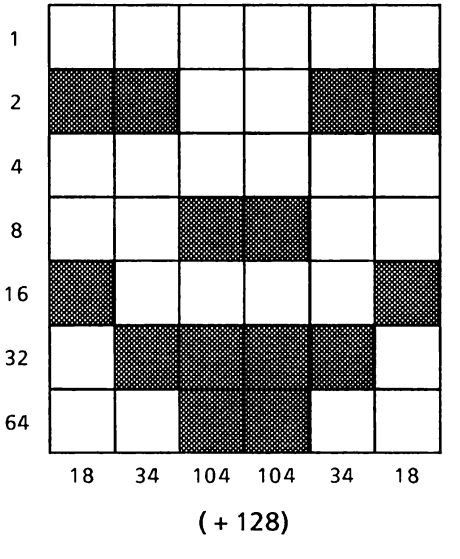
Characters are designed on a 6 by 7 grid like this:



User Defined Character Grid

Notice that the grid is not the same as the one used for defining characters on the 64!

Characters are created by placing dots on the grid where a dot should appear on the paper as in the diagram below:



A User Defined Character

To obtain the **DATA** statement defining the character take one column of the grid at a time and add together the values of the rows in that column containing a dot. Add 128 to the result and repeat the operation for each of the columns.

The data for the character is sent to the printer as a string, following a CHR$(8). The next program will print the character on the grid opposite:

```
10    REM USER DEFINED CHAR PROG
20    FOR Z=0 TO 5
30    READ D
40    A$=A$+CHR$(D)
50    NEXT Z
60    OPEN 4,4
70    PRINT#4,CHR$(8)A$
80    CLOSE 4
90    END
100   DATA 146,162,232,232,162,146
```

You can use the graphics facility to draw patterns on the printer as demonstrated by this program:

```
10    OPEN 4,4
20    FOR N=0 TO 30
30    FOR R=0 TO 6
40    D$=CHR$((2↑R)+128)
50    PRINT#4,CHR$(8)D$
60    NEXT R
70    FOR R=6 TO 0 STEP -1
80    D$=CHR$((2↑R)+128)
90    PRINT#4,CHR$(8)D$;
100   NEXT R
110   NEXT N
120   CLOSE 4
```

Tab Print Head

You can move the printer head under program control to any position using CHR$(16). A two

digit number following the CHR$(16) specifies the position between 0 and 79. The next program shows this control character in use.

```
10    REM PRINT HEAD DEMO
20    OPEN 4,4
30    FOR I=10 TO 70 STEP 5
40    P$=STR$(I)
50    H$=MID$(P$,2,1)
60    L$=RIGHT$(P$,1)
70    PRINT#4,CHR$(16)H$L$"POS"P$
80    NEXT
90    CLOSE 4
```

Reverse On / Reverse Off

Reverse text can be printed by preceding the text by CHR$(18). Turn off reverse text with CHR$(146).

```
10    OPEN 4,4
20    A$="REVERSED TEXT"
30    FOR I=1 TO LEN(A$)
40    R=1-R
50    IF R=0 THEN R$=CHR$(146):GOTO
      70
60    R$=CHR$(18)
70    PRINT#4,R$;MID$(A$,I,1);
80    NEXT I
90    CLOSE 4
100   END
```

Double Width Characters

You can highlight important information by printing the text using double width characters. Double width characters have the same dot pattern as standard characters but being twice as wide, only half as many will fit on a line. To print double width characters, precede the text with CHR$(14).

```
10    REM DOUBLE WIDTH CHARACETRS
20    OPEN 4,4
30    PRINT#4,"80 COLUMNS OF TEXT
      THIS SIZE"
40    PRINT#4,CHR$(14)"OR 40 COLUMNS
      OF THIS SIZE"
50    CLOSE 4:END
```

Line Feeds and Carriage Returns

You can control the printer to some extent by sending line feed characters (CHR$(10)) and Carriage return characters (CHR$(13)). These will allow you to create blank lines in your printouts:

```
10    OPEN 4,4
20    PRINT#4,"LINE 1"
30    FOR Z=0 TO 9
40    PRINT#4,CHR$(10)
50    NEXT Z
60    PRINT#4,"LINE 12"
```

Specify Dot Address

The span of the print head can be divided up into 480 columns each one dot wide. CHR$(27) CHR$(16) allows you to select one of these positions to commence printing data. Since there are 480 possible positions the dot address is specified in two bytes which follow the CHR$(27)CHR$(16) like this:

```
PRINT#4,CHR$(27)CHR$(16)CHR$(1)
CHR$(44)
```

specifies dot address 1*256+44=300 and moves the print head to that position.

Repeat Graphic Data

CHR$(26) allows you to repeat a byte of data in graphics mode a specified number of times. For example, if you wanted to draw a thick line across the paper, you would need to repeat a single vertical line many times.

The following program shows how you might do this, using each pin of the print head to create a vertical line character like this: |

```
10      OPEN 4,4
20      PRINT#4,CHR$(8)CHR$(26)CHR$
        (100)CHR$(255)
30      CLOSE 4
```

The single character CHR$(255) is repeated at each dot address across the paper, for 100 times - specified by CHR$(100) - in this example. You could repeat for any number of times between 0 and 255, but to cover the entire width of the paper would require two commands.

You can use CHR$(26) to underline headings as the next program shows:

```
10      OPEN 4,4:PRINT#4,CHR$(15)
20      PRINT#4,TAB(30)"A CENTRED
        HEADING"
30      PRINT#4,TAB(29)CHR$(8)CHR$
        (26)CHR$(114)CHR$(129)
40      CLOSE 4:END
```

With a different choice of character you could make the underlining more bold.

Screen Dump

The printer manual contains a program to output the contents of a standard screen to the printer, but

it will not print reverse video characters. The following program rectifies that omission and performs the same operation in rather less space.

```
60000 OPEN 4,4:PRINT#4,CHR$(15)
60010 FOR A=1024 TO 2023 STEP 40
60020 FOR X=0 TO 39
60040 D=PEEK(X+A)
60050 IF D>127 THEN D=D-128:
      R$=CHR$(18):O$=CHR$(146)
60060 P=D-((D<32 OR D>95)*64)-((D>63
      AND D<96)*32)
60070 P$=P$+R$+CHR$(P)+O$
60080 R$="":O$=""
60090 NEXT X
60100 PRINT#4,P$
60110 P$="":NEXT A
60120 CLOSE 4:END
```

In order to obtain hardcopy of screens containing upper and lower case characters, line 60000 must be changed to :

```
60000 OPEN 4,4,7:PRINT#4,CHR$(145)
```

Using the Printer with Machine Code

There are occasions when you might need to control the printer from within a machine code program – for example in a word processing package where you want to dump the contents of an area of memory onto the printer.

This can be achieved quite easily with the use of several of the kernal routines. A full listing of these is given in Appendix 6 and an explanation of them in Chapter 2.

