# USEFUL SUBROUTINES AND UTILITIES FOR THE COMMODORE 64

IAN SINCLAIR

# Useful Subroutines and Utilities for the Commodore 64

**Other books for Commodore 64 users**

*Business Systems on the Commodore 64*
Susan Curran and Margaret Norman
0 246 12422 9

*Adventure Games for the Commodore 64*
A. J. Bradbury
0 246 12412 1

*Commodore 64 Computing*
Ian Sinclair
0 00 383070 5

*Commodore 64 Disk Systems and Printers*
Ian Sinclair
0 246 12409 1

*The Commodore 64 Games Book*
Owen Bishop
0 246 12258 7

*Software 64: Practical Programs for the Commodore 64*
Owen Bishop
0 246 12266 8

*Introducing Commodore 64 Machine Code*
Ian Sinclair
0 246 12338 9

*40 Educational Games for the Commodore 64*
Vince Apps
0 246 12318 4

*Advanced Machine Code Programming for the Commodore 64*
A. P. Stephenson and D. J. Stephenson
0 246 12442 3

*Commodore 64 Graphics and Sound*
Steve Money
0 246 12342 7

*Commodore 64 Wargaming*
Owen Bishop and Audrey Bishop
0 00 383010 1

*Filing Systems and Databases for the Commodore 64*
A. P. Stephenson and D. J. Stephenson
0 00 383011 X

*Data Handling on the Commodore 64 Made Easy*
Jim Gatenby
0 246 12454 7

# Useful Subroutines and Utilities for the Commodore 64

Ian Sinclair

COLLINS
8 Grafton Street, London W1

Collins Professional and Technical Books
William Collins Sons & Co. Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Collins Professional and Technical Books 1985

Distributed in the United States of America
by Sheridan House, Inc.



*British Library Cataloguing in Publication Data*
Sinclair, Ian R.
Useful subroutines and utilities for the
Commodore 64.
1. Commodore 64 (Computer)—Programming
I. Title
001.64'2 QA76.8.C64

ISBN 0-00-383012-8

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed and bound in Great Britain by
Mackays of Chatham, Kent

# Contents

# Introduction

The Commodore 64, though it is capable of excellent displays of computing, is handicapped with a very elementary version of the BASIC programming language. This means that programming the C 64 for yourself in BASIC is an uphill struggle which many owners never even attempt. These owners who have programmed for themselves must often look at machines like the BBC Micro, and wish that they too could have commands which would renumber their lines, delete groups of lines, program the function keys and so on.

There are several solutions to this problem. One is to buy an extended BASIC for the C 64. There are at least two available, of which one runs well, with few bugs. These extended BASIC programs, however, are costly both in terms of cash and in memory. Once you have your extended BASIC installed, for example, you will find that you have as little as 12K of memory left for your BASIC program. Another alternative is to buy a type of program which is described as a 'toolkit'. This is also a good way to extend the abilities of your C 64, particularly if you need and can use almost all of the facilities that are provided. Once again, however, this is costly in cash and memory, particularly if there is just one facility that you really need.

This book offers you yet another approach. Contained here are listings of utilities and subroutines which can make your C 64 more useful to you. These are simple listings of fairly elementary routines. I don't claim that the contents of this book can ever replace a good 'toolkit' or an extended BASIC. What it can do, however, is to provide you with several enhancements to the BASIC of the C 64 which you will find in most toolkits, and a number of subroutines that you will find useful. These are all simplified versions of routines which could otherwise be placed on cassette and sold separately. The reason for using simplified versions is twofold. The first reason is that

I feel sure that few readers are happy with long listings. Not all readers are expert typists, and the shorter a listing is, the more likely it will be that you will type it correctly. The second reason is that when you have typed a listing for yourself, and have read notes about it, you don't need so much prompting from the program itself. The most important feature of the listings, however, is explanation. When you buy a toolkit or an enhanced BASIC, you don't get listings and you are unlikely to be told how the routines work. This book features full explanations with each listing. Once again, there are two reasons for this. One is that it makes it possible to reduce the length of the listings. The more important one is that you will be able to learn a lot about the workings of your C 64 from these listings. Most readers of this book will have had some experience of programming the C 64 and, with the information that is presented here, they will be able to develop their own more advanced utilities from the 'skeleton' versions that are shown here.

Many of the utilities make use of machine code routines. If you program only in BASIC, you may think that the explanation of how these machine code sections work is of little interest. This is not so. You will learn from these explanations what the effect of various POKE commands will be, and your mastery of the machine will be greatly enhanced. If you find that machine code has attractions and you want to know how to get started, then I suggest that you take a look at my book *Introducing Commodore 64 Machine Code.*

If you are already a machine code programmer, then the explanation of the utility routines will be very useful to you in enabling you to develop your own routines in future. The original routines were all written with the aid of the excellent MIKRO assembler cartridge, written by Andrew Trott and sold by Supersoft. If you do not already use an assembler, I thoroughly recommend this editor/assembler/monitor package, which is in cartridge form and which also offers a few useful enhancements to BASIC.

All of the listings in this book have been photocopied directly from the printouts. For the sake of legibility, the printer that I used was the Epson MX-80, connected through the Microport printer-link. Since this printer does not reproduce Commodore graphics symbols directly, I have avoided using any keys that print symbols. This makes the listings very much easier to read. Because the printouts have been photocopied, there will be no typing errors in the listings – each listing is as it was when the program was running. What you see is what I used.

Finally, when a book contains listings of short routines, there is

always a problem of what line numbers to use. I have used high line numbers, starting at 60000 for most of the short subroutines and utilities. The longer programs, and programs which delete themselves are numbered from line 10 in the conventional way. I have made no provision for joining routines together, because no two readers of this book will want the same mix of routines. I have, however, indicated in the text which of the machine code routines can be shifted to different addresses so that more than one routine can be put into memory at the same time. Happy programming!

Ian Sinclair

# Part One
# Utilities

Anyone who has programmed more than one type of computer will know that the differences between machines extend to much more than the number of keys on the keyboard or the size of the memory. A keen programmer will always prefer to use a machine which allows a lot of editing facilities – such as the ability to delete several lines with one command – and other aids to programming – such as line re-numbering and auto line numbering. These commands are often called *utilities*; they are not a part of the BASIC programming language, but they are very useful to anyone who programs in BASIC or any other language apart from machine code. A machine which has all of these utilities built-in may, however, use so much memory for ROM that rather a small amount, less than 32K, will be available for the programmer to use. Some recent designs have overcome this by using separate memory for the screen display or by switching between RAM and ROM as the program runs.

The Commodore 64 uses a very small and simple version of BASIC, with a very good screen editing system. To allow enough memory for high resolution graphics, practically all of the utility commands which nowadays are considered a normal part of a computer's BASIC have been omitted. The C 64 programmer need not feel at a disadvantage, however, because utilities can be added. This section describes a number of routines which compensate for the most noticeable omissions from the ROM of the C 64. Most of these involve the use of machine code because utilities generally have to be used while the machine is being programmed rather than being part of a program themselves. So that the BASIC language programmer is not placed at a disadvantage, the utilities are all in the form of BASIC programs which can be run and then deleted. Many will automatically delete themselves after they run, leaving the machine code in memory. In addition, the assembly language of the machine code has been given so that the machine code programmer can follow the action of the utilities. Following the scheme which was explained in the Introduction, the utilities have been kept short, even if this has meant restricting the range of uses.

# 1
# Appending BASIC Programs

If you read any modern text on how to design BASIC programs, you will come across the same good advice – that you should design your programs to make the greatest possible use of subroutines. A program which is designed in this desirable 'top-down' way will consist of a core, usually quite short, which consists mainly of GOSUB commands. The subroutines carry out all the work of the program, and the core directs and organises this work. A program which has been written in this way is much easier to correct, amend, and understand than one which has been written haphazardly, with GOTOs sprinkled all over the listing.

In addition, you will find that when you program in this way, you can make use of the same subroutines in many different programs. Some such subroutines are listed in this book. Most computer owners can keep cassettes or disks of these subroutines, and they can add them to their new core routines simply by using a MERGE command. Unfortunately, the C 64 has no such MERGE command. You can't add a subroutine to a core program by using LOAD, because LOAD has the effect of clearing out the program which is in the memory, leaving you only with the subroutine.

Fortunately, this MERGE (more correctly, APPEND) action can be obtained, like so many desirable things on the C 64, with a few POKEs. The C 64 uses some low-numbered memory addresses to keep a note of how its memory is being used. Addresses 43 and 44 are used to keep a note of the address at which a BASIC program starts. When you switch on your C 64, the number in address 43 is 1 and the number in address 44 is 8. When an address is stored in two consecutive places like this, its value is found by multiplying the *second* number by 256 and adding the result to the first number. In this case, 8*256+1 gives 2049, and this is the address at which you will find the first byte of a BASIC program when you type it or load it from cassette.

Another pair of addresses, 45 and 46, is used to hold the 'start of variables'. Each variable name that you use in a program, together with its value (if it is a number) or an address (if it is a string) is held in a list which follows the last bytes of the BASIC program. These last bytes are both zero, and they act as a signal to the machine that there are no more lines to read. The last actual byte of your program, then, is at an address two places less, at the number given by:

PEEK(45)+256*PEEK(46)−2

The use of these addresses allows us to append one program to another (see Fig. 1.1). Suppose that you have a program sitting in the memory of the computer. It ends at the address which is two places lower than the address held in 45 and 46. If you now poke this ending address into 43 and 44, the machine will from then on treat this as the start of BASIC. Anything which you type, or load from cassette, will be stored at addresses which follow on from the last part of your original program. When you restore the correct starting address into 43 and 44, the machine then links in the new piece of program just as if you had typed it or loaded it in one operation.

---

Type: POKE 43,PEEK(45)−2:POKE 44,PEEK(46) press RETURN
LOAD next program
Type: POKE 43,1:POKE 44,8 - press RETURN.

---

*Fig. 1.1.* A reminder of the appending procedure.

Nothing is ever perfect, though. You must make sure that the subroutine or program which you are adding has line numbers which are *all higher than the highest line number that was used in the original program.* If it doesn't, the append action will still work, but the new section will not be correctly linked in and will not RUN correctly. If there is likely to be a clash of line numbers, then you will have to renumber one of the programs. If only one line conflicts, you may be able to renumber that line by using the editing commands, before you attempt to run the merged programs. In this book, many of the routines have been numbered starting with line 60000 to ensure that you can merge them with other routines with no need for renumbering unless you want to merge more than one subroutine.

# 2
# Linefinder

When you have a BASIC program in the memory of your C 64, the program is stored as a set of bytes in the memory, starting normally at address 2049, and with a zero stored at address 2048. Each of the command words, like PRINT or MID$, is stored as one code only, rather than as ASCII codes for the commands. For example, PRINT is stored as 153 instead of the ASCII codes 80,82,73,78,84 – a saving of four bytes of memory. Anything which is not a command, however, is stored as ASCII code.

In addition, however, five bytes in each line are used to 'index' the line. This extra five bytes is often known as the 'overhead', because whenever you take another line, you have automatically made use of five more bytes of memory. If you use long lines, separating each section with a colon, it is possible to cram a much longer program into the memory – but that's another matter, and the 80-character limit to C 64 line length makes it less useful to the C 64 owner in any case.

The structure of a line is illustrated in Fig. 2.1. Each line starts with two bytes which give the address of the *next* line. This is how the machine is able to locate a line in the memory, by stepping from one line to the next. These two 'next-line address' bytes are in the form which is usual for integer numbers, low byte first, then high byte. To find the next-line address, then, you have to multiply the high byte by 256, and add the low byte. The next two bytes are the line number of the present line. Once again, these are in low-high byte order, so for line numbers less than 255, the high byte will be zero. Following the high byte of the line number, you will have the instruction bytes and ASCII codes of the line. At the end of the line, a zero byte is used as a marker. This zero, along with the next-line bytes and the line number bytes, constitutes the five bytes of overheads. At the end of the complete program, this zero byte is followed by two more zeros. When the computer attempts to read the next line address, the fact

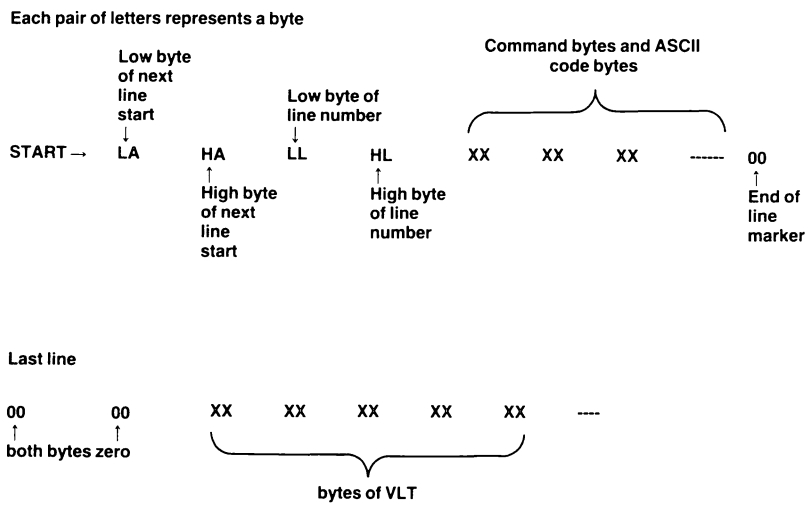


*Fig. 2.1.* How a line of BASIC is stored in the memory of your C 64.

that this is zero forces the machine to return to the READY state, waiting for another command. In other words, the program is finished.