The following program will output the contents of a small block of memory to the printer. The last character in the block is CHR$(13) – a carriage

return which causes the printer buffer to be emptied – ensuring that all the data is printed.

```
 10          ;****************
 20          ;* PRINTER DEMO *
 30          ;****************
 40          ;
 50 ORG $C000            start address
 60 !
 70          LDA #0
 80          JSR SETNAM  0 = no filename
 90          LDA #4      LFn = 4
100          TAX         Device No. = 4
110          LDY #255    No Secondary Add
120          JSR SETLFS  Set up Logical File
130          JSR OPEN    Open file
140          LDX #4      LFn = 4
150          JSR CHKOUT  Open channel for
                         output
160          LDX #0      Initialise counter
170 OP       LDA TAB,X   Get character from
                         table
180          JSR CHROUT  Output to printer
190          INX             Increment counter
200          CPX #34     Last item in table?
210          BNE OP      No, then get next
                         character
220          JSR CLALL   All done so close all
                         channels
230          RTS         Back to BASIC
240 TAB      BYT 80,82,73,78,84,69,
             68,32,70,82,79,77,32,77,
             65,67,72,73,78,69
250          BYT 32,67,79,68,69,32,
             80,82,79,71,82,65,77,13
270 SETNAM = 65469
280 SETLFS = 65466
290 OPEN   = 65472
300 CHKOUT = 65481
310 CHROUT = 65490
```

```
320 CLALL  = 65511
```

The program will output the data in the table to the printer - displaying the message 'PRINTED FROM MACHINE CODE PROGRAM'. The data in the table could contain control characters to change the output from the printer.

The program could be easily modified to dump the contents of any area of memory to the printer by increasing the size of the loop labelled OP.

The following is a BASIC program to load the machine code. The code is relocatable - it can be placed anywhere in memory. To change the start address you must change the value of S in line 10.

```
1      REM LOADER FOR PRINTER M/C
10     S=49152
20     FOR Z=S TO S+71
30     READ D:POKE Z,D
40     NEXT Z
50     END
20000  DATA 169,0,32,189,255,169,4,
       170,160,255,32,186,255,32,192,
       255
20010  DATA 162,4,32,201,255,162,0,
       189,38,192,32,210,255,232,224,
       34
20020  DATA 208,245,32,231,255,96,80,
       82,73,78,84,69,68,32,70,82
20030  DATA 79,77,32,77,65,67,72,73,
       78,69,32,67,79,68,69,32
20040  DATA 80,82,79,71,82,65,77,13
```

APPENDIX 1

ABBREVIATIONS

LISTING	MEANING	KEYS TO PRESS
{BLK}	BLACK	CTRL and 1
{WHT}	WHITE	CTRL and 2
{RED}	RED	CTRL and 3
{CYN}	CYAN	CTRL and 4
{PUR}	PURPLE	CTRL and 5
{GRN}	GREEN	CTRL and 6
{BLU}	BLUE	CTRL and 7
{YEL}	YELLOW	CTRL and 8
{ORG}	ORANGE	CBM and 1
{BRN}	BROWN	CBM and 2
{LTRED}	LIGHT RED	CBM and 3
{DKGRY}	DARK GREY	CBM and 4
{GRY}	GREY	CBM and 5
{LTGRN}	LIGHT GREEN	CBM and 6
{LTBLU}	LIGHT BLUE	CBM and 7
{LTGRY}	LIGHT GREY	CBM and 8
{RVS}	REVERSE VIDEO ON	CTRL and 9
{ROF}	REVERSE VIDEO OFF	CTRL and 0
{CLS}	CLEAR SCREEN	SHIFT and CLR / HOME
{HOM}	CURSOR HOME	CLR / HOME
{CU}	CURSOR UP	SHIFT and ↑CRSR↓
{CD}	CURSOR DOWN	↑CRSR↓
{CL}	CURSOR LEFT	SHIFT and CRSR ⇆
{CR}	CURSOR RIGHT	CRSR ⇆

APPENDIX 2

DOS ERROR MESSAGES

20 READ ERROR

The DOS has been unable to read a block of data – either an illegal track and sector have been requested or the disk has been corrupted or has been protected against copying.

21 READ ERROR

The sync byte of the requested track cannot be read – either an unformatted disk or one formatted under another DOS is present, the disk isn't inserted in the drive correctly, or the disk drive unit requires servicing to align the head.

22 READ ERROR

An incorrectly written block has been encountered – an illegal track and sector have been specified.

23 READ ERROR

A checksum error has occured – one or more of the bytes in a block has been read incorrectly, causing the checksum to fail.

24 READ ERROR

The DOS has decoded the data incorrectly – possibly owing to poor electrical connections within the drive or between the drive and the 64.

25 WRITE ERROR

The data block written to disk has been checked against that in DOS memory and been found to be incorrectly written.

26 WRITE PROTECT ON

The DOS has detected the presence of a write protect tab over the write protect notch.

27 READ ERROR

A checksum error in the header for a data block has been detected and the block has not been transferred to DOS memory – possibly due to poor earthing connections.

28 WRITE ERROR

A time-out error has occurred as the DOS tried to locate the sync byte for a data block – caused by bad disk format or hardware failure.

29 DISK ID MISMATCH

The DOS has detected a non initialised disk.

30 SYNTAX ERROR

The command sent to DOS via Channel 15 is illegal.

31 SYNTAX ERROR

The command sent to DOS is invalid.

32 SYNTAX ERROR

The command sent to DOS contains more than 58 characters.

33 SYNTAX ERROR

The file name in a command is invalid.

34 SYNTAX ERROR

Either no file name was sent or the syntax of the command was incorrect causing the DOS to misinterpret the command.

39 SYNTAX ERROR

An unrecognisable command has been sent to DOS.

50 RECORD NOT PRESENT

The file pointer has been positioned at a point after the last record of a relative file on the disk. This doesn't constitute an error if data is to be written (i.e. if a new record is being created).

51 OVERFLOW IN RECORD

An attempt has been made to write too much data to a record in a relative file. Remember that the carriage return terminator counts as one character.

52 FILE TOO LARGE

The file pointer has been positioned to a point where a disk overflow will occur if data is written.

60 WRITE FILE OPEN

An attempt has been made to open a file for reading before it has been closed after a write operation.

61 FILE NOT OPEN

An attempt has been made to access a file that has not been opened.

62 FILE NOT FOUND

An attempt has been made to access a non-existent file.

63 FILE EXISTS

An attempt has been made to save a file under a name which is already in the directory.

64 FILE TYPE MISMATCH

The file type in a DOS command differs from the type given in the directory for that file.

65 NO BLOCK

An attempt has been made to allocate a block which is unavailable according to the BAM. The parameters returned with this message define the track and sector numbers of the next free higher numbered block.

66 ILLEGAL TRACK AND SECTOR

An attempt has been made to access an illegal track and sector.

67 ILLEGAL SYSTEM TRACK OR SECTOR

An attempt has been made to access an illegal system track or sector.

70 NO CHANNEL

The requested channel is not available, or all channels are in use.

71 DIRECTORY ERROR

A discrepancy exists between the BAM count and the directory – possibly caused by overwriting the BAM. Re-initialise the disk to force DOS to re-create the BAM.

72 DISK FULL

Either no blocks are available or the maximum of 144 directory entries has been reached.

73 DOS MISMATCH

An attempt has been made to write to a disk formatted under a non-compatible DOS.

74 DRIVE NOT READY

No disk is present in the drive.

APPENDIX 3

SUMMARY OF DOS COMMANDS

COMMAND	FORMAT
NEW	N: DISK NAME, ID
COPY	C: NEW FILE = 0: OLD FILE
RENAME	R: NEW NAME = OLD NAME
SCRATCH	S: FILE NAME
INITIALISE	I
VALIDATE	V
BLOCK-READ	"B-R:"Channel;Drive;Track;Block
BLOCK-WRITE	"B-W:"Channel;Drive;Track;Block
BLOCKALLOCATE	"B-A:"Drive;Track;Block
BLOCK FREE	"B-F:"Drive;Track;Block
BUFFER POINTER	"B-P:"Channel;Position
POSITION	"P"CHR$(Chan)CHR$(Low)CHR$(Hi)CHR$(Pos)
BLOCK-EXECUTE	"B-E:"Channel;Drive;Track;Block
MEMORY-READ	"M-R:"CHR$(Low)CHR$(High)
MEMORY-WRITE	"M-W:"CHR$(Low)CHR$(High)CHR$(no. of Chars)
USER	"Un:"

APPENDIX 4

BASIC TOKENS & ABBREVIATIONS

COMMAND	ABBREVIATION	DEC. TOKEN	HEX TOKEN
ABS	aB	182	B6
AND	aN	175	AF
ASC	aS	198	C6
ATN	aT	193	C1
CHR$	cH	199	C7
CLOSE	clO	160	A0
CLR	cL	156	9C
CMD	cM	157	9D
CONT	cO	154	9A
COS	---	190	BE
DATA	dA	131	83
DEF	dE	150	96
DIM	dI	134	86
END	eN	128	80
EXP	eX	189	BD
FN	---	165	A5
FOR	fO	129	81
FRE	fR	184	B8
GET	gE	161	A1
GET#	---	---	---
GOSUB	goS	141	8D
GOTO	gO	137	89
IF	---	139	8B
INPUT	---	133	85

INPUT#	iN	132	84
INT	---	181	B5
LEFT$	leF	200	C8
LEN	---	195	C3
LET	lE	136	88
LIST	ll	155	9B
LOAD	lO	147	93
LOG	---	188	BC
MID$	ml	202	CA
NEW	---	162	A2
NEXT	nE	130	82
NOT	nO	168	A8
ON	---	145	91
OPEN	oP	159	9F
OR	---	176	B0
PEEK	pE	194	C1
POKE	pO	151	97
POS	---	185	B9
PRINT	?	153	99
PRINT#	pR	152	98
READ	rE	135	87
REM	---	143	8F
RESTORE	reS	140	8C
RETURN	reT	142	8E
RIGHT$	rI	201	C9
RND	rN	187	BB
RUN	rU	138	8A
SAVE	sA	148	94
SGN	sG	180	B4
SIN	sI	191	BF
SPC	sP	166	A6

SQR	sQ	186	BA
STATUS	ST	---	---
STEP	stE	169	A9
STOP	sT	144	90
STR$	stR	196	C4
SYS	sY	158	9E
TAB	tA	163	A3
TAN	---	192	C0
THEN	tH	167	A7
TIME	TI	---	---
TIME$	TI$	---	---
TO	---	164	A4
USR	uS	183	B7
VAL	vA	197	C5
VERIFY	vE	149	95
WAIT	wA	146	92
+	---	170	AA
-	---	171	AB
*	---	172	AC
/	---	173	AD
=	---	178	B2

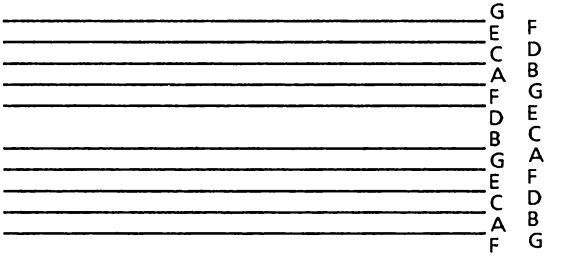
NOTE: --- indicates either no abbreviation or a non-tokenised keyword

APPENDIX 5

MUSICAL NOTATION

This appendix will not teach you all about music, but it contains the basic information you need to translate sheet music into 64 programs.

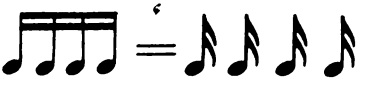
Music is written by positioning symbols which represent the length of notes on a framework (called a stave) representing the pitch.



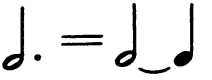
The lengths of notes are indicated by the note shape:

A Semibreve		is twice as long as
a Minim		which is twice as long as
a Crotchet		which is twice as long as
a Quaver		which is twice as long as
a Semiquaver		

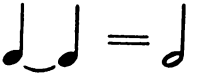
Tails on notes may go up  or down  . The feathers on quavers and shorter notes may be joined where they appear in groups:



A dot after a note means that it is made half as long again as a normal note:



The mark  is a tie which means the notes are joined together.



Volume is indicated by markings below the stave.

ff	Very loud
f	Loud
mf	Moderately loud
mp	Moderately soft
p	Soft
pp	Very soft
	Get louder
	Get softer

Speed is indicated by markings which may be above or below the stave. Examples are:

Presto	which means Fast
Allegro	Quite fast
Allegro moderato	Moderately fast
Moderato	Medium pace
Andante	Slow
Largo	Very slow
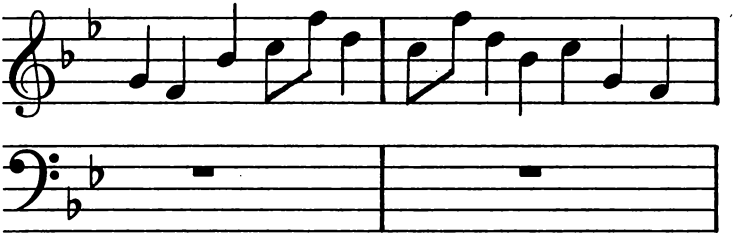
Unfortunately, there are many other Italian words and phrases which may be used. The best thing to do is to adjust your program until the speed sounds right, and not worry too much about what is written on the music.

Two other markings which may appear next to notes are # and b. # (sharp) means that the note should be raised by one semitone and b (flat) means that the note is lowered one semitone.

All other markings which may appear on sheet music (and there are many of them) can be ignored.

Pictures at an Exhibition

Here is the beginning of *Pictures at an Exhibition*, one of the examples used in Chapter 7. The notes



shown in this music correspond to the notes represented by the first two items of every group of six in the **DATA** statements in the program.

Note the 'flat' symbols (b) on the lines of the stave corresponding to the notes B and E. These mean that all B s and E s throughout the music are sharpened.

APPENDIX 6

KERNAL ROUTINES

ACPTR $FFA5 65445

Inputs a byte from the serial bus into the accumulator. Before use call TALK and TKSA.

CHKIN $FFC6 65478

Selects a previously opened logical file to be an input channel.

CHKOUT $FFC9 65481

Selects a previously opened logical file to be an output channel.

CHRIN $FFCF 65487

Gets a character from a previously opened input channel into the accumulator. If no channel is opened, the keyboard is assumed to be the input device.