## How to use it

Now a lot of interesting things can be done when you know how a program is stored in the memory. Just to take an example, suppose you want to conceal a copyright notice. You could have as your first line something like this:

```
10 A=22.5:B=3.14:REM IAN SINCLAIR
```

of which the first two instructions are vital to the running of the program. Anyone who copies the program is therefore unlikely to remove this line. If you then poke a zero byte in place of the second colon, however, the machine will not show this colon, nor anything following it. On the screen, line 10 will appear as:

```
10 A=22.5:B=3.14
```

with *no* copyright notice. The bytes are still in the memory, but no-one who uses the program will know this unless they examine the memory closely. If you come across a copy which you think is of your program, you only have to poke back the colon in place of the zero, list, and you will see your copyright notice once again.

Another possibility is that you can change line numbers. Now the sequence of the lines in a program, once the RETURN key has been pressed at the end of entering a line, is determined by one thing alone. That is the address of the next line which occurs as the first two bytes of the program. What two bytes are used as line numbers is quite irrelevant once the lines have been entered. You can change all of the line numbers to 10, if you like, or to zero, or number them all in reverse order. This will *not* affect the way in which a program runs. It will, however, play havoc with any attempts to list a particular line, or edit a particular line, and it's another way of making a program more secure against copying. You can also poke a set of three zeros into a program so that the machine is fooled into believing that there are no lines at all, or that there is only one line. The program then cannot be run until the correct pokes have been performed.

All of these tricks depend on being able to find where lines have been stored in the memory of the computer. The short routine in Fig. 2.2 is designed to do just that. When the program runs, you are asked for a line number. As usual, this must be less than 64000, which is the limit of the line numbers that the C 64 can normally accept. Oddly enough, though, it is possible to renumber lines up to 65535. The answer you will get from the program will either be the address of the first byte of the line (the low byte of the 'next-line'

```
63000 PRINTCHR$(147):PRINTTAB(15)"LINEFI
NDER"
63010 PRINT:PRINT"PLEASE TYPE LINE NO.-
";
63020 INPUT S
63030 IF S<1 OR S>63999 THEN 63110
63040 PRINT:PRINT"PLEASE WAIT--------"
63050 A1=PEEK(43)+256*PEEK(44)
63060 T=PEEK(A1+2)+256*PEEK(A1+3)
63070 IF T=0 THEN 63110
63080 IF T<>STHEN A1=PEEK(A1)+256*PEEK(A
1+1):GOTO63060
63090 PRINTCHR$(147):PRINT:PRINT"LINE ";
S;" STARTS AT ADDRESS ";A1
63100 END
63110 PRINTCHR$(147):PRINT:PRINT"LINE ";
S;" DOES NOT EXIST."
63120 END
```

*Fig. 2.2.* The linefinder program.

address), or the reply 'DOES NOT EXIST', if the line cannot be found.

## How it works

The line number that you have requested is stored as variable S, in line 63020. Line 63050 then finds the address of the start of BASIC. This is taken from addresses 43 and 44 rather than assuming a start at 2049, just in case the BASIC has been shifted to another part of memory as described in the routine in Fig. 1.1. Line 63060 then finds the line number. This is done by using the third and fourth bytes in the line, remembering the low-high order. If this number is zero, then this normally indicates that there are no more program bytes to be found, and the line has not been found. This test will cause problems if you have a program with a line number zero, however, so be careful! Line 63080 compares the line number which you requested, S, with the number that has been found, T. If the two are not identical, the address A1 is changed to the next-line address, taken from the first two bytes of the current line. The search is then restarted.

The line numbers for the linefinder have been set to 63000 onwards, because you will normally want to keep this program on tape, and MERGE it with the BASIC program whose lines you need to investigate. Since this is a purely BASIC program, you can, of course, renumber it as you wish, but it is more useful to have it at the end of the main program. You will need to run this program by using GOTO 63000, because RUN, used by itself, will cause the main program to run. If the linefinder is renumbered with low line numbers, then RUN will cause it to operate before the main program, and you will need to use a STOP line to prevent the main program from running. If you want to alter bytes in the main program, however, you must keep the main program in the same part of memory, so you can't edit or renumber after poking the memory.

# 3
# Machine Code to DATA

This utility is one which I use a great deal for myself, and in fact it was extensively used in the preparation of the listings for this book. The C 64 is not a machine which is well-equipped for working with machine code. Most other machines come with instructions in BASIC for saving or loading machine code programs, but the C 64 lacks even this elementary provision. If you use the MIKRO assembler cartridge, or any of the other assemblers or machine code monitors that are available, you will have a set of commands which add these useful provisions to your machine. If you don't have MIKRO, however, then the use of machine code is only available by roundabout methods. Even if you do use an assembler/monitor cartridge, there are many occasions on which you want to include a piece of machine code along with the BASIC of a main program.

The reasons are not hard to find. BASIC is adequate for a lot of computing work, and especially for arithmetic or mathematics. For fast-moving displays, however, BASIC is much less useful, and the small number of BASIC commands that the standard C 64 supports does not help. If you write your own machine code routines with the help of an assembler, then you will end up with machine code stored in the memory of the computer, but no easy method of attaching it to a BASIC program. If you use other programs which contain machine code sections, and you would like to use the machine code in your own programs, then, once again, this is not particularly easy.

This program, however, alters the whole situation completely. The first version of this program that I wrote was for a TRS-80 in 1979, and it was simply so that I could attach machine code to BASIC programs and use it. The TRS-80 allowed machine code to be loaded in directly from tape, but in many cases this meant loading the machine code, and then loading the BASIC. What is needed is to transform the bytes of the machine code into numbers in DATA lines. You can then write a short 'loader' program which pokes these

bytes into memory, so that they can be run wherever they are needed.

At one time, I had listed the bytes of the machine code laboriously on paper, and then written DATA lines. For a few bytes, this is tolerable, but when you get machine code programs of 20 bytes or more, it gets tedious. More seriously, it gets error-prone, and it's a waste of time, because the machine can do all of this for you!

## How to use it

This program (Fig. 3.1) reads the bytes of a machine code program, and rewrites them in the form of DATA lines. This means that the program alters itself as it runs, and you should therefore keep several copies of the program in its original form. When you run the program

```
100 PRINTCHR$(147):PRINTTAB(15)"DATAFIND
ER"
110 PRINT:PRINT"PLEASE TYPE STARTING ADD
RESS (DECIMAL)
120 INPUT S
130 PRINT:PRINT"PLEASE TYPE FINISHING AD
DRESS (DECIMAL)"
140 INPUT F
150 PRINTCHR$(147):PRINTTAB(214)"PLEASE
WAIT"
160 A1%=PEEK(43)+256*PEEK(44):H%=F-S:IF
SGN(H%)=-1THEN GOSUB500:GOTO100
165 T%=PEEK(A1%+2)+256*PEEK(A1%+3)
170 IF T%<>1000THEN A1%=PEEK(A1%)+256*PE
EK(A1%+1):GOTO165
180 Q%=A1%+4
190 Z%=0:FOR N=0TOH%-1
200 L$=STR$(PEEK(S+N)):LL=LEN(L$)
210 FOR X=2TOLL:Y%=ASC(MID$(L$,X,1)):Z%=
Z%+1
220 POKE Q%+Z%,Y%:NEXTX:Z%=Z%+1:POKE Q%+
Z%,44
230 IF Z%<62THEN NEXT N
232 IF N>=H%THEN 250
234 POKEQ%+Z%,32:FORD=1TO3:IF PEEK(Q%+Z%
+D)=38THEN POKE Q%+Z%+D,32
236 NEXTD
240 Q%=Q%+71:Z%=0:NEXTN
250 PRINT:PRINT"TRANSFER OF ";H%;" BYTES
 COMPLETE"
```

```
310 PRINT:PRINT"LINES 1000 ON NOW CONTAI
N DATA FOR YOUR MACHINE CODE"
320 END
500 PRINT"END ADDRESS IS GREATER THAN ST
ART ADDRESS, PLEASE TRY AGAIN"
510 FORX=1TO1000:NEXT:RETURN
1000 DATA&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1010 DATA&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1020 DATA&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1030 DATA&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1040 DATA&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1050 DATA&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1060 DATA&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1070 DATA&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1080 DATA&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1090 DATA&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
```

*Fig. 3.1.* The machine code to DATA program in BASIC.

you will be asked to supply the (decimal) starting address and finishing address of the machine code. The finishing address must be greater than the starting address, otherwise you will get an error message! This is easy enough if you have written the code for yourself with an assembler, because you will then have decided on these addresses. The assembler will supply the addresses in hex, but the conversion is simple enough (see Fig. 3.2). It's not so easy if you want to make use of a machine code routine which is part of another program. The start can often be found from a SYS command, but the end may have to be guessed. A disassembler can often help you in this, but you need a reasonable knowledge of machine code to use it correctly. If you have no other clues, you may have to guess a length, trying numbers like 25, 50, 100 and so on until you find that the machine code works in your version. Once you have found a length that works, you can try lower finishing addresses until you find one that doesn't work. One clue is that the last byte of the program is

**(a) Denary to hex**

| Denary | Hex |
|---|---|
| Numbers 0 to 9 are identical | |
| 10 | 0A |
| 11 | 0B |
| 12 | 0C |
| 13 | 0D |
| 14 | 0E |
| 15 | 0F |

**(b) Converting hex addresses into denary**

(1) A hex address contains up to four digits. Separate these digits, and convert each one into its denary equivalent, using the table of (a) above.
(2) Deal with the values according to their place in the number. Don't change the right-hand side value. Multiply the next to the left by 16, the next by 256 and the next by 4096. Add all of the results. This is the denary address.

*Examples:*

(1) Hex number $4A2C
Split into 4 10 2 12
Values to add are 12 + 2*16 + 10*256 + 4*4096, giving 18988.
(2) Hex number $80E
Split into 8 0 15
Values to add are 15 + 0*16 + 8*256 = 2062.

**(c) Separating a denary address into (denary) bytes.**

Divide the address number by 256. The whole number part is the upper byte, the remainder is the lower byte.

*Example:*

Address 41235 is divided by 256 to give 161 and a remainder of 19.
The high byte is 161, and the low byte is 19.
This is not so easy on a calculator, because a calculator does not give remainders. To find the bytes on a calculator, divide by 256 as before and write down the whole number part. Now multiply this whole number part by 256. Subtract this result from the original number to find the remainder.

*Example:*

Address 23225 is divided by 256 to give the whole number part of 90.
90*256 is 23040, and subtracting this from 23225 gives 185.
The high byte of this address is 90 and the low byte is 185.

*Fig. 3.2.* Hex codes and addresses. (a) How the ordinary (denary) numbers from 0 to 15 are converted to and from hex. (b) How to convert a hex address into denary. (c) How to split a denary address into two bytes.

often 96 (the return-from-subroutine command in machine code), or three bytes which start with 76 (the jump instruction). You can then look for these numbers in your DATA lines.

Once you have entered the finishing address, the program prints a 'PLEASE WAIT' notice, and gets to work. Lines 1000 to 1090 of the program contain the & sign, 65 of them in each line. This is done so as to reserve memory space for the DATA bytes. The ten lines that are shown here are enough for most short machine code segments, but if you want to make use of jumbo-size machine code programs, you might want to add other DATA lines. If you do so, remember that each line *must* contain the exact number of & signs if the program is to work correctly. When the program ends, you will see a notice on the screen. This tells you how many bytes of machine code have been read, and reminds you that the codes are now in the form of DATA numbers in lines 1000 onwards. You can then delete the lines of the program from 10 to 510 (if you use the MIKRO assembler, you can use the command DELETE 10-510 for this) and you can also delete any DATA lines which contain only & signs. This leaves you with lines which contain numbers, and normally with one line that ends with a comma and a few & signs. You can use the excellent editing facilities of the C 64 (its best feature) to remove the surplus & signs and the last comma. You now have a set of DATA lines for a machine code program.

To make use of these lines, you will need to write your own 'loader'. This simply means that you will need to read the numbers and poke them into memory. Such a loader might take the form that is shown in Fig. 3.3. Note that you can choose the memory address for the start

```
100 FOR N=0 TO 56:READ D%
110 POKE AD+N,D%:NEXT
```

*Fig. 3.3.* A typical 'loader' program for poking bytes into memory. You will see this type of BASIC loader used in many of the programs that follow.

of the program, but unless you know a lot about the program, it's best to use the same addresses as you read it from. There's nothing to stop you reading bytes from addresses 49152 to 49182 and writing a loader that pokes these bytes into other addresses. You may find, however, that (a) these addresses are being used for something else, or (b) the machine code program is non-relocatable. A non-relocatable machine code program *must* be placed in the same range of addresses each time; it cannot be shifted without modifying the program bytes.

Some of the programs in this book are completely relocatable, so that you can place them anywhere in memory that you like. You need only make sure that the addresses are not used by BASIC to store anything else. This can be done by poking addresses 52 and 56 before the program runs. These addresses contain the high byte of the number that decides the highest address that BASIC can use.