CHROUT $FFD2 65490

Sends the character in the accumulator to a previously opened channel. Channel must have been opened by OPEN and CHKOUT routines.

CIOUT $FFA8 65488

Sends the character in the accumulator to a serial bus device. The LISTEN and SECOND routines

must have been called to prepare the device for the data.

CINT $FF81 65409

Initialises VIC-II chip and screen editor at the start of a program.

CLALL $FFE7 65511

Closes all open files and resets all I/O chanels.

CLOSE $FFC3 65475

Closes the logical file whose number is held in the accumulator.

CLRCHN $FFCC 65484

Resets all input/output channels, called by CLALL.

GETIN $FFEA 65508

Either loads one character from the keyboard buffer into the accumulator (accumulator contains 0 if buffer is empty), or gets a character from a serial device.

IOBASE $FFF3 65523

Loads the address of the start of memory mapped input/output into the X (low byte) and Y (high byte) registers.

IOINIT $FF84 65412

Initialises all input/output devices and routines.

LISTEN $FFB1 65457

Commands the serial bus device specified by number in the accumulator to listen (be ready to accept data).

LOAD $FFD5 65493

Loads data from a device into memory. If acumulator = 0 then memory is overwritten by new data; a 1 specifies a verify operation, the result of which is returned in ST. SETLFS and SETNAM routines must be used before this routine.

MEMBOT $FF9C 65436

If carry bit is set, returns address of lowest byte of RAM in X and Y registers. If carry bit is cleared before calling, the contents of X and Y specify lowest RAM address.

MEMTOP $FF99 65433

If carry bit is set, returns address of the highest byte of RAM in X and Y. If carry bit is cleared before calling, pointer to highest byte of RAM is set to contents of X and Y registers.

OPEN $FFC0 65472

Opens the logical file set up by SETLFS and SETNAM routines.

PLOT $FFF0 65520

If carry flag is set when called, PLOT loads current cursor position into X and Y registers. If carry flag is cleared before calling, cursor position is specified by contents of X and Y registers.

RAMTAS $FF87 65415

Performs RAM test, sets top and bottom of memory pointers, sets up cassette buffer and initialises screen.

RDTIM $FFDE 65502

Reads the high, middle and low bytes of the system clock into accumulator, X and Y registers respectively.

READST $FFB7 65463

Returns system status word in the accumulator.

RESTOR $FF8A 65418

Restores default values of all system vectors.

SAVE $FFD8 65496

Saves the contents of memory to a device which has previously been setup by SETLFS and SETNAM routines. When called the accumulator must contain an offset to a two byte pointer (stored in page zero) to the start of memory to be saved. The X and Y registers contain the address of the last byte to be saved.

SCNKEY $FF9F 65439

Scans keyboard for key presses. ASCII value of a key is stored in keyboard buffer.

SCREEN $FFED 65517

Returns number of screen columns and rows in X and Y registers.

SECOND $FF93 65427

Sets secondary address for an input/output device in a LISTEN operation.

SETLFS $FFBA 65466

Sets up a logical file, whose number is in the accumulator, device number is in the X register and command is in the Y register.

SETMSG $FF90 65424

Controls the output of error and control messages. If Bit 7 of the accumulator is set, an error message will be output; if Bit 6 is set a control message will be output.

SETNAM $FFBD 65469

Sets up a file name whose length is in the Accumulator. The X and Y registers must contain the start address of the file name in low-high byte order.

SETTIM $FFDB 65499

Sets jiffy clock to the time specified in the accumulator, X and Y registers.

SETTMO $FFA2 65442

Sets timeout flag when an IEEE card is connected.

STOP $FFE1 65505

Tests for STOP key being pressed after calling UDTIM routine. If STOP key was pressed the zero flag will be set.

TALK $FFB4 65460

Commands the serial device whose number is in the accumulator to talk.

TKSA $FF96 65430

Sends a secondary address to a serial device after the TALK routine.

UDTIM $FFEA 65514

Updates the system clock when interrupt routines have been modified.

UNLSN $FFAE 65454

Commands all serial devices to stop receiving data.

UNTALK $FFAB 65451

Commands all serial devices to stop sending data.

VECTOR $FF8D 65421

If carry bit is set when this routine is called, the current contents of the system vector jump addresses are stored in a table whose start address is defined by the X and Y registers. If carry is cleared before calling, the contents of the table pointed to by the X and Y registers are transferred to the system vectors.

APPENDIX 7

GRAPHICS LOADER PROGRAMS

This Appendix contains BASIC listings of loader programs for all the machine code graphics package programs described in Chapters 3 and 4. Refer to these chapters for details of how the programs are used.

How to Use the Loader Programs

The machine code has been split into 5 programs to make it easier for you to enter it in small stages. Because the routines are divided you do not have to type them all in at once – which will reduce the chances of error.

To obtain the machine code for the complete graphics package follow these instructions:

1 Type in the BIT-MAP routine loader, **SAVE** and **RUN** it.

2 Save the machine code by following the instructions at the end of the BIT-MAP routine.

3 Type in the PLOT routine loader, **SAVE** and **RUN** it.

4 Save the machine code by following the instructions at the end of the PLOT routine.

5 Type in the FILL routine loader, **SAVE** and **RUN** it.

6 Save the machine code by following the instructions at the end of the FILL routine.

7 Type in the DRAW routine loader , **SAVE** and **RUN** it.

8 Save the machine code by following the instructions at the end of the DRAW routine.

9 Type in the MIX MODE routine loader , **SAVE** and **RUN** it.

10 Save the machine code by following the instructions at the end of the MIX MODE routine.

11 When all five machine code routines have been saved, you can combine them into a single program by loading each one like this:

```
LOAD "PROG",8,1
```

for a program on disk, or:

```
LOAD "PROG",1,1
```

for a program on tape.

12 When all five routines have been loaded in this way, you can save them as one program like this:

```
POKE 44,192:POKE 43,0

POKE 46,197;POKE 45,110

SAVE "GRAPHICS MC",8:REM OR ,1 FOR
TAPE
```

13 To load the graphics package subsequently, use the command:

```
LOAD "GRAPHICS MC",8,1
```

to load from disk, or:

```
LOAD "GRAPHICS MC",1,1
```

to load from tape.