## How it works

The start address is held as variable S and the finishing address as F. Line 160 (see Fig. 3.1) then finds the starting address for the BASIC program by peeking addresses 43 and 44. This is the same action as was used in the linefinder program. In this same line, the length of the machine code program, H% is calculated, and if this is negative, the error message in line 500 is printed. Lines 160 to 170 carry out the linefinder action, looking for line 1000, which is the first DATA line. When this is found, the number A1% is the address of the start of line 1000.

In line 180, Q%=A1%+4 sets the address at which the first instruction code of the line is located. This will be the code for DATA, which is 131. We now need to read the bytes of machine code, convert them to ASCII code numbers, and poke into place. For example, if we read a byte 146, then this has to be converted into ASCII codes 49,52,54, because DATA lines must contain only ASCII codes. Line 190 prepares for this by setting a variable Z% to zero, and starting a loop using N. N is the variable which will count out the bytes of the machine code program from 0 to H%−1. You can't use an integer variable here, because the C 64, unlike so many other machines, will not accept an integer variable as a loop counter! The variable Z% is used to select the address in the DATA line to which each ASCII code will be poked.

Line 200 then peeks the machine code byte from address S+N, and converts this number into a string. The conversion into a string means that the number now exists as a set of ASCII codes, and the second part of the line makes LL carry the length of this string. Now we have to be careful here. When STR$ is used, the string that is formed *always* starts with a blank, to allow space for a + or − sign. The length figure is therefore one more than the number of digits in the string. For example, if you have the number 143, the length of this will be given as four characters, not three. When we read this, we

don't want to use the blank, so we will start reading the string from its *second* character, not its first.

Line 210 does this, using X=2 TO LL. Each ASCII code is assigned to Y%, using ASC(MID$(L$,X,1)) to find the character and its ASCII code. The variable Z% is incremented, and then line 220 pokes Q%+Z% (the DATA line in the memory) with the ASCII code, Y%. By incrementing Z% before poking, you ensure that address Q% is not poked - this contained the DATA code, you remember. The NEXTX ensures that all of the characters of the byte are poked, incrementing Z% each time, then Z% is incremented again, and 44 poked into memory. The 44 is the ASCII code for the comma, so by this stage we have completed poking the ASCII codes for the first byte of machine code.

We then have to test for the end of a line. We must not poke more than 65 characters into a DATA line under any circumstances. If we do so, the 'next-line' address will be corrupted, and the program will run wild. Line 230 allows poking to continue only for as long as Z% is less than 62, giving a safety margin. Line 232 then tests for reading past the end of the machine code bytes. If 62 or more characters have been poked into a line, then 234 is executed. This pokes a space, and tests the rest of the line. If a '&' sign (ASCII 38) is present, this is replaced by a space, until there are no more '&' signs. Line 240 then shifts the value of Q% to the DATA code for the next DATA line, and the loop which starts in line 190 then continues. By the end of the program in line 250, all of the bytes of machine code have been transformed into DATA numbers for use in your own BASIC programs.

The machine code programs that are shown in this book are each illustrated in two forms. One is the form of assembler code, so that you can follow the program if you know enough about machine code. The other form is as a BASIC loader, using DATA bytes that have been obtained by using this program. This has eliminated the problem that plagues many authors of transforming bytes of machine code into DATA lines by hand and suffering the inevitable mistakes. The general principles that are used in this program can be applied to other purposes of your own, and you will find them used, for example, in the Sprite Editor program later in this book.

# 4
# Auto Line Numbering

A lot of published programs have been renumbered so that the first line is numbered 10 and subsequent lines are numbered in tens. When you want to type such a program into your computer, it is rather time-wasting to have to type the line number of each line before you type the instructions, and you may even sometimes type the wrong number. Many machines have an AUTO command, which places the number 10 on the screen, allows you to type the line of that number, and will replace it with a 20 when RETURN has been pressed. At each use of RETURN after that, a new line number will appear. The C 64 does not have an AUTO command, but the action can be obtained by making use of machine code.

## How to use it

This is the first of the full-blown machine code programs in this book, and two versions are shown. Figure 4.1 shows the assembly language listing, as printed by the MIKRO assembler. If you are a machine code programmer, you will be able to follow the 'How it works' section with reference to this listing. If you program only in BASIC, Fig. 4.2 shows a BASIC loader with DATA lines which will place the machine code into memory and run it for you. Since this BASIC program exists only to place the machine code bytes into memory, it deletes itself after running, and all that you will notice is a brief interval between the message in lines 10 to 25, and the READY prompt which appears when the program is ready to use. Each press of the RETURN key will now generate a new line number, starting with 10 and going up in tens. You can type the instructions of a BASIC program in these lines, and when you have finished, press the RUN/STOP and RESTORE keys to stop the machine code action. You should then SAVE your BASIC program in the usual way.

```
100 C000          *=$C000
110 C000          KBUF          =   $0277
120 C000          KNUM          =   $C6
130 C000 78                     SEI
140 C001 A915                   LDA #<AUTO
150 C003 8D1403                 STA $0314
160 C006 A9C0                   LDA #>AUTO
170 C008 8D1503                 STA $0315
180 C00B 58                     CLI
190 C00C A90A                   LDA #10
200 C00E 85FB                   STA 251
210 C010 A900                   LDA #0
220 C012 85FC                   STA 252
230 C014 60                     RTS
250 C015 A5C5     AUTO          LDA $C5
260 C017 C901                   CMP #1
270 C019 D039                   BNE EXIT
280 C01B A6FB                   LDX 251
290 C01D A5FC                   LDA 252
300 C01F 8562                   STA $62
310 C021 8663                   STX $63
320 C023 A290                   LDX #$90
330 C025 38                     SEC
340 C026 2049BC                 JSR $BC49
350 C029 20DDBD                 JSR $BDDD
360 C02C A0FF                   LDY #$FF
370 C02E C8       LOOP          INY
380 C02F B90101                 LDA $0101,Y
390 C032 997702                 STA $0277,Y
400 C035 D0F7                   BNE LOOP
410 C037 C8                     INY
420 C038 A900                   LDA #$00
430 C03A 997702                 STA $0277,Y
440 C03D 84C6                   STY $C6
470 C03F 18                     CLC
480 C040 A5FB                   LDA 251
490 C042 690A                   ADC #$0A
492 C044 85FB                   STA 251
500 C046 9002                   BCC DELY
510 C048 E6FC                   INC 252
512 C04A A2FF     DELY          LDX #$FF
514 C04C A0FF     LOOP2         LDY #$FF
516 C04E 88       LOOP1         DEY
518 C04F D0FD                   BNE LOOP1
520 C051 CA                     DEX
```

```
522 C052 D0F8                BNE LOOP2
530 C054 4C31EA EXIT         JMP $EA31
```

*Fig. 4.1.* The AUTO program, in assembly language.

Because this is a program which pokes codes directly into memory, you should type the lines carefully, particularly the DATA lines. A mistake in a DATA line can completely ruin any machine code program, so *always* save such a program before you attempt to run it. In this case, it is particularly important to SAVE the program, because it deletes itself after it has run. When you have finished typing, count the number of DATA items, which should be 87. Check that there are no errors in the numbers, such as might be caused by a sticking key, for example. My own C 64 has a sticky '2' key, so that numbers like '21' often come out as just '1'. Record the program, and then RUN it. If the machine locks up, then it indicates a fault in the machine code, probably a faulty DATA number.

## How it works

The two important clues to how this program works are the IRQ interrupt and the key-buffer. When the C 64 is operating with no program running, an 'interrupt' occurs 50 times per second (60 times per second in the US version). An 'interrupt' is what the name suggests – the action of the machine is interrupted, and the keyboard is scanned to find if a key has been pressed. If a key has been pressed, its ASCII code is put into a piece of memory called the keyboard buffer. If no key has been pressed, there is no action. In either case, the machine automatically returns to whatever it has been doing previously, such as displaying something on the screen. These interrupts are selectively disabled while the machine is running a BASIC program, and the only keys that are recognised are the RUN/STOP and RESTORE keys, which allow you to break into a BASIC program to stop it. On each interrupt also, if there is no program running, anything that has been stored in the keyboard buffer is put into the main memory of the machine to be processed after the interrupt.

The keyboard buffer is very important, because this is where the bytes of each command to the computer are stored temporarily when you type them. Instructions are held in ASCII form here, and are converted into single byte codes only when the RETURN byte is encountered. If you poke bytes into this buffer, the effect will be the

same as if you had typed them. The keyboard buffer starts at address $277 (decimal 631) and extends for ten bytes.

The AUTO action is carried out by intercepting the interrupt, and testing to find if the RETURN key has been pressed. If it has not, the interrupt is allowed to proceed normally. If the RETURN key has been pressed, however, a line number is taken from two addresses in the memory, converted into ASCII codes, and these codes are poked into the keyboard buffer. At the end of the interrupt, then, the keyboard buffer contains the same bytes as it would have if you had typed the number. The machine will therefore place the bytes on the screen and await the rest of the line which starts with that number. Meanwhile, the normal interrupt action has been restored until the next use of RETURN.

Figure 4.1 shows the assembly language coding of this routine. As usual, all numbers that are prefixed with the dollar sign are in hex. The word KBUF is used to mean the address of the start of the keyboard buffer, and KNUM is address $C6, the address which contains the keyboard buffer count, the record that the machine keeps of how many bytes are waiting to be read from the buffer. Addresses 251 and 252 (decimal) have been used to contain line numbers. If you want your auto numbering to start with 100 rather than with 10, you can alter the starting number in 251 for yourself.

Lines 100 to 230 of the assembly language program deal with setting up. The first main action is to alter the address to which the machine goes when an interrupt occurs. This is normally $EA31, but it can be altered by poking a new number into addresses $314 and $315. While this is being done, however, an interrupt must not be allowed to occur. If an interrupt occurred while the address was being altered, the machine could be forced to jump to an address which would cause a crash. The only way out of such a crash is to switch off – losing the program. That's a good reason for always recording a machine code program before trying it. The interrupt is disabled by using the SEI command. The address in locations $314 and $315, called the 'interrupt vector', is then altered. The address which will now be used is the one which is labelled AUTO, which is $C015. If you want this program to run at any other address, you will have to alter the bytes in lines 140 to 160 to suit. Normally, you would do this by altering the assembly address, but if you are operating on the DATA bytes in the BASIC loader, then it's more difficult. Once the address has been shifted, interrupts can be restored by using the CLI command. When this has been done, the effect of an interrupt will be to make the machine jump to address $C015. Lines 190 to 220 then

load the number bytes for the first line number into addresses 251 and 252. In this case, a starting number of 10 has been chosen. The RTS in line 230 then finishes off the set-up section of the program. Each interrupt will now cause the program that starts at $C015 to run, and we now have to examine this part of the code.

## The AUTO action

The first action of AUTO is to load the accumulator from address $C5. This is an address which stores temporarily a code for the last key that was pressed. This is *not* the ASCII code for the key, and Appendix A shows the codes that are used in this address. The use of the RETURN key leaves a '1' in this address, and this is detected in line 260. If the code is not '1', then line 270 sends the program back to the correct interrupt address of $EA31, labelled EXIT. If the last key that was pressed was the RETURN key, however, the AUTO action takes place, using line 280 onwards.

The first step is to load the X and A registers with the bytes of the line number. The X register is loaded with the low byte and the A register with the high byte, which will be zero at the start. These bytes are then stored in addresses $62 and $63, so that one of the machine's own subroutines can deal with them. This is a routine which will convert a number from two-byte form into a string of ASCII codes, something that no machine code programmer would particularly fancy writing for himself. To make this routine work, we have to start with loading the number $90 into the X register, and setting the carry bit. Calling subroutines $BC49, followed by $BDDD, will then carry out the conversion. The result of the conversion is to create a string which starts at address $0100. The sign (+ or −) is kept here, and the ASCII codes for the figures start at $0101. By loading $FF into the Y register and incrementing, we make the Y register store zero, and start a loop which reads from $0101 + Y, and stores at $0277 + Y. In other words, the ASCII codes are being read out of their temporary store into the keyboard buffer. This continues until a zero is read – the conversion programs place a zero at the end of the ASCII codes. We now place a zero in the keyboard buffer – the additional loading of A with a zero is not really necessary, but I like to play safe! Register Y now contains the count of bytes in the buffer, and this number is stored at address $C6. All of this ensures that the line number will be printed on the screen just as if you had typed it.

We haven't finished yet, though. Before we go back to the normal

routine, we have to update the line number for next time. This is done in lines 470 to 510. The number ten is added to 251, and if this causes a carry to be generated, then 1 has to be added to 252. The last part of the program is then a delay loop. The reason for this is that interrupts occur fifty times per second. Even if you smack the RETURN key pretty smartly, you will normally hold it down for a lot longer than one fiftieth of a second, so that it's possible to increment and print several line numbers in that time. By using a delay, brief though it is, you get time to release the key, ensuring that the numbers are printed in an orderly way. It takes quite a large delay loop to do this, such is the speed of machine code! Finally, the program jumps to address $EA31, which is the correct interrupt address for this type of interrupt.

This program can be placed in any part of the memory which is free, providing that you alter the addresses in $314 and $315 to suit. It could be modified so that pressing, for example, CTRL A, stopped the AUTO action, rather than the clumsier use of RUN/STOP and RESTORE. The present program has the merits of being short and not too much to type in BASIC poke form (Fig. 4.2).

```
10 PRINTCHR$(147):PRINT:PRINTTAB(18)"AUT
O":AD=49152
20 PRINT:PRINT"PROVIDES AUTOMATIC LINE N
UMBERING IN":PRINT"TENS, STARTING WITH 1
0."
25 PRINT"PRESS STOP/RESTORE TO END."
30 FORN=0 TO 86:READ D%:POKE AD+N,D%:NEX
T
40 SYS49152:NEW
1000 DATA120,169,21,141,20,3,169,192,141
,21,3,88,169,10,133,251,169,0,133
1010 DATA252,96,165,197,201,1,208,57,166
,251,165,252,133,98,134,99,162
1020 DATA144,56,32,73,188,32,221,189,160
,255,200,185,1,1,153,119,2,208
1030 DATA247,200,169,0,153,119,2,132,198
,24,165,251,105,10,133,251,144
1040 DATA2,230,252,162,255,160,255,136,2
08,253,202,208,248,76,49,234
```

*Fig. 4.2.* A BASIC loader version of AUTO.

# 5
# Renumber BASIC Lines

One of the commands which users of other computers greatly treasure is a RENUMBER. During the development of a BASIC program, even careful planning can sometimes fail to avoid the need for some extra lines placed between existing lines. This leads to an untidy-looking program, with line numbers that do not ascend neatly in tens. Renumbering cleans all of this up, and makes the program much easier for another user to enter, particularly when using AUTO line numbering.

Now a *real* renumbering program is quite a long and formidable thing. It's not just a matter of changing the line numbers, you see. Each GOTO and GOSUB and THEN in a program is likely to be followed by a line number that will also have to be changed. Changing these numbers, which are in ASCII code, is much less simple than changing line numbers which are stored as two bytes. Just to give an example of the difficulties, if a GOTO75 is to become a GOTO1200, two more bytes of memory are now needed to store the two extra codes. This means that the whole of the program from this point is going to have to be shifted two places higher in the memory! The size of program that is needed to do all this properly for a program that uses a lot of GOTOs and GOSUBs is well beyond the size that you could be expected to type accurately.

This, then, is a compromise 'skeleton' renumbering program. It will renumber the lines *only*, ignoring GOSUBs and GOTOs, so that you will have to renumber these for yourself by editing. Its simplicity makes up for its lack of sophistication, because it is reasonably short and simple to use. It is particularly useful for assembly language programs that have been written for MIKRO, because these, of course, contain no GOTO or GOSUB statements. It's also very useful for DATA lines which you get from the machine code to DATA program (see Fig. 3.1), because these also need no editing afterwards. It would undoubtedly be a pain to use in a program which contained a lot of GOTO and GOSUB statements.

## How to use it

Figure 5.1 shows the BASIC version of the program. This contains reminders about the limitations of the program, and asks for a

```
63000 PRINTCHR$(147):PRINTTAB(16)"RENUMB
ER"
63010 PRINT:PRINT"REMEMBER THAT THIS DOE
S NOT RENUMBER"
63020 PRINT"GOTO AND GOSUB LINES. MAKE A
 NOTE OF"
63030 PRINT"WHERE EACH GOTO AND GOSUB GO
ES TO, AND"
63040 PRINT"EDIT THESE IN LATER. DO NOT
TRY TO USE"
63050 PRINT"VERY HIGH LINE NUMBERS, MORE
 THAN THE"
63060 PRINT"LIMIT OF 63999."
63070 PRINT:PRINT"STARTING LINE NUMBER (
LESS THAN 1000)-"
63080 INPUT S:IF S>1000 OR S<1 THEN GOSU
B63180:GOTO63070
63090 PRINT:PRINT"INCREMENT (LESS THAN 2
56)-"
63100 INPUT C%:IF C%>255 OR C%<1 THEN GO
SUB63180:GOTO63090
63110 PRINT:PRINT".......PLEASE WAIT....
....."
63120 AD=49152:X%=INT(S/256)
63130 POKE828,S-X%*256:POKE829,X%:POKE83
0,C%
63140 FOR N=0 TO 59
63150 READ D%:POKE AD+N,D%:NEXT
63160 SYS49152
63170 END
63180 PRINT"INCORRECT NUMBER- PLEASE RE-
DO":RETURN
63900 DATA24,165,43,133,251,165,44,133,2
52,160,0,177,251,72,200,177,251
63910 DATA72,200,173,60,3,145,251,200,17
3,61,3,145,251,173,60,3,109,62,3
63920 DATA176,16,141,60,3,104,133,252,10
4,133,251,208,216,165,252,208,212
63930 DATA96,238,61,3,24,144,234
```

*Fig. 5.1.* The BASIC form of the line renumber program.

starting line number and increment. The starting line number should be no more than 1000, and the increment should be less than 255. Normally, you would start with 10 and increment by 10. The program, to save space, contains no safeguards against 'silly' renumbering, meaning choosing values which would make the later lines of the program have values greater than 63999, the normal limit. It's up to you to choose sane numbers! Once you have done so, the machine code bytes are poked into place, and the renumbering is carried out. The renumbering program uses lines 63000 onwards so that this program can be merged with the program that you want to renumber. An alternative is to remove line 63160, delete the BASIC part of the program, and then load in the BASIC program that you want to renumber. Addresses 828 to 830 will then have to be poked, and typing SYS49152 will then carry out the renumbering.

## How it works

Figure 5.2 shows the assembly language version of this program. Decimal addresses 251 and 252 are used to hold the address numbers of the lines of BASIC, and addresses 828, 829 store the line numbers in low byte, high byte form. Address 830 is used to store the increment number. These addresses 828 to 830 are part of the cassette buffer memory of the C 64, and they will be altered if a program is loaded from cassette or stored on cassette. This is why you will have to re-poke these addresses if you delete the BASIC loader and then load in the program that you want to renumber.

The program starts in lines 90 to 120 by placing the start-of-BASIC address into 251 and 252. From this point, the action consists of a long loop which starts in line 130. By using register Y as an index, the first two bytes of a BASIC line are placed on the stack in the order low byte, then high byte. These, remember, are the numbers which give the address of the start of the *next* BASIC line, and they will eventually replace the numbers which are at present stored in 251 and 252. The Y register is then incremented again, so that it points at the third byte in the line. This is the low byte of the line number, and it's the first byte that will have to be changed in this line. The accumulator is loaded with the low byte of the new starting line number from address STL (=808), and this number is stored as the third byte in the line. Y is incremented again and the same action used to store the high byte of the new line number. Line 250 then starts the action of changing the numbers in 808 and 809 to the next value. The

```
 20 C000          *=$C000
 30 C000          ADL          =  251
 40 C000          ADH          =  252
 50 C000          STL          =  828
 60 C000          STH          =  829
 70 C000          INC          =  830
 80 C000 18                    CLC
 90 C001 A52B                  LDA 43
100 C003 85FB                  STA ADL
110 C005 A52C                  LDA 44
120 C007 85FC                  STA ADH
130 C009 A000    LOOP          LDY #0
140 C00B B1FB                  LDA (ADL),Y
150 C00D 48                    PHA
160 C00E C8                    INY
170 C00F B1FB                  LDA (ADL),Y
180 C011 48                    PHA
190 C012 C8                    INY
200 C013 AD3C03                LDA STL
210 C016 91FB                  STA (ADL),Y
220 C018 C8                    INY
230 C019 AD3D03                LDA STH
240 C01C 91FB                  STA (ADL),Y
250 C01E AD3C03                LDA STL
260 C021 6D3E03                ADC INC
270 C024 B010                  BCS BUMP
280 C026 8D3C03  NXT           STA STL
290 C029 68                    PLA
300 C02A 85FC                  STA ADH
310 C02C 68                    PLA
320 C02D 85FB                  STA ADL
330 C02F D0D8                  BNE LOOP
340 C031 A5FC                  LDA ADH
350 C033 D0D4                  BNE LOOP
360 C035 60                    RTS
370 C036 EE3D03  BUMP          INC STH
380 C039 18                    CLC
390 C03A 90EA                  BCC NXT
```

*Fig. 5.2.* The assembly language form of the program.

accumulator is loaded from STL (line 250), and the value of INC is added. If this causes a carry, the branch to BUMP increments STH, and returns with the carry cleared (the CLC is not strictly necessary, but it's a good precaution). The new low byte number is stored back

into STL, so that STL and STH now contain the two bytes of the next line number.

Now we need to get to the start of the next line to carry out the action all over again. Lines 290 to 320 recover the address bytes for the next line from the stack, and place them into ADL and ADH. Because the low byte was stored on the stack first, it is recovered last. If this low byte is not zero, we certainly haven't reached the end of the program, and we loop back. If the low byte is zero, then we have to test the high byte. If the high byte is zero, we have reached the end of the program, and this is detected by line 350. If the high byte is not zero, then the program loops back to renumber the next line. The two tests are needed, because the end of the program is marked by having both the high and the low byte of the address number both equal to zero, not just one of these bytes.

# 6
# Delete a Block of Lines

When you design your own programs, you very often find that a subroutine which you have used is likely to be of use in a lot of other programs. The subroutine might be, for example, in your lines 5000 to 5250. To isolate it from the rest of the program might mean deleting lines 10 to 4500 and 6000 to 10000. On other machines, this is perfectly simple. You simply type DELETE 10–4500 (RETURN) and then DELETE 6000–10000 (RETURN) and it's done. The C 64, alas, has no block DELETE command, and you will be forced to type each line number, then RETURN, until your typing fingers ache with the effort. You will then find that because you were not concentrating, you have deleted the lines that you wanted as well...

The action of deleting blocks of lines can be carried out using machine code, using a method which is very similar to the AUTO line number program (No. 4). The same methods can be much more simply used in a BASIC program, however. Though the deleting action is very much slower, it does have the advantage of giving you time for second thoughts, so that you can stop the action by pressing the RUN/STOP key if you have decided that you want to save some other lines after all. The method that is used is to poke bytes into the keyboard buffer, a method that was explained in the AUTO program. This time, however, the poking is done from a BASIC program, and it is used to create a new line of program on each loop as well as to delete a line of the main program. As usual, this subroutine has been numbered 63000 on so that it can be merged with the program whose lines you want to delete.

### How to use it

When the DELETE program has been merged, run it by typing GOTO63000. You should not type RUN, remember, because this

would cause the main program to run. You will then be prompted for the first and last line numbers in the block that you want to delete. A minor inconvenience of this program is that it assumes that your lines are numbered in tens, but with the AUTO numberer and the renumber programs behind you, this is hardly a bother! When you enter the start and finish numbers, you will see a line numbered 63040 flashing on the screen with different line numbers. These show the progress of the deleting action, and when the READY prompt returns, the action is complete. Note that when you type the program into the machine, you *should not omit* line 63040.

## How it works

In line 63000 of Fig. 6.1, the start and finish line numbers are allocated to variables S and F respectively. Line 63010 clears the

```
63000 INPUT"START NO., END NO. ";S,F:PRI
NTCHR$(147)
63010 PRINTCHR$(19)S:S=S+10:IF S>F+10THE
N63060
63020 PRINT"63040S="S":F="F:PRINT"GOTO63
040"
63030 POKE631,19:FORN=632TO634:POKEN,13:
POKE198,4:NEXT:SYS42115
63040 REM DUMMY LINE
63050 GOTO63010
63060 POKE198,0:PRINTCHR$(147):END
```

*Fig. 6.1.* The DELETE program.

screen and homes the cursor because of the action of CHR$(19), and then prints the first line number, allocated to S. The second part of this line then adds 10 to the value of S, so as to form the next line number. The last part of the line tests for the line number exceeding the limit, so that the program can be stopped. After this comes the fancy stuff, and as before, it's all concerned with the C 64 keyboard buffer action.

The next line, 63020, prints some very odd things. To start with, it prints the line number 63040, and then follows this with S=. Outside the quotes, variable S is allocated, and then ":F=" will cause another piece of printing, with another F once again outside the quotes. The line ends with printing GOTO63040. The next line 63030 then pokes numbers into the keyboard buffer. The numbers are one 19 (clear and

home cursor), followed by three RETURN codes of 13. The size of the keyboard buffer in address 198 is poked with 4, so that the machine will empty four numbers from the buffer and act on them. The SYS42115 is a 'warm start' address, which forces the machine to clear the buffer and carry out anything that is waiting to be done.

Well, what is to be done? The first RETURN code acts on the first line of printing on the screen, which is S, the line number. If S is 10, then what appears is 10, then the carriage return from the buffer causes this line to be deleted, just as if you had typed 10 (RETURN). The next RETURN causes the line 63040 to be entered, printing S= followed by a value. You can't omit this, because if you do, the value of S is lost. The use of ":F="F will similarly preserve the value of F in the program when the warm-start occurs. The last RETURN then causes the action GOTO63040 to be carried out, so that this line, having been entered into the memory by the *previous* return, is now carried out. It replaces the dummy line which was put there originally. Having reallocated values of S and F with line 63040, the program then moves to line 63050, which returns the action to line 63010 again. This causes another value of S to be selected, and the whole process repeats. When no more lines remain to be deleted, the GOTO 63060 in line 63020 causes the address 198 to be cleared, so that nothing more can be read from the keyboard buffer. That's it – your lines have been deleted.