BIT-MAP ROUTINE

```
10    REM BIT MAP LOADER
20    FOR X=49152 TO 49331
30    READ D:POKE X,D
40    NEXT X
50    PRINT"BIT MAP CODE LOADED"
60    END
10000 DATA 169,1,141,177,2,208,5,
      169,0,141,177,2,169,96,133,
      <252,169
10010 DATA 0,133,251,162,64,32,154,
      192,169,92,133,252,169,0,133,
      251,173
10020 DATA 174,2,10,10,10,10,13,175,
      2,162,8,32,154,192,173,177,2
10030 DATA 208,16,169,216,133,252,
      169,0,133,251,162,8,173,176,
      2,32,154
10040 DATA 192,173,2,221,9,3,141,2,
      221,173,0,221,41,252,9,2,141
10050 DATA 0,221,173,17,208,9,32,
      141,17,208,169,120,141,24,208,
      173,177
10060 DATA 2,208,8,173,22,208,9,16,
      141,22,208,96,173,2,221,9,3
10070 DATA 141,2,221,173,0,221,41,
      252,9,3,141,0,221,173,17,
      208,41
```

```
10080 DATA 223,141,17,208,173,22,
      208,41,239,141,22,208,169,21,
      141,24,208
10090 DATA 96,160,127,145,251,136,
      16,251,72,24,165,251,105,128,
      133,251,169
10100 DATA 0,101,252,133,252,104,
      202,208,231,96
```

Saving the Bit-Map Routine machine code

After runnning the loader program type in the following commands to save the machine code:

```
POKE 44,192:POKE 43,0

POKE 46,192:POKE 45,180

SAVE "BIT MAP":REM ADD ,8 FOR DISK
DRIVE
```

PLOT ROUTINE

```
10    REM PLOT LOADER
20    FOR X=49336 TO 49593
30    READ D:POKE X,D
40    NEXT X
50    PRINT"PLOT CODE LOADED"
60    END
11000 DATA 173,169,2,74,74,74,141,
      186,193,173,168,2,74,173,167,
      2,106
11010 DATA 74,174,177,2,240,1,74,
      141,187,193,173,169,2,41,7,
      141,188
11020 DATA 193,173,186,193,133,253,
      169,0,133,254,162,6,32,175,
      193,202,208
11030 DATA 250,165,254,133,252,165,
      253,133,251,32,175,193,32,175,
      193,24,165
```

```
11040 DATA 253,101,251,133,251,165,
      254,101,252,133,252,169,0,133,
      254,173,187
11050 DATA 193,133,253,32,175,193,
      32,175,193,32,175,193,24,165,
      253,101,251
11060 DATA 133,251,165,254,101,252,
      133,252,24,173,188,193,101,
      251,133,251,169
11070 DATA 0,101,252,133,252,24,169,
      0,101,251,133,251,169,96,101,
      252,133
11080 DATA 252,173,177,2,240,48,173,
      167,2,41,7,141,189,193,56,
      169,7
11090 DATA 237,189,193,141,189,193,
      24,169,1,174,189,193,240,4,10,
      202,208
11100 DATA 252,160,0,174,173,2,240,
      5,17,251,145,251,96,73,255,
      49,251
11110 DATA 145,251,96,173,167,2,41,
      3,141,189,193,56,169,3,237,
      189,193
11120 DATA 10,141,189,193,160,0,173,
      173,2,41,3,174,189,193,240,
      4,10
11130 DATA 202,208,252,141,190,193,
      169,252,174,189,193,240,5,56,
      42,202,208
11140 DATA 252,49,251,13,190,193,
      145,251,96,169,0,6,254,6,253,
      101,254
11150 DATA 133,254,96
```

Saving the Plot Routine machine code

After runnning the loader program, type in the following commands to save the machine code:

```
POKE 44,192:POKE 43,184

POKE 46,193:POKE 45,188

SAVE "PLOT":REM ADD ,8 FOR DISK
DRIVE
```

FILL ROUTINE

```
10     REM FILL LOADER
20     FOR X=50402 TO 50466
30     READ D:POKE X,D
40     NEXT X
50     PRINT "FILL CODE LOADED"
60     END
13000 DATA 173,169,2,141,35,197,32,
       184,192,24,173,169,2,105,1,
       141,169
13010 DATA 2,173,172,2,205,169,2,
       176,236,173,35,197,141,169,2,
       24,173
13020 DATA 167,2,105,1,141,167,2,
       173,168,2,105,0,141,168,2,
       56,173
13030 DATA 171,2,237,168,2,173,170,
       2,237,167,2,176,198,96
```

Saving the Fill Routine Machine Code

After runnning the loader program, type in the following commands to save the machine code:

```
POKE 44,196:POKE 43,226

POKE 46,197:POKE 45,35

SAVE "FILL":REM ADD ,8 FOR DISK
DRIVE
```

DRAW ROUTINE

```
10     REM DRAW LOADER
20     FOR X=49608 TO 50372
30     READ D:POKE X,D
40     NEXT X
50     PRINT"DRAW CODE LOADED"
60     END
12000 DATA 173,170,2,141,197,196,
       173,171,2,141,198,196,173,169,
       2,141,199
12010 DATA 196,169,0,141,224,196,56,
       173,172,2,237,169,2,176,15,
       173,224
12020 DATA 196,73,1,141,224,196,56,
       173,169,2,237,172,2,141,202,
       196,56
12030 DATA 173,170,2,237,167,2,141,
       200,196,173,171,2,237,168,2,
       176,24
12040 DATA 173,224,196,73,1,141,224,
       196,56,173,167,2,237,170,2,
       141,200
12050 DATA 196,173,168,2,237,171,2,
       141,201,196,208,10,173,200,
       196,208,5
12060 DATA 169,1,141,224,196,173,
       202,196,208,5,169,1,141,224,
       196,173,201
12070 DATA 196,208,11,173,200,196,
       205,202,196,176,3,76,10,195,
       56,173,170
12080 DATA 2,237,167,2,173,171,2,
       237,168,2,176,3,32,160,196,
       173,202
12090 DATA 196,141,205,196,169,0,
       141,206,196,173,200,196,141,
       203,196,173,201
12100 DATA 196,141,204,196,32,51,
       196,173,167,2,141,208,196,173,
       168,2,141
```

```
12110 DATA 209,196,32,200,195,141,
      212,196,173,223,196,141,213,
      196,174,224,196
12120 DATA 240,21,24,173,212,196,
      109,169,2,141,212,196,173,213,
      196,105,0
12130 DATA 141,213,196,76,188,194,
      173,169,2,237,212,196,141,212,
      196,169,0
12140 DATA 237,213,196,141,213,196,
      173,168,2,141,209,196,173,167,
      2,141,208
12150 DATA 196,32,200,195,174,224,
      196,208,10,24,109,212,196,141,
      169,2,76
12160 DATA 228,194,56,173,212,196,
      237,222,196,141,169,2,32,184,
      192,24,173
12170 DATA 167,2,105,1,141,167,2,
      173,168,2,105,0,141,168,2,56,
      173
12180 DATA 170,2,237,167,2,173,171,
      2,237,168,2,176,181,76,181,
      195,56
12190 DATA 173,172,2,205,169,2,176,
      3,32,160,196,173,200,196,141,
      205,196
12200 DATA 173,201,196,141,206,196,
      173,202,196,141,203,196,169,0,
      141,204,196
12210 DATA 32,51,196,173,169,2,141,
      208,196,169,0,141,209,196,32,
      200,195
12220 DATA 141,212,196,174,224,196,
      240,18,24,109,167,2,141,212,
      196,169,0
12230 DATA 109,168,2,141,213,196,76,
      105,195,173,167,2,237,212,196,
      141,212
```

```
12240 DATA 196,173,168,2,233,0,141,
      213,196,173,169,2,141,208,196,
      169,0
12250 DATA 141,209,196,32,200,195,
      174,224,196,208,19,24,109,212,
      196,141,167
12260 DATA 2,173,213,196,109,223,
      196,141,168,2,76,162,195,56,
      173,212,196
12270 DATA 237,222,196,141,167,2,
      173,213,196,237,223,196,141,
      168,2,32,184
12280 DATA 192,24,173,169,2,105,1,
      141,169,2,205,172,2,144,182,
      240,180
12290 DATA 173,198,196,141,168,2,
      173,197,196,141,167,2,173,199,
      196,141,169
12300 DATA 2,96,173,214,196,141,217,
      196,173,215,196,141,218,196,
      173,216,196
12310 DATA 141,219,196,169,0,141,
      220,196,141,221,196,141,222,
      196,141,223,196
12320 DATA 141,210,196,141,211,196,
      160,24,78,219,196,110,218,196,
      110,217,196
12330 DATA 144,37,24,173,208,196,
      109,220,196,141,220,196,173,
      209,196,109,221
12340 DATA 196,141,221,196,173,210,
      196,109,222,196,141,222,196,
      173,211,196,109
12350 DATA 223,196,141,223,196,14,
      208,196,46,209,196,46,210,196,
      46,211,196
12360 DATA 136,208,193,173,222,196,
      96,169,0,141,216,196,141,215,
      196,141,214
```

```
12370 DATA 196,141,207,196,160,24,
      56,173,206,196,237,203,196,
      141,206,196,173
12380 DATA 207,196,237,204,196,141,
      207,196,8,46,214,196,46,215,
      196,46,216
12390 DATA 196,14,205,196,46,206,
      196,46,207,196,40,144,21,173,
      206,196,237
12400 DATA 203,196,141,206,196,173,
      207,196,237,204,196,141,207,
      196,184,80,18
12410 DATA 173,206,196,109,203,196,
      141,206,196,173,207,196,109,
      204,196,141,207
12420 DATA 196,136,208,192,46,214,
      196,46,215,196,46,216,196,96,
      173,170,2
12430 DATA 172,167,2,141,167,2,140,
      170,2,173,171,2,172,168,2,141,
      168
12440 DATA 2,140,171,2,173,172,2,
      172,169,2,141,169,2,140,172,
      2,96
```