If you have a program which has line numbers scattered all over the place, with odd numbers that are not in tens, don't despair. You can replace the S=S+10 in line 63010 with S=S+1. The program will then increment line numbers by 1 each time, whether there are lines with these numbers or not. The only disadvantage of this is that it's rather slow, so you can slope off and make yourself a coffee while your lines are being deleted. Even so, it certainly beats having to type in each line number and then press RETURN!

# 7
# Sound Off!

The SOUND commands of the C 64 are, like so many others, very complicated to use, and depend on a large number of POKE commands. For simple musical sounds, it's possible to listen and 'edit' the sound by making use of a sound editor program, included in this book. For complicated sound effects, particularly when more than one channel is to be used, one of the problems is that you have to carry out a lot of pokes to clear out a sound command. Now if you don't clear out all the sound POKEs, you may achieve silence, but still leave several numbers stored in the memory which will make your next attempt at a sound go wrong. What you really need to do is to poke a zero into all of the sound register numbers, and ideally, you should be able to do this with just one keypress.

## How to use it

Well, if you can press two keys at once, this is certainly possible, given a bit of machine code magic. This program allows you to switch off

```
100 PRINTCHR$(147):PRINT:PRINTTAB(16)"SI
LENCER"
110 PRINT:PRINT"ENABLES CTRL-X TO SWITCH
 OFF SOUND":A=49152
120 FOR N=0 TO 39:READ D%:POKE A+N,D%:NE
XT
130 SYS A:NEW
1000 DATA169,11,141,143,2,169,192,141,14
4,2,96,72,165,197,201,23,208,18
1010 DATA173,141,2,201,4,208,11,169,0,17
0,157,0,212,232,224,29,48,248,104
1020 DATA76,72,235
```

*Fig. 7.1.* The BASIC form of SOUND-OFF.

all sound commands by pressing CTRL and X keys together. The BASIC program, in Fig. 7.1, pokes the machine code numbers into memory, starts the machine code operating, and then deletes itself. You can 'disconnect' the program by pressing RUN/STOP and RESTORE together, but for as long as you have it in the memory, you can cancel all sound instructions with CTRL-X. Type the program, not forgetting to check that you have 40 numbers in the DATA lines, and *save it*. This is essential, because the program deletes itself after loading in the machine code. Then try some sound commands. When you have a sound playing, press CTRL-X. The sound should stop at once. Instant control at last!

## How it works

As usual, the assembly language listing of Fig. 7.2 shows how this works. Basically, it's very simple, because all that has to be done is to poke a zero into each of the sound register numbers. What makes it a little more complicated is linking the program in so that the CTRL-X key combination will make the code run. In this example, it's done by

```
100 C000           *=$C000
110 C000 A90B                  LDA #<PROG
120 C002 8D8F02                STA $28F
130 C005 A9C0                  LDA #>PROG
140 C007 8D9002                STA $290
150 C00A 60                    RTS
160 C00B 48        PROG        PHA
170 C00C A5C5                  LDA 197
180 C00E C917                  CMP #23
190 C010 D012                  BNE EXIT
200 C012 AD8D02                LDA 653
210 C015 C904                  CMP #4
220 C017 D00B                  BNE EXIT
230 C019 A900                  LDA #0
240 C01B AA                    TAX
250 C01C 9D00D4    LOOP        STA 54272,X
260 C01F E8                    INX
270 C020 E01D                  CPX #29
280 C022 30F8                  BMI LOOP
290 C024 68        EXIT        PLA
300 C025 4C48EB                JMP $EB48
```

*Fig. 7.2.* The same program in assembly listing.

making use of the addresses $28F and $290. These give the address of the routine which decodes the keyboard, a routine which is normally located at $EB48. This keyboard decode routine is used each time a key is pressed, and it's another way of getting another command inserted into the BASIC of the C 64. We can change the address in $28F and $290, because this is in RAM. Having made this address into one that we can write, we then have to make sure that whatever byte is in the accumulator is preserved, and is passed to the routine at $EB48 when we are finished. Provided that we do this correctly, the routine is 'transparent' - you don't know it's there until you press CTRL-X.

The first part of the program simply changes the address numbers in $28F and $290. These are changed to the address PROG, which in this example is $C00B. If you want to relocate this program, you will have to make sure that you alter these two address bytes. Each time a key is pressed now, the routine which starts at PROG will be run. At the end of the routine, or if the key that has been pressed is of no interest, the program will return to the address $EB48, which is the address that is normally held in $28F and $290.

PROG starts by pushing the accumulator byte on to the stack. This byte will not be an ASCII code, but an internal code number, and it has to be preserved to be used by the decoding routine. Having saved it in this way, we then load the accumulator from address 197 ($C6 in hex). This is the 'last key pressed' address, and we can then test what has been loaded from this address. The X key gives the result '23' here (*not* ASCII codes, remember), and lines 180,190 test for this value. If the X key has not been struck, the program jumps to EXIT, to proceed as normal. If the X key was struck, we now need to find if the CTRL key was held down as well. This is done by loading from address 653 and looking for a '4' in the accumulator. Once again, if this value does not appear, the CTRL key was not held down, and the program goes to EXIT. If both X and CTRL have been used, however, both tests are passed, and line 230 starts the sound-delete action. The accumulator is zeroed, and the same value passed to the X-register. In the loop which follows, the address of the first sound register, 54272 is used as a base address, and X as the index. The zero value in the accumulator is stored in the address, and then X is incremented. This continues until X reaches 29, at which point the loop breaks off, the accumulator value is pulled from the stack, and the program jumps to $EB48. This is not the simplest possible method; it would have been neater to start with X=29 and decrement X, using a BNE to detect the end of the loop. It just shows that the best version of a program is always the 26th or so!

# 8
# OLD Times Remembered

It's a familiar scenario for anyone who has ever laboured over a program past the hour of midnight. You have just seen the last SYNTAX ERROR, corrected the last loop, and your program has worked in all its glory. You prise your eyes open, type NEW – and suddenly remember that you didn't record it. It's gone, and all of your work with it – or is there still hope? If you use the BBC Micro or the Electron, then you simply type OLD, press RETURN, and you have your program back again. Wouldn't it be useful to have something like this for the C 64?

It would, and it's not too difficult. The only snag is that it needs a machine code program which has to be in the memory *before* you need it. It *is* possible to load it after doing a NEW on a program, but I can't guarantee that it would work then, because you would have to append the program on to the end of the one which has just been NEWed, and the ordinary MERGE routine doesn't work so simply in these circumstances. The problem is that the address for the end of the BASIC program has been reset by the NEW action. Let's assume, then, that you have had the foresight to make sure that this routine was in place!

Let's see why all this restoration work is possible. When you type NEW and press RETURN, you don't clear the memory of the C 64 (or any other computer). All that you do is to make the first two bytes of the program memory into zeros. Now these first two bytes are normally the address bytes for the next line. When they're both zero, this is a signal to the computer that there is no next line, the program is at an end. If these zero bytes are at the start of a program, it signals that the computer has no program in place. All the other bytes of the program are in the memory, along with the variable list table and everything else. The effect of having the first two bytes put to zero, however, is to alter all of the BASIC 'pointers'. The start-of-BASIC address in 43 and 44 remains unchanged, but the end-of-BASIC

address in 45 and 46, and other pointers (for arrays and strings) in 47,48 and 49,50 are also reset – all with the address of the *third* byte, the one which follows the two zeros.

If you want to restore a program after NEW, then, you have to poke in the correct address bytes for the second line of the program, and then restore all of the pointers. You must do this without loading another program into the same part of memory, and without typing any new lines. You can, however, type direct commands – it's anything with a line number that you have to avoid. You can only MERGE another program in if you make a guess about where the old program might have ended. Very often, if you set the start of BASIC to a very high value, you will get away with entering a program that can repair a NEW. To be on the safe side, though, it's better to enter the program first, before you start a marathon computing session. The program is one which deletes itself after poking the machine code into place, so you can start on your own program immediately you see the READY prompt.

## Using the program

The BASIC program is shown in Fig. 8.1. As usual, you should type it carefully, check it, particularly the data lines, and make sure that you have 72 items of data. Then record the program at least once. When you RUN the program, it will poke memory, and then delete itself. Check that it has deleted by typing LIST (RETURN), and then

```
10 PRINTCHR$(147):PRINT:PRINTTAB(17)"UN-
NEW":A=49152
20 PRINT:PRINT"RUN THIS PROGRAM":PRINT"T
YPE SYS49152 TO RECOVER OLD PROGRAM"
30 FOR N=0 TO 71:READ D%:POKE A+N,D%:NEX
T
40 NEW
1000 DATA162,4,232,189,0,8,208,250,232,1
42,1,8,169,8,168,141,2,8,138,208
1010 DATA31,152,208,28,160,0,230,251,240
,38,200,192,2,208,247,165,251,133
1020 DATA45,133,47,133,49,165,252,133,46
,133,48,133,50,96,134,251,132,252
1030 DATA160,0,177,251,170,200,177,251,1
68,76,18,192,230,252,208,214
```

*Fig. 8.1.* The program for recovering from NEW, in BASIC loader form.

type SYS49152 (RETURN). When you type LIST (RETURN) again, you should find that your program has been recovered. If the program looks corrupted, or if the machine locks up, then you will have to switch off, reload the program, and check the DATA even more carefully. When the program is operating correctly, make sure that you have several copies on tape, and load one in each time you start programming, unless you are going to use other utility programs in the same part of the memory. If you type NEW on a BASIC program that you really wanted to keep, then SYS49152 will be your salvation.

This program is relocatable, and can be poked without modification of DATA lines into any part of memory which is not otherwise engaged. You have to remember to use the correct starting address following SYS, however. Note that this program *cannot* be used to un-NEW a program which has been shifted and which does not start at the normal beginning of BASIC at 2049.

## How it works

The assembly language version of the program is illustrated in Fig. 8.2. The first part of the operation is concerned with finding the correct address for the second line of the program. If you take the address 2048 as the start of program memory, address 2048 itself *always* contains a zero, and for a program which has been NEWed, addresses 2049 and 2050 will also contain zero bytes. The low byte of the line number, in 2051, will also be a zero unless the first line of the program had a line number of more than 255. Following that, the addresses in the memory will contain code numbers, and there won't be another zero until the last byte of this first line. If we can find this address, the address of the next line is one step beyond.

The program starts, therefore, by loading the X register with 4. This is going to be used as an index to the address 2048, in a loop which starts at the label MORE. At this point, X is incremented, which on the first pass will make the stored value equal to 5. The accumulator is then loaded from 2048+5=2053, which is the first byte of code of the program, having skipped over the first zero byte, the two zeros that have replaced the next line address, and the two line number bytes. Line 140 then tests to see if this byte is zero. If it isn't (and it certainly will not be on the first pass), the program returns to MORE, incrementing X and trying again. This loop continues

```
100 C000          *=$C000
110 C000 A204                 LDX #4
120 C002 E8       MORE        INX
130 C003 BD0008               LDA 2048,X
140 C006 D0FA                 BNE MORE
150 C008 E8                   INX
160 C009 8E0108               STX $801
170 C00C A908                 LDA #8
180 C00E A8                   TAY
190 C00F 8D0208               STA $802
200 C012 8A       VAR         TXA
210 C013 D01F                 BNE NXT
220 C015 98                   TYA
230 C016 D01C                 BNE NXT
231 C018 A000                 LDY #0
232 C01A E6FB     LOOP        INC 251
233 C01C F026                 BEQ CARRY
234 C01E C8       RTN         INY
235 C01F C002                 CPY #2
236 C021 D0F7                 BNE LOOP
240 C023 A5FB                 LDA 251
242 C025 852D                 STA 45
243 C027 852F                 STA 47
244 C029 8531                 STA 49
245 C02B A5FC                 LDA 252
246 C02D 852E                 STA 46
247 C02F 8530                 STA 48
248 C031 8532                 STA 50
260 C033 60                   RTS
270 C034 86FB     NXT         STX 251
280 C036 84FC                 STY 252
290 C038 A000                 LDY #0
300 C03A B1FB                 LDA (251),Y
310 C03C AA                   TAX
312 C03D C8                   INY
320 C03E B1FB                 LDA (251),Y
330 C040 A8                   TAY
340 C041 4C12C0               JMP VAR
370 C044 E6FC     CARRY       INC 252
380 C046 D0D6                 BNE RTN
```

*Fig. 8.2.* The assembly language program.

until the byte which has been loaded into the accumulator in line 130 is a zero.

When this happens, 2948+X is pointing to the last byte in the first

line. By using INX in line 150, we make the address point to the first byte of the next line, the address that we need. Now in hex, the start of BASIC memory is $800, and the address that we have reached now is $800+X. The number in X is therefore the low byte of the address of the start of the second line, and we can store this value in $801, decimal 2049. The high byte of this address is easier to deal with, because it must be 8 if BASIC begins at its usual place. This number is put into address $802, and is also temporarily stored in the Y register for later use.

That deals with the resetting of the second line address. For a lot of computers, no more than this is needed, because the position of variables is settled when the program is run. For the C 64, however, these pointers must be set *before* the program can be run or listed, and the next section of the program does just that. What we have to do is to skip through the bytes of the program, looking for the end which is marked by the sequence of three zeros. We can then set the next address as the pointer address in the three sets of pointers. Once again, this has to be done by setting up a loop. At the start of line 200, the low byte of the present address is in X and the high byte is in A. First time round, this will be the address of the start of the second line of program. We then test these bytes. One of them might be zero, but unless we have reached the end of the program, they will not *both* be zero. Until we find that both are zero, then, the program jumps to the section which is labelled NXT. This stores the start-of-line address at 251 and 252, and then uses indirect Y addressing on these numbers. What this means is that we load the accumulator from the address in 251 and 252, indexed to Y. This gets the low byte of the *next* line address into X and the high byte into Y. Since this is the same use of X and Y as before, we can go back to VAR to test for the end of the program. This loop will continue, jumping from the start of one line to the start of the next, until the last line is found and the loop is broken. The next line of the assembly language program is then 231.

At line 231, the Y register is zeroed. We have to move two steps on from the address which is presently stored in 251 and 252, but with some care. Adding 2 to 251 might generate a carry into 252 if, for example, the byte in 251 happened to be 254 or 255. We use lines 232 to 236 to increment Y and also address 251, checking for a carry, and incrementing 252 if there is one. When Y has been incremented to 2, the loop is broken, and the last part of the program starts in line 240. The accumulator is loaded from 251, which now contains the correct low byte of the end-of-program address. This number is then placed

into 45,47 and 49. The accumulator is then loaded from 252 to get the high byte, and this is placed in 46,48 and 50. That's it – the RTS then returns you to normal BASIC, and your program is restored.

# 9
# These FUNCTION Keys

If you have ever watched the owner of a BBC Micro, you will have been impressed by the function keys. The owner types:

*KEY1 PRINT (RETURN)

and from then on, every time the F1 key is pressed, the machine shows the word PRINT. It can be a great saving of time, particularly when the keys are programmed for common words like LIST, PRINT, TAB and all the others that you use when you are typing in a program. Users of the Colour Genie, the Amstrad, and to a lesser extent, the Spectravideo, have the same good fortune. Where does that leave you, the C 64 owner?

The answer is - puzzled. The C 64 has a set of four F-keys, but nothing that you can do using BASIC will program them. A lot of owners believe that these keys are purely ornamental, but they can, in fact, be programmed. The difference between the C 64 and the others is that the C 64 contains no machine code instructions for programming these keys. When one of these keys is pressed, it returns a code number into the machine - but there are no instructions about what has to be done with this code! If you want to use these keys, then you have to write code to program them for yourself.

This routine shortens your effort, and also shows how the keys can be programmed. The routine is part BASIC, part machine code, and has deliberately been kept simple. For example, only commands of up to 30 characters are catered for, though you could (if you are a skilled machine code programmer) alter the routine to allow for many more than this. In addition, as it stands at the moment, it will not accept multistatement lines like PRINT:PRINT. The modifications that are needed to accept this are much simpler, and only to the BASIC portion. The difficult part is the machine code, and you should not try to alter this unless you have an assembler or some experience.

## How to use it

This is a fairly long program (Fig. 9.1) for a utility, and because of the machine code section your typing has to be accurate. In addition, because the program deletes itself after the machine code has been poked in place, you will have to ensure that you have at least one perfect copy on tape before you try to use the program. When you run this routine, you will be asked for commands in turn for each of the function keys. The odd-numbered keys 1,3,5,7 are requested first,

```
10 PRINTCHR$(147):PRINT:PRINTTAB(17)"F-K
EYS":AD=49152
20 PRINT:PRINT"PLEASE TYPE YOUR COMMAND
FOR EACH KEY."
22 PRINT"MAX. OF 30 CHARACTERS, PLEASE"
25 FORN=49664 TO 49920:POKE N,0:NEXT
30 FOR N=1 TO 7 STEP 2:A=49664+32*INT(N/
2)
40 GOSUB 1000:NEXT
50 FOR N=2 TO 8 STEP 2:A=49664+32*(N/2+3
)
60 GOSUB 1000:NEXT
70 FOR N=0 TO 72
80 READ D%:POKEAD+N,D%:NEXT
90 SYS49160
100 END
1000 PRINT"KEY ";N;" = ";
1010 INPUT A$:L=LEN(A$)
1012 IF L>30THENPRINT"TOO LONG-PLEASE TR
Y AGAIN":GOTO1010
1020 FOR X=1TO L
1030 POKEA+X-1,ASC(MID$(A$,X,1)):NEXT
1040 POKEA+X-1,13
1050 RETURN
5000 DATA0,32,64,96,128,160,192,224,120,
169,29,141,20,3,169,192,141,21
5010 DATA3,88,169,0,133,251,169,194,133,
252,96,32,159,255,166,198,240,34
5020 DATA202,189,119,2,201,133,144,26,20
1,141,176,22,56,233,133,168,185
5030 DATA0,192,168,232,177,251,240,9,157
,119,2,134,198,232,200,208,243
5040 DATA76,49,234
```

*Fig. 9.1.* Programming your function keys – the BASIC program.

because these are the keys which are easiest to use. Following this, the four even-numbered keys are allocated – in use, you have to press the SHIFT key along with the function key. It makes sense, then, to have the commands which you will use most often allocated to the odd-numbered keys. The commands are obtained using an INPUT step, so that pressing RETURN terminates the command. Because of this, it is impossible to put in a command which contains a comma or a colon because these will be rejected by the INPUT step in line 1010. If you don't want to allocate a key, just press RETURN instead of typing a command. When all eight keys have been allocated, the commands are poked into memory, the machine code is also poked in place, and the BASIC program deletes itself. Now, however, when you press a function key, you will see the command which you have programmed for it appear. If you want to change a command, you need only poke different codes into place, so you can use a shorter version of the BASIC program, with lines 70 to 90, and the DATA lines, omitted.

## How it works

In this example, there are two parts to how it works, the BASIC and the machine code. The BASIC part performs the actions of allowing commands to be input, and then pokes the ASCII codes of the letters of the commands into memory. The memory that is used lies between 49664 and 49920, which is above the part used by the machine code. There is plenty of space here, so larger numbers of characters can be poked, provided that the allocation of memory is correctly made. Allocation of memory is done in lines 30 and 50 by the formulae shown in these lines. These formulae use a 32-character line, because the character following a command has to be a carriage return, CHR$(13), and this must be followed by a zero. The formulae cause all the data for the odd keys to be stored in sequence, followed by the data for the even-numbered keys. This is something which is much easier to arrange in BASIC than in machine code, and it also makes the task of the machine code section of the program much easier.

The machine code section (Fig. 9.2) contains the codes which cause the keys to be useful. The technique here is to break into the interrupt sequence, as we did for the AUTO-number program. This allows the pressing of a function key to be detected, and when the key is identified, its corresponding section of data in memory can be located. The ASCII codes that are stored in high memory are then

```
100 C000          *=$C000
110 C000 002040                 BYT 0,$20,$40,$60,
                                $80,$A0,$C0,$E0
120 C008 78       START         SEI
130 C009 A91D                   LDA #<KEY
140 C00B 8D1403                 STA $0314
150 C00E A9C0                   LDA #>KEY
160 C010 8D1503                 STA $0315
170 C013 58                     CLI
180 C014 A900                   LDA #0
190 C016 85FB                   STA 251
200 C018 A9C2                   LDA #$C2
210 C01A 85FC                   STA 252
220 C01C 60                     RTS
230 C01D 209FFF KEY             JSR $FF9F
240 C020 A6C6                   LDX $C6
250 C022 F022                   BEQ EXIT
260 C024 CA                     DEX
265 C025 BD7702                 LDA $0277,X
270 C028 C985                   CMP #133
280 C02A 901A                   BCC EXIT
290 C02C C98D                   CMP #141
300 C02E B016                   BCS EXIT
310 C030 38                     SEC
320 C031 E985                   SBC #133
330 C033 A8                     TAY
340 C034 B900C0                 LDA $C000,Y
350 C037 A8                     TAY
360 C038 E8                     INX
370 C039 B1FB     LOOP          LDA (251),Y
380 C03B F009                   BEQ EXIT
390 C03D 9D7702                 STA $0277,X
400 C040 86C6                   STX $C6
410 C042 E8                     INX
420 C043 C8                     INY
430 C044 D0F3                   BNE LOOP
450 C046 4C31EA EXIT            JMP $EA31
```

*Fig. 9.2.* The assembly language for the machine code which is included in the BASIC program.

transferred to the keyboard buffer. At the end of the interrupt, then, these codes are treated just as if the command had been typed directly, and another RETURN will carry out the command if needed. If you are using the function keys to provide words like

PRINT or TAB in a program, of course, you will have typed a line number and you will follow PRINT or TAB with other data. Unless you are accustomed to writing machine code, it's not easy to adapt AUTO to work with this program, because both use the interrupt. The method is to test for the carriage return, as in AUTO, and if the key is not the RETURN key, continue on the tests for the function keys. The AUTO routine can then be placed starting at $C050.

Looking at the assembly language, then, the program starts by storing a set of additions to a base address. The base address is the start of the data for the keys, and the additions allow intervals of 32 bytes for each key. This part will have to be extensively modified if you want to use more than 30 characters (+ RETURN) in a key code. The true start of the program is labelled START, and consists of replacing the IRQ interrupt address bytes in addresses $314 and $315. While this is being done, interrupts have to be disabled, and they must be enabled again in line 170 after the changes have been made. The base address for the data is $C200, and this is then loaded into addresses 251 and 252 by lines 180 to 210. Once this has been done, the set-up routine is complete, and command returns to BASIC.

Once this has been done, the code which begins at address $C01D, line 230, labelled KEY, will run fifty times per second (sixty per second in the USA). The first action of this interrupt is to search for a key being pressed, using the subroutine $FF9F. If a key has been pressed, the keyboard buffer count will not be zero, and this count number is contained in address $C6. This address is tested in line 240, and if it contains zero, then the program jumps to EXIT, which takes it to the normal interrupt address of $EA31. If a key has been pressed, however, the program finds out which key it was. The X register is decremented, because its value is of one more than the total number of bytes in the buffer. The accumulator is then loaded from the buffer address, indexed to X.

The next step is to decide whether or not the key is a function key. If it is, then there is more to do, but if it isn't, we can go directly to EXIT. The tests are carried out in lines 270 to 300. The byte is compared with 133, and the EXIT path is taken if the byte is less than 133, because this is the code for the first F-key. The next test compares it with 141, and takes the EXIT path if the byte is more than 141. This covers the range of the F-keys, and if our key code has slipped through both of these tests, it must be the code for a function key.

Having ensured that we have the code for a function key in the accumulator, the next thing is to locate the correct piece of memory

to use. Each key will have a piece of memory from address $C200 on, allocated to it. Having selected to use 32 byte chunks of data makes this search very much easier. Lines 310,320 now subtract 133 from the key code, leaving a number which must be in the range of 1 to 7. This number is transferred to the Y register in line 330. Line 340 then loads a byte from address $C000, Y indexed. Now what is stored at $C000 is the set of eight addition bytes. If Y contains 1, for example, then the number $20 is loaded, if Y contains 2, then $40 is loaded and so on. This new number, 0, $20, $40 or whatever is then put into the Y register in line 350, and then the X register is incremented so as to point to a vacant part of the keyboard buffer.

The loop which starts in line 370 will then load from the data address of $C200, Y indexed. This is done by using indirect Y addressing, with the address of $C200 held in addresses 251 and 252. These are the spare zero-page addresses of the C 64. The bytes are loaded into the accumulator and stored in the keyboard buffer, using X indexing for the keyboard buffer address. The end of a command is detected by the zero which follows the 13 at the end of each command. These zeros were placed by the BASIC program, and this is why a maximum of 30 characters can be used – 30 characters of command, plus a carriage return, and a zero left at the end uses up our quota of 32 characters per key. To go to a larger number, you will have to be prepared to put up with much more involved addressing methods. When the zero byte is found, the loop is broken, and the program returns to the normal IRQ address of $EA31 at EXIT.

In practice, the limit of 30 characters is not irksome, and it permits much shorter and simpler code than could be used if a greater number of characters had to be employed. The most awkward part is that a multistatement line cannot be entered. This, however, is purely due to the use of an INPUT line in the BASIC. If characters are entered by using a GET A$ type of command, with A$ tested and added to a longer string, then pressing RETURN can be used to terminate the string, but colons will not. This would allow a multistatement line to be entered, and the changes to this part of the BASIC are comparatively simple to make.