Saving the DRAW routine machine code

After runnning the loader program, type in the following commands to save the machine code:

```
POKE 44,193:POKE 43,200

POKE 46,196:POKE 45,197

SAVE "DRAW":REM ADD ,8 FOR DISK
DRIVE
```

MIXED MODE LOADER

```
10    REM MIX MODE LOADER
20    FOR X=50468 TO 50541
```

```
30     READ D:POKE X,D
40     NEXT X
50     PRINT "MIX MODE CODE LOADED"
60     END
14000 DATA 120,169,67,141,20,3,169,
      197,141,21,3,169,1,141,26,208,
      169
14010 DATA 0,141,18,208,173,17,208,
      41,127,141,17,208,88,96,173,
      25,208
14020 DATA 41,1,240,33,141,25,208,
      173,18,208,208,11,32,69,192,
      169,200
14030 DATA 141,18,208,76,101,197,32,
      114,192,169,0,141,18,208,104,
      168,104
14040 DATA 170,104,64,76,49,234
```

Saving the MIXED MODE routine machine code

After runnning the loader program, type in the following commands to save the machine code:

```
POKE 44,197:POKE 43,36

POKE 46,197:POKE 45,110

SAVE "MIX MODE":REM ADD ,8 FOR DISK
DRIVE
```

SAVING THE GRAPHICS PACKAGE

After creating the machine code for each of the five routines comprising the graphics package and saving them individually, you can load them all into the 64 using the commands:

```
LOAD"PROG",8,1:REM FOR DISK
```

```
LOAD"PROG",1,1:REM FOR TAPE
```

With all the routines in the machine you can save them together as one program by typing:

```
POKE 44,192:POKE 43,0

POKE 46,197:POKE 45,110

SAVE "GRAPHICS MC",8
```

DATAMAKER

This program will convert the contents of any area of memory into a series of **DATA** statements which are appended to the DATAMAKER program. To use the program enter the line number at which you want the **DATA** to start, and the start and end address of the area of memory you want to convert into **DATA** statements. When the program has finished, delete lines 5 to 500 and save the **DATA** statements for use in your program.

```
5     REM DATAMAKER
10    INPUT "FIRST DATA LINE
      NUMBER";LN
20    INPUT "START ADDRESS OF
      CODE";S
30    INPUT "END ADDRESS OF CODE";E
100   F=S+16:IF F>E THEN F=E
110   PRINT"{CLS}"LN"DATA";
120   FOR I=S TO F
130   C = PEEK(I)
140   C$ = MID$(STR$(C),2)
150   CK = CK+C
160   PRINT C$;
170   IF I<F THEN PRINT",";
200   NEXT
210   PRINT
300   S=F
```

```
310   IF S<E THEN PRINT"LN="LN+10":
      S="S+1":E="E":GOTO100"
320   IF S=E THEN PRINT"LIST"
330   POKE 198,3
340   POKE 631,19
350   POKE 632,13:POKE 633,13
500   END
```

APPENDIX 8

THE 6510 INSTRUCTION SET

ADC

Adds the contents of a memory location, or a number, to the accumulator, including the carry bit. Deposits the result in the accumulator.

AND

Performs the logical AND operation between the accumulator and data. Deposits the result in the accumulator.

ASL

Shifts the contents of the accumulator or memory left by one position. Bit 7 moves into the carry flag and Bit 0 is set to zero.

BCC

If the carry flag is clear program branches to current address plus a signed displacement (+127 to -128).

BCS

If the carry flag is set program branches to current address plus a signed displacement (+127 to -128).

BEQ

If the zero flag is set program branches to current address plus a signed displacement (+127 to -128).

BIT

Performs the logical AND operation between the accumulator and memory. If the comparison succeeds the zero flag is set and Bits 6 and 7 of the memory location are copied into the V and N flags.

BMI

If the N flag is set (the result of the last operation was negative) the program branches to the current address plus a signed displacement (+127 to -128).

BNE

If the zero flag is clear the program branches to current address plus a signed displacement (+127 to -128).

BPL

If the N flag is clear the program branches to current address plus a signed displacement (+127 to -128).

BRK

Saves the program counter and status register on the stack and copies the contents of $FFFE and $FFFF into PCLow and PCHigh.

BVC

If the overflow flag is clear the program branches to current address plus a signed displacement (+127 to -128).

BVS

If the overflow flag is set the program branches to current address plus a signed displacement (+127 to −128).

CLC

Clears the carry flag.

CLD

Clears the decimal flag.

CLI

Clears the interrupt mask to enable interrupts.

CLV

Clears the overflow flag.

CMP

Compares the accumulator contents with memory. Sets the Z flag if they are equal or clears it if not. The C flag is set if the contents of the memory location are greater than those of the accumulator.

CPX

Compares the X register with memory data .

CPY

Compares the Y register with memory data .

DEC

Decrements the specified memory location.

DEX

Decrements the X register.

DEY

Decrememts the Y register.

EOR

Performs the Exclusive OR operation between memory and the accumulator, storing the result in the accumulator.

INC

Increments the specified memory location.

INX

Increments the X register.

INY

Increments the Y register.

JMP

Program execution continues at the specified memory location.

JSR

Program execution continues at the subroutine commencing at the sepcified address.

LDA

Loads the accumulator with data.

LDX

Loads the X register with data.

LDY

Loads the Y register with data.

LSR

Shifts the contents of the accumulator or memory location right one position. Bit 7 is set to zero and Bit 0 transferred to the carry flag.

NOP

Performs no operation for two clock cycles.

ORA

Performs the logical OR operation between the accumulator and data.

PHA

Pushes the contents of the accumulator on to the stack.

PHP

Pushes the processor status register onto the stack.

PLA

Pulls the first item of data from the stack and loads it into the accumulator.

PLP

Pulls the first item of data from the stack and loads it into the processor status register.

ROL

Rotates the contents of the specified memory location one position to the left. The carry flag is transferred into Bit 0 and Bit 7 is moved into the carry flag.

ROR

Rotates the contents of the specified memory location one position to the right. The carry flag is transferred into Bit 7 and Bit 0 is moved into the carry flag.

RTI

Retrieves the status register and program counter from the stack and returns from an interrupt routine.

RTS

Restores and increments the program counter after a subroutine call and returns from the subroutine.

SBC

Subtracts data from the accumulator with a borrow and stores the result in the accumulator.

SEC

Sets the carry flag.

SED

Sets decimal mode.

SEI

Sets the interrupt disable mask.

STA

Stores the accumulator contents at a specified memory location.

STX

Stores the X register contents at a specified memory location.

STY

Stores the Y register contents at a specified memory location.

TAX

Transfers the accumulator contents to the X register.

TAY

Transfers the accumulator contents to the Y register.

TSX

Transfers the stack pointer contents to the X register.

TXA

Transfers the X register contents to the accumulator.

TXS

Transfers the X register contents to the stack pointer.

TYA

Transfers the Y register contents to the accumulator.

APPENDIX 9

SID REGISTERS

ADDRESS	7	6	5	4	3	2	1	0	REGISTER FUNCTION
									VOICE 1
54272	F 7	F 6	F 5	F 4	F 3	F 2	F 1	F 0	LOW FREQ
54273	F 15	F 14	F 13	F 12	F 11	F 10	F 9	F 8	HIGH FREQ
54274	PW 7	PW 6	PW 5	PW 4	PW 3	PW 2	PW 1	PW 0	PULSE WIDTH LOW
54275	-	-	-	-	PW11	PW10	PW 9	PW 8	PULSE WIDTH HIGH
54276	NOI	PUL	SAW	TRI	TEST	RING	SYNC	GATE	CONTROL REGISTER
54277	ATK 3	ATK 2	ATK 1	ATK 0	DEC 3	DEC 2	DEC 1	DEC 0	ATTACK / DECAY
54278	SUS 3	SUS 2	SUS 1	SUS 0	REL 3	REL 2	REL 1	REL 0	SUSTAIN / RELEASE
									VOICE 2
54279	F 7	F 6	F 5	F 4	F 3	F 2	F 1	F 0	LOW FREQ
54280	F 15	F 14	F 13	F 12	F 11	F 10	F 9	F 8	HIGH FREQ
54281	PW 7	PW 6	PW 5	PW 4	PW 3	PW 2	PW 1	PW 0	PULSE WIDTH LOW
54282	-	-	-	-	PW11	PW10	PW 9	PW 8	PULSE WIDTH HIGH
54283	NOI	PUL	SAW	TRI	TEST	RING	SYNC	GATE	CONTROL REGISTER
54284	ATK3	ATK2	ATK1	ATK0	DEC 3	DEC 2	DEC 1	DEC 0	ATTACK / DECAY
54285	SUS 3	SUS 2	SUS 1	SUS 0	REL3	REL 2	REL 1	REL 0	SUSTAIN / RELEASE

ADDRESS	7	6	5	4	3	2	1	0	REGISTER FUNCTION
									VOICE 3
54286	F 7	F 6	F 5	F 4	F 3	F 2	F 1	F 0	LOW FREQ
54287	F 15	F 14	F 13	F 12	F 11	F 10	F 9	F 8	HIGH FREQ
54288	PW 7	PW 6	PW 5	PW 4	PW 3	PW 2	PW 1	PW 0	PULSE WIDTH LOW
54289	-	-	-	-	PW11	PW10	PW 9	PW 8	PULSE WIDTH HIGH
54290	NOI	PUL	SAW	TRI	TEST	RING	SYNC	GATE	CONTROL REGISTER
54291	ATK 3	ATK 2	ATK 1	ATK 0	DEC 3	DEC 2	DEC 1	DEC 0	ATTACK / DECAY
54292	SUS 3	SUS 2	SUS 1	SUS 0	REL 3	REL 2	REL 1	REL 0	SUSTAIN / RELEASE
									FILTER
54293	-	-	-	-	-	FC 2	FC 1	FC 0	LOW FILTER
54294	FC 10	FC 9	FC 8	FC 7	FC 6	FC 5	FC 4	FC 3	HIGH FILTER
54295	RES 3	RES 2	RES 1	RES 0	FILTX	FILT 3	FILT 2	FILT 1	RESONANCE / FILTER
54296	3 OFF	HP	BP	LP	VOL 3	VOL 2	VOL 1	VOL 0	MODE / VOLUME
									MISCELLANEOUS
54297	PX 7	PX 6	PX 5	PX 4	PX 3	PX 2	PX 1	PX 0	POT X
54298	PY 7	PY 6	PY 5	PY 4	PY 3	PY 2	PY 1	PY 0	POT Y
54299	OSC 7	OSC 6	OSC 5	OSC 4	OSC 3	OSC 2	OSC 1	OSC 0	OSCIL 3 / RANDOM
54300	ENV 7	ENV 6	ENV 5	ENV 4	ENV 3	ENV 2	ENV 1	ENV 0	VOICE 3 ENVELOPE

APPENDIX 10

VIC-II REGISTERS

ADDRESS	FUNCTION
53248	SPRITE 0 X POSITION
53249	SPRITE 0 Y POSITION
53250	SPRITE 1 X POSITION
53251	SPRITE 1 Y POSITION
53252	SPRITE 2 X POSITION
53253	SPRITE 2 Y POSITION
53254	SPRITE 3 X POSITION
53255	SPRITE 3 Y POSITION
53256	SPRITE 4 X POSITION
53257	SPRITE 4 Y POSITION
53258	SPRITE 5 X POSITION
53259	SPRITE 5 Y POSITION
53260	SPRITE 6 X POSITION
53261	SPRITE 6 Y POSITION
53262	SPRITE 7 X POSITION
53263	SPRITE 7 Y POSITION
53264	SPRITE X POSITION MSB
53265	CONTROL REGISTER
53266	RASTER REGISTER
53267	LIGHT PEN X POSITION
53268	LIGHT PEN Y POSITION
53269	SPRITE ENABLE

53270	CONTROL REGISTER
53271	SPRITE Y EXPAND
53272	VIDEO MEMORY POINTERS
53273	INTERRUPT REGISTER
53274	ENABLE INTERRUPT
53275	SPRITE TO DATA PRIORITY
53276	SPRITE MULTI COLOUR
53277	SPRITE X EXPAND
53278	SPRITE - SPRITE COLLISION
53279	SPRITE - DATA COLLISION
53280	BORDER COLOUR
53281	BACKGROUND COLOUR 0
53282	BACKGROUND COLOUR 1
53283	BACKGROUND COLOUR 2
53284	BACKGROUND COLOUR 3
53285	SPRITE MULTICOLOUR 0
53286	SPRITE MULTICOLOUR 1
53287	SPRITE 0 COLOUR
53288	SPRITE 1 COLOUR
53289	SPRITE 2 COLOUR
53290	SPRITE 3 COLOUR
53291	SPRITE 4 COLOUR
53292	SPRITE 5 COLOUR
53293	SPRITE 6 COLOUR
53294	SPRITE 7 COLOUR