# 10
# Sound Your Keys!

Several computers feature a keybeep, which means that a short beep sounds whenever a key is struck. It's quite useful for the type of machine which has a dodgy keyboard, and also if you are unaccustomed to the feel of a real keyboard. My own C 64 has a sticky '2' key, and it works only if it is thumped really hard. Because I was tired of typing 0 in place of 20, and omitting quote marks in PRINT statements, I wrote this routine which makes each key-press sound a beep in the speaker of your TV. This routine has some odd side-effects, but they are comparatively harmless. It has certainly eased the problem of the sticky key, and if you like to concentrate on the keyboard and ignore the screen as you type, then it could be very useful for you. The program is, once again, one which pokes codes into the memory and then deletes itself.

```
10 PRINTCHR$(147):PRINT:PRINTTAB(16)"KEY
BEEP"
20 PRINT:PRINT"......PLEASE WAIT......":
A=49152
30 FOR N=0 TO 65:READ D%:POKE A+N,D%:NEX
T
40 SYS49152:NEW
1000 DATA169,11,141,143,2,169,192,141,14
4,2,96,72,169,15,141,24,212,169
1010 DATA9,141,5,212,169,81,141,1,212,16
9,97,141,0,212,169,33,141,4,212
1020 DATA169,50,133,251,133,252,198,252,
208,252,198,251,208,246,169,0,141
1030 DATA4,212,141,5,212,141,6,212,104,7
6,72,235
```

*Fig. 10.1* The keybeep program in BASIC.

## How to use it

To use the keybeep, type the program as listed in Fig. 10.1, and be sure to check it carefully. There should be 66 items in the data lines. Record the program because it will delete itself after running. When the program has run, you will find that pressing a key causes a sound. If you hear nothing, it might be because the volume control of your TV is turned down. If you are one of the select few who use a monitor in place of a TV, then this program is not for you unless you can hook up a sound amplifier and loudspeaker to the C 64 – it does, in fact, provide for this. If you don't like the pitch of the sound, then you will have to alter the machine code, and I have indicated the address in Fig. 10.2.

---

Sound pitch depends on bytes in addresses 54273 and 54272. The number in 54273 exerts more control, and the other number is for fine adjustments only. The two numbers which are used in this program are 81 and 97, of which 81 is the main control number. This is the sixth item in DATA line 1010. The fine control number 97 is the eleventh DATA item in the same line.

---

*Fig. 10.2.* How to alter the pitch of the keybeep.

## How it works

Compared with the F-key program, the assembly language listing (given in Fig. 10.3) is fairly simple. Each time the machine detects a key being pressed, it switches to a routine which pokes the sound registers, waits for a moment, then pokes zeros in place to shut off the sound. All that we need to look at, then, is how the key-press is detected and how the sound registers are used. These methods may be of considerable interest to you if you want to develop machine code routines of your own in future.

The address of the start of the keybeep routine is $C00B, and this is poked into addresses $28F and $290. These addresses are for the keyboard decode routine, which is located at $EB48, and it is to this address that the routine must return after each beep has been sounded. Once the new routine address has been put into place, the control returns to BASIC until a key is pressed.

When a key is pressed, the new address in $28F and $290 will cause

```
100 C000           *=$C000
110 C000 A90B                  LDA #<BEEP
120 C002 8D8F02                STA $28F
130 C005 A9C0                  LDA #>BEEP
140 C007 8D9002                STA $290
150 C00A 60                    RTS
160 C00B 48        BEEP        PHA
170 C00C A90F                  LDA #15
180 C00E 8D18D4                STA 54296
190 C011 A909                  LDA #9
200 C013 8D05D4                STA 54277
210 C016 A951                  LDA #81
220 C018 8D01D4                STA 54273
230 C01B A961                  LDA #97
240 C01D 8D00D4                STA 54272
250 C020 A921                  LDA #33
260 C022 8D04D4                STA 54276
270 C025 A932                  LDA #50
280 C027 85FB                  STA 251
290 C029 85FC      LOOP1       STA 252
300 C02B C6FC      LOOP2       DEC 252
310 C02D D0FC                  BNE LOOP2
320 C02F C6FB                  DEC 251
330 C031 D0F6                  BNE LOOP1
340 C033 A900                  LDA #0
350 C035 8D04D4                STA 54276
360 C038 8D05D4                STA 54277
370 C03B 8D06D4                STA 54278
380 C03E 68                    PLA
390 C03F 4C48EB                JMP $EB48
```

*Fig. 10.3.* The assembly language program.

the routine marked BEEP to be run. The byte which is in the accumulator has to be pushed on to the stack, because it will be needed later, as it must be decoded by the routine at $EB48. The BEEP routine simply loads numbers into the various sound registers so as to create a sound of a pitch which is easily heard. Following the loading of the register addresses, the sound has to be maintained by means of a delay loop, which starts in line 270. The sound then has to be switched off, and this is done by placing a zero in each of three register addresses. Once this has been done, the correct content of the accumulator is pulled back from the stack, and the routine jumps to the correct decode address of $EB48.

The method of intercepting the key-decode is useful if you want to

perform any action each time a key is pressed. It's not so useful if you are looking for a specific key, because the codes which appear in the accumulator bear no simple relationship to the keys, and you will find that several keys give the same code. This is because the byte at this place is only one of a pair which decides the final keycode. Several other programs in this book make use of routines which can detect which key has been pressed, however.

# 11
# Disable RUN/STOP

This is another quicky which disables the action of the RUN/STOP key, using machine code. We'll be looking later, in the Copy Protection utility (No. 14), at methods of preventing people from looking at your listings (cheeky devils!) and this is a foretaste. Apart from copying, though, when you have written a BASIC masterpiece of a program, you have to protect it from beginners. This is particularly important for an educational program, where tiny tots may be using the keyboard. You, of course, know what to do if you press the RUN/STOP key by accident – you type CONT, then RETURN and the program goes on its way. A less experienced user might not have the advantage of this knowledge, and in any case, it gives an inexperienced user no confidence in a machine. How would you feel if you were a beginner, and the program kept stopping with a message just because your left hand little finger slipped. With this routine in place, your worries are over!

## Using it

Type the short BASIC program of Fig. 11.1, and make sure that there are 12 items of data. After careful checking, record the program, because it will delete itself after running, leaving only the machine code bytes in place. Pressing RESTORE along with RUN/STOP

```
10 PRINTCHR$(147):PRINT:PRINT"..PLEASE W
AIT FOR READY":A=49152
20 FORN=0TO11:READ D%:POKE A+N,D%:NEXT
30 SYS49152:NEW
1000 DATA169,11,141,40,3,169,192,141,41,
3,96,96
```

*Fig. 11.1.* Disabling RUN/STOP, in BASIC.

will put things back to normal, so that you can use the RUN/STOP key again.

## How it works

This is simplicity itself! Referring to Fig. 11.2, addresses 808 and 809 in the C 64 are a junction box for the RUN/STOP key routine. Every time this key is pressed, the machine will take an address from these locations and jump to that address, which normally is $F6ED. By changing the bytes at 808, 809, we force the machine to go to address $C00B when the key is pressed. This address, however, contains only a return from subroutine instruction, so that the STOP routine will not be used, and so the STOP key has no effect. The RTS at address $C00A is for the 'loader' section of the program. Pressing RESTORE with the RUN/STOP key has the effect of restoring all of these important addresses, which is why it has been mentioned so much in this book.

```
100 C000           *=$C000
110 C000 A90B                    LDA #<GO
120 C002 8D2803                  STA 808
130 C005 A9C0                    LDA #>GO
140 C007 8D2903                  STA 809
150 C00A 60                      RTS
160 C00B 60        GO            RTS
```

*Fig. 11.2.* The assembly language version.

# 12
# Pause in the Listing

Debugging a BASIC program is usually quite an easy task on most computers. You type LIST, press RETURN, and as the listing scrolls, you stop it with the SHIFT key so that you can look at it in screen-sized sections. As usual, though, the C 64 owner gets a raw deal, because there's no provision for interrupting a list and then continuing with it. You can always stop a listing by pressing the RUN/STOP key (if you haven't disabled it!) but you can't thereafter type CONT (RETURN) and see the listing continue. There is simply no provision in the machine for doing this.

As usual, then, we have to make the provision for ourselves. The program in this section will allow you to make use of the SHIFT and SHIFT-LOCK keys to control a listing. Using the SHIFT key, the listing will scroll until you press the SHIFT key, and will then stop. Scrolling will continue when you release the SHIFT key again. If you press the SHIFT-LOCK key while a listing is scrolling, it will remain steady on the screen until you press the SHIFT-LOCK key again. It's a very useful aid to programming, and one which you will use more and more when you have sampled it. Certainly if you have been used to being able to control screen scrolling on any other machine, you will want to have this utility on your C 64!

## Using the routine

Type the program in Fig. 12.1, and check carefully. In particular, make sure that you have 21 DATA items, and that each number is as shown in the listing. Now record the program at least once, and preferably twice. This is because the program deletes itself after running, leaving only the machine code in place. RUN the program, and then LIST. You will find now that pressing the SHIFT key controls the scrolling of the listing, but you will appreciate the action

```
10 PRINTCHR$(147):PRINTTAB(17)"PAUSE":PR
INT
20 PRINT"PLEASE WAIT....":A=49152
30 FOR N=0 TO 20:READ D%:POKE A+N,D%:NEX
T
40 SYS49152
50 NEW
1000 DATA169,11,141,6,3,169,192,141,7,3,
96,72,173,141,2,208,251,104,76
1010 DATA26,167
```

*Fig. 12.1.* Controlling scrolling, in BASIC.

more when you use it with a longer program. You can rewrite this program with higher line numbers (63000 for example) if you want to be able to merge it with other programs. You can also shift the machine code to other addresses, but only if you alter the address bytes in the program.

## How it works

As usual, the action of this program depends on the existence of an address which is stored as two bytes in the memory of the C 64. The bytes are at $306 and $307, and they normally contain an address $A71A which is the start of the routine which prints the keywords in a listing. If we intercept this routine at $306,$307, we can hold a loop going until the SHIFT key is released. This in turn is possible because the use of the SHIFT key causes a signal at address $28D. When this location is zero, the SHIFT key is not pressed, but if the SHIFT key is pressed, then the byte is *not* zero. Our routine, then, tests this location, and if the SHIFT key is pressed, simply repeats the test until the key has been released. The SHIFT-LOCK key uses the same address, so the action will also work with the SHIFT-LOCK key. It will also work with the CTRL and C= keys, as noted in the explanation which follows.

Figure 12.2 shows the assembly language coding. Lines 110 to 150 replace the address of $A71A in addresses $306, $307 with the address $C00B of the new routine, starting at PSE. This starts by pushing the byte in the accumulator on to the stack, and then loading the accumulator from location $28D. This tests the SHIFT or SHIFT-LOCK keys. It also, incidentally, tests the CTRL and C= keys. The shift key will make the byte in $28D equal to 1, the CTRL key will

```
100 C000           *=$C000
110 C000 A90B                  LDA #<PSE
120 C002 8D0603                STA $0306
130 C005 A9C0                  LDA #>PSE
140 C007 8D0703                STA $0307
150 C00A 60                    RTS
160 C00B 48        PSE         PHA
170 C00C AD8D02    LOOP        LDA $28D
180 C00F D0FB                  BNE LOOP
190 C011 68                    PLA
200 C012 4C1AA7                JMP $A71A
```

*Fig. 12.2.* The assembly language program.

make it 4, and the C= key will make it 2. Since the test in line 180 is for zero only, any of these keys will have the effect of interrupting the listing. The loop repeats until the key has been released. Note that you can't do anything else, like PRINT A, while the listing is halted in this way, because the microprocessor is in a closed loop in lines 170, 180. When the SHIFT (or other) key is released, the loop stops, the correct value in the accumulator is pulled from the stack, and the normal address of $A71A is used.

# 13
# Quick Colour Shift

We have spent a lot of time equipping the C64 with commands which exist on a lot of other machines. This time we're going to create a new command for the C64 exclusively – a colour shift command. You can get colours, of course, by making use of the C64 keys directly. In this program, however, the whole screen background colour is shifted each time you press the CTRL and Q keys together. This can be very useful if you are experimenting with pokes to the screen, because you often have to change the background colour in order to see the effect of the poke. In addition, the methods that are used in this program will be of considerable interest to the beginner machine code programmer who wants to flex a few muscles and try new ways of controlling actions on the C64.

## How to use it

The program relies entirely on machine code, and the BASIC loader in Fig. 13.1 simply pokes the bytes into place and deletes itself. Once it

```
10 PRINTCHR$(147):PRINT:PRINTTAB(13)"CTR
L-Q COLOUR":A=49152
20 FORN=0 TO 69:READ D%:POKE A+N,D%:NEXT
30 SYS A:NEW
1000 DATA120,169,17,141,20,3,169,192,141
,21,3,88,169,0,133,251,96,72,32
1010 DATA62,241,165,197,201,62,208,39,17
3,141,2,201,4,208,32,165,251,141
1020 DATA33,208,24,105,1,201,16,144,2,16
9,0,133,251,169,0,133,197,133,198
1030 DATA162,255,160,255,136,208,253,202
,208,248,104,76,49,234
```

*Fig. 13.1.* The BASIC loader version of the colour changer.

has been installed, pressing CTRL-Q will change the screen background colour, cycling through all the colours that the C64 permits if you keep pressing the keys. Type the program, and check it carefully. Make sure that you have 70 items of DATA altogether, and that the numbers are correct. Record the program at least once, preferably twice, before you attempt to run it, because it is self-deleting. Once the program has run, you can experiment with screen background changes simply by using the CTRL-Q keys.