INDEX

1525 Printer 193
1540 Disk Drive 190
1541 Disk Drive 138
16 Sprites 97
6510 microprocessor 25

Absolute Addressing 29, 39
Accumulator 25
Adding commands 10
Addressing Modes 29, 39
Arithmetic in Machine Code 31
Array Element 22
Array Header 21
Array Variables 20
Assembler 27
Auto Loading 152

BAM 144
BASIC Storage 2,17
BEEP 11
Bit-Mapped Graphics 52
Block-Allocate 167
Block-Execute 188
Block-Free 168
Block-Read 164
Block-Write 165
Branching 35
Buffer Pointer 170

CHRGET Routine 9
Comparisons 38
COPY 149
Copying Tapes 192
Cursor Down Mode 197
Cursor Up Mode 197

Directory 143,150
Disk Drives 190
Disk Status 148, 152
Display Memory 53, 55
DOS 139
DOS Support Program 149
Dot Address 202
Dot Matrix 193
Double Width Mode 201
DRAW routine 71
Drawing Lines 70

Error status 151
Executing programs 9

Fill Program 82
Floating Point Variables 17
Floppy Disks 139
Formatting 142

Graphics Mode Printing 198
Graphics Programs 84, 227

Headache program 87
High level language 24
Homebase Program 173
Housekeeping 147

IEEE Disk Drives 191
Immediate Addressing 29
Index 250
Index Registers 25
Indexed Addressing 40
Indirect Addressing 40
Initialise 148
Instruction set 27
Integer Variables 16
Interpreter 1
Interrupts 45,91,104
Interrupt Enable Reg 93

Interrupt Status Reg. 93
JR's hat program 89
Jumps 42
Kernal 1,50
Link Pointer 5
Loading Programs 145,152
Logical Operations 46
Machine Code 24,50,188
Memory Execute 189
Memory Read 189
Memory Write 188
Menu 105
Mice 106
Microprocessor 24
Mixed Mode Display 102
MPS801 printer 193
Multicolour Graphics 56
Music 122
New disks 142
Notes 123, 217
NMI 45
Plotting Points 63
Pictures program 130
Pixel 62
Processor status register 26
Program Counter 26
Quitting DOS 5.1 153
Random Files 163
Raster Register 92
Raster Scan 91
Relative addressing 41
Relative Files 157
Rename 149
Renumber Program 7
Reserved Words 4
Reverse Mode Printing 202
Running Machine Code 48
Saving Programs 145,152
Scratch 149
Screen Dump Program 203
Screen Memory 53, 55
Sector 139
Sequential Files 154
Shifts 47
SID Chip 121
Side Sectors 158
Simple variables 15
Sketchpad 107
Stack 43
Stack Pointer 27, 43
Status Register 25
Storing Machine Code 49
String Variables 17
Subroutines 42
Tab Print Head 200
Token 4
Track 139
Tunes 121
TV operation
User 189
User Friendly 105
Validate 148
Variable display prog 18
Variable dump program 20
Variable Storage 15
VIC-II Chip 53, 91
X register 25
Y register 25
Zero Page Addressing 30

Available in Century's Science & Technology series

Dictionary of New Information Technology
A. J. MEADOWS, M. GORDON and A. SINGLETON
What to Buy for Business: A Handbook of New Office Technology
JOHN DERRICK and PHILLIP OPPENHEIM
The Electronic Mail Handbook
STEPHEN CONNELL and IAN A. GILBRAITH
The Way the New Technology Works
KEN MARSH
Century Computer Programming Course
PETER MORSE and IAN ADAMSON
Computers and Your Child
RAY HAMMOND
International Dictionary of Graphic Symbols
JOEL ARNSTEIN
The Really Easy Guide to Home Computing: the ZX Spectrum
SUE BEASLEY and RUTH CLARK

Century Microguide to the ZX Spectrum
Century Microguide to the BBC Micro
Century Microguide to the Dragon
Century Microguide to the Commodore 64
edited by PETER MORSE

The ORIC ATMOS Handbook
PETER LUPTON and FRAZER ROBINSON
The Commodore 64 Handbook
PETER LUPTON and FRAZER ROBINSON
The Atari XL Handbook
PETER LUPTON and FRAZER ROBINSON

Assembly Programming Made Easy for the BBC Micro
IAN MURRAY

In association with 'Personal Computer World'

Microcomputing for Business: A User's Guide
edited by DICK OLNEY
The Microcomputer Handbook: A Buyer's Guide
edited by DICK OLNEY
The Spectrum Handbook
TIM LANGDELL
The Intimate Machine
NEIL FRUDE
35 Educational Programs for the BBC Micro
IAN MURRAY
Educational programs for the Dragon 32
IAN MURRAY
Educational Programs for the Spectrum
IAN MURRAY

Information Technology Yearbook
edited by PHILIP HILLS
The Database Primer
ROSE DEAKIN
Computer Gamesmanship
DAVID LEVY

Best of PCW: Software for the BBC Micro
Best of PCW: Software for the Dragon
Best of PCW: Software for the Spectrum
Best of PCW: Software for the Electron
Best of PCW: Software for the Commodore 64

The ORIC Handbook
PETER LUPTON and FRAZER ROBINSON
35 Programs for the Dragon 32
TIM LANGDELL

CENTURY SOFTWARE VOUCHER

The major programs in this book are available on cassette from the publishers.

Please cut out or copy this voucher and send it together with a cheque / postal order for £6 to:

George Philip Services Ltd.,
Arndale Road,
Wick,
Littlehampton
West Sussex
BN17 7EN

VOUCHER

Please send me one cassette of the major programs in the ADVANCED COMMODORE 64 HANDBOOK at the special price of £6 including postage and packing.

NAME .....

ADDRESS .....

.....

.....

Please allow 28 days for delivery.

The Commodore 64 is a very powerful micro and a beginners' guide can only hint at some of the possibilities achievable.

The *Advanced Commodore 64 Handbook* builds on the material in the authors' companion volume, *The Commodore 64 Handbook*, but explains how to exploit the advanced features of the Commodore 64 even further.

The Advanced Handbook concentrates particularly on graphics, sprites and sound, and develops the potential of these with program examples and utilities. Machine Code programming is introduced to show what a significant effect it can have on running speed, especially for graphics applications. Also included is a comprehensive treatment of available peripherals including use of discs and printers.

The *Advanced Commodore 64 Handbook* takes the reader far beyond just using the Commodore 64 and on to being a really creative Commodore 64 user.

A companion volume to *The Commodore 64 Handbook*.

ISBN 0 7126 0420 0 ●
CENTURY COMMUNICATIONS LTD

0784
NR03971
156,25



This was brought to you

from the archives of

http://retro-commodore.eu