## How it works

It's not quite so simple as it looks, as you can see from Fig. 13.2. This time, the pressing of the key is being detected by using the keyboard interrupt (IRQ) which we also used in the F-key program. In the course of these programs, I have used several different methods of inserting new routines into the key-action of the C64. This is deliberate, because it shows you examples of how these methods work so that you can use them for your own programs. In this case, the IRQ is suspended, using the SEI command, while the address in $314 and $315 is changed to point to $C011, COL. After restoring interrupts with CLI, a zero is loaded into address 251. This is the address which will be used to keep the colour number. This number can be changed in the range 0 to 15, and we will arrange the program so that when the number reaches 15, another press of CTRL-Q will set it back to 0. In the meantime, the set-up section of the program is complete, and we return to BASIC in line 170.

The colour-change routine starts in line 180. The byte in the accumulator, if any, is pushed on to the stack, and a subroutine $F13E is used to decode which key was pressed. The accumulator is then loaded from 197 ($C5), which contains a code for the last key pressed. This is compared with 62, which is the value that is given when the 'Q' key is used. If the value isn't 62, then the program goes to location EXIT, when the accumulator byte is restored, and the program jumps to the IRQ address of $EA31. If the 'Q' key was pressed, however, there is another test to carry out, for the CTRL key. This, as we have seen previously, makes use of location 653 ($28D), which contains the 'flag' byte for the CTRL, SHIFT and C= keys. We are looking for a value of 4 in this address, to indicate that the CTRL key has been pressed. If this value is not found, then once again, the program goes to the EXIT address.

The real action then starts in line 250. The colour number from

```
100 C000           *=$C000
105 C000 78                    SEI
110 C001 A911                  LDA #<COL
120 C003 8D1403                STA $314
130 C006 A9C0                  LDA #>COL
140 C008 8D1503                STA $315
145 C00B 58                    CLI
150 C00C A900                  LDA #0
160 C00E 85FB                  STA 251
170 C010 60                    RTS
180 C011 48        COL         PHA
185 C012 203EF1                JSR $F13E
190 C015 A5C5                  LDA 197
200 C017 C93E                  CMP #62
210 C019 D027                  BNE EXIT
220 C01B AD8D02                LDA 653
230 C01E C904                  CMP #4
240 C020 D020                  BNE EXIT
250 C022 A5FB                  LDA 251
260 C024 8D21D0                STA 53281
265 C027 18                    CLC
270 C028 6901                  ADC #1
272 C02A C910                  CMP #16
274 C02C 9002                  BCC NXT
276 C02E A900                  LDA #0
280 C030 85FB      NXT         STA 251
290 C032 A900                  LDA #0
300 C034 85C5                  STA 197
310 C036 85C6                  STA 198
311 C038 A2FF                  LDX #255
312 C03A A0FF      LOOP2       LDY #255
313 C03C 88        LOOP1       DEY
314 C03D D0FD                  BNE LOOP1
315 C03F CA                    DEX
316 C040 D0F8                  BNE LOOP2
320 C042 68        EXIT        PLA
330 C043 4C31EA                JMP $EA31
```

*Fig. 13.2.* How it works - the assembly language version.

address 251 is loaded into the accumulator, and then stored in address 53281 ($D021). This is the background colour address, so that the effect is to change the background colour on the screen. We then have to increment the number in 251. Since the number exists in the accumulator, we can do this by adding 1, and we can also compare the

number with 16. If the number now equals 16 we reset it to 0, but if it does not, and the carry is clear, then we simply store it back in 251 again, ready to use the next time. We then clear addresses 197 and 198, the last-key and buffer count address numbers. This ensures that the keyboard buffer has been cleared before anything else is done. Line 311 then starts a delay loop. As we saw before, this is essential because the IRQ action repeats 50 (60 in the USA) times per second. With no delay at this point, you would find that the colour code number incremented several times while you had the CTRL-Q keys pressed. The delay is the maximum that can be obtained by using two registers, and it's just about right, fortunately. At the end of the delay, the EXIT action restores the accumulator, and jumps to address $EA31 as normal.

If you can program in machine code, you are not stuck with just changing background colour. If the CTRL key is tested first, for example, you can test for various letter keys following it. You could use CTRL-Q for background, CTRL-A for border, and CTRL-Z for character colour (address 646 or $286). You can also use this key-detecting method to control anything else you fancy, like a sprite position, a sound, even a complete subroutine like un-NEW.

# 14
# Copy Protection

It's always a worry, when you have written a good program, that someone will manage to make a copy and sell cassettes. Pirating like this is the bane of software writers, and accounts for the very high prices of a lot of rather trivial games programs. After all, if you know that you will sell only a few thousand copies of a game, and there there will be ten pirate copies for each genuine one, you are tempted to put the price up to ten times what it would be if each user bought a copy. The other side of this problem, however, is backup. Cassettes are notoriously unreliable for data storage, and even disks can fail at times. If you are wise, then, you will want to have a backup copy of each useful program that you have. Commercial users very often have at least two backup copies. I personally will not buy any software that is copy-protected unless the price includes at least one backup copy, and the facility to buy extra copies at a lower price on exchange of a faulty copy.

Now this book is not intended for use by professional software writers, who know all of the tricks to make software difficult to copy. Nor is it intended for the pirate, who also knows all of the tricks for copying. You, the reader, are much more likely just to want to make your program difficult to copy by a casual user. The usual situation is that your program is on demonstration at the local Computer Club. You want to make sure that the Club's smart-Aleck doesn't make a copy while you are looking at something else. Full copy protection with the C64 is not really easy. The reason is that, though you can disable the STOP key and the STOP/RESTORE action, and you can prevent SAVE, LIST or LOAD from operating, it's not so easy to prevent anyone from simply loading the program in and then saving on another cassette, because your protection works only when the program starts to run. A lot of computers make it possible to save a program in such a way that it starts to run as soon as it has loaded. This is possible with the C64, but it needs a rather long

piece of machine code to carry it out. For the sort of limited protection that we are talking about, it's hardly necessary. Nor does that sort of protection work with the guy who takes a twin-cassette recorder around with him!

This protection sytem, then, is a 'budget' version which is reasonably simple to type in and use, but which offers protection against the type of copier who will try to load and save, or who might try to stop your program while it is running. The scheme is to shift the start of BASIC, and put a piece of machine code at the start of memory, followed by the program that you want to protect. This whole lot is then recorded as a single program. When this is loaded in again, any attempt to list shows only one line of nonsense, and if this program is saved on another tape, the pirate will still have only one line of nonsense to look at unless he can find out how to run the program. If the program is run, then the RUN/STOP, RESTORE, SAVE, LOAD commands are disabled to prevent copying. You must be absolutely sure that your program is 100% debugged, however, because the protection system will also zap the program if any error occurs during running.

## How to use it

Start with the machine completely reset, preferably by switching off

```
10 PRINTCHR$(147)
20 POKE44,9:POKE46,9:POKE48,9:POKE50,9
30 AD=2051:FORN=0TO11:READ D%
40 POKEAD+N,D%:NEXT
50 AD=49152:FORN=0TO19:READ D%
60 POKEAD+N,D%:NEXT
70 SYS49152
100 DATA169,16,133,43,169,8,133,44,96,0,
0,0
110 DATA32,3,8,169,18,133,45,133,47,133,
49
120 DATA169,8,133,46,133,48,133,50,96
READY.


RUN,LOAD,POKE 43,1,SAVE

LOAD, SYS 2051,THEN RUN
```

*Fig. 14.1.* The BASIC program which provides you with copy protection.

and then on again. Type in the program as shown in Fig. 14.1. This is a BASIC loader program which places machine code into two sets of locations. The loader occupies a piece of memory which is *not* at the normal start of BASIC, and it places some machine code at the normal start of BASIC address, and some at address 49152 ($C000) onwards. When you have typed the program, check it carefully – there are 32 items in the DATA lines altogether. Record the program at least once.

Now RUN the program. If all is well, you should see the READY prompt come up very quickly. Now LOAD in your own program, the one that you want to protect. You may want to edit this program now, though it's better to have done this earlier. Put in a new line at the start of your own program, as shown in Fig. 14.2. This pokes locations which will disable the RUN/STOP action, and send the computer to clear the memory when an error or other message (like PRESS RUN KEY OF RECORDER) is issued. Now carry out a POKE 43,1, which restores the normal start of BASIC. Record this program, using whatever filename you like. It's a good idea to use a filename which reminds you that this is a protected program.

When this program is LOADed, any attempt to LIST will show a line of nonsense. RUN will similarly produce rubbish. To make your program appear, you have to type SYS 2051 (RETURN) and then RUN (RETURN). I haven't tried SYS2051:RUN (RETURN), and it's most likely that this would work as well. As your program runs, any attempt to stop it will be ignored, and any attempt to SAVE or LOAD will clear the computer. It won't baffle the expert – but experts are seldom interested in ripping off other people's programs anyway.

```
1 POKE 808,66:POKE 768,226:POKE 769,252
```

*Fig. 14.2.* The line which must be the first line of your own program.

## How it works

The assembly language of Fig. 14.3 shows how the program works. When you have the machine code in place, the BASIC loader uses SYS49152 to force the machine code to start from address $C000, line 190 of the listing. This causes a jump to a subroutine at $803, address 2051. At this address, the instructions load a new 'start-of-BASIC' address into locations $2B and $2C. This new starting address is $0810, 2064 in decimal terms. When this new address has been put into

place, the routine returns to line 200. From then on, the other pointers are changed so as to allow a BASIC program to be loaded in starting at 2064. When all of the pointers have been changed, the program ends. This section is not used again.

```
  100 0803          *=$0803
  110 0803 A910                   LDA #<STRT
  120 0805 852B                   STA $2B
  130 0807 A908                   LDA #>STRT
  140 0809 852C                   STA $2C
  150 080B 60                     RTS
  160 080C 000000                 BYT 0,0,0,
0
  170 0810          STRT           =  *
  180 C000          *=$C000
  190 C000 200308                 JSR $0803
  200 C003 A912                   LDA #<STRT
+2
  210 C005 852D                   STA $2D
  220 C007 852F                   STA $2F
  230 C009 8531                   STA $31
  240 C00B A908                   LDA #>STRT
  250 C00D 852E                   STA $2E
  260 C00F 8530                   STA $30
  270 C011 8532                   STA $32
  280 C013 60                     RTS
```

*Fig. 14.3.* The protection explained in assembly language.

When you LOAD your BASIC program, then, it will start at address 2064. When you POKE 43,1, the normal start of BASIC address at 2049 is restored. Recording will now cause the whole mixture of machine code (addresses 2051 to 2063) and BASIC (your own program) to be recorded. The machine code at 49152 is *not* recorded, because it is not needed again. When this mixture is then loaded back into a C64, LIST or RUN will work from the usual address of 2049. At this address, what is present is a set of machine code bytes, and the machine will intepret this as being a peculiar BASIC program. Because of the three zero bytes at the end of this section, the machine will take it that this program ends at address 2062, and that there is nothing else present. If you use SYS 2051, however, the machine code at 2051 is run. This will alter the start-of-BASIC address to the correct value so that you can LIST or RUN your program. Once the program starts to RUN, the POKE

statements in the first line will protect it. To make a copy, a pirate has to know that SYS2051 must be used before LIST or RUN. If you leave the program running, and you don't let anyone see the commands you have typed to run it, it is reasonably secure.

It can, of course, be made still more secure if you want to spend more effort on it. For example, the machine code could end with a RUN command, so that SYS2051 would not only shift the start of BASIC but also cause the BASIC to start running. This can be done (I haven't tried it myself) by clearing the accumulator, and then excecuting, in order, JSR $A65E (the CLR command), then JSR $A68E (sets pointers to the start of the program) and then JSR $A7AE (execute next statement command).

# Part Two
# Subroutines

The utilities that comprise the first part of this book are all aimed at enhancing, in various ways, the usefulness of the C64 as a programmable machine. They all provide you, in one way or another, with ways of making it easier to write and assemble programs.

In this section, the emphasis is on *subroutines* which would normally be incorporated into programs rather than used as tools for creating programs. In the course of programming, a lot of actions seem to be needed over and over again. Some of these actions are not particularly easy to obtain, especially if you are not accustomed to writing BASIC for the C64. At the start of this book, I mentioned how programs could be constructed from a library of 'building block' routines, which could then be MERGEd with any program core that you wrote. What follows now is a selection of some of the subroutines which I have found useful, and some which I have found to cause a lot of difficulty for programmers. Please feel free to try them out and use them as you wish. In some cases, I have included pieces of program which will test the subroutines for you, because some of them are by no means easy to test. Happy programming!

# 15
# Variable Lister

Unless you plan your BASIC programs very carefully, you are most likely to forget just what variable names you have used. Of course, if you stick to programs of a few lines, using just N, X and T$, this isn't likely to perplex you. Sooner or later, though, you'll design a blockbuster of a program which makes full use of a lot of variables – and unless you have kept a sheet of paper with each variable name listed, you'll lose track of them. This is where this program comes in, because it will examine your own program and print all the variable names that it finds. It's really more of a utility than a subroutine.

The basis of the action is that each variable name, and each variable value is held in a list which is called the 'variable list table', or VLT. Space is allocated for this list as you type a program. When a program is run, the values and names for the variables are entered into this list. A Rolls-Royce version of such a variable-listing program would not only tell you what variable names had been used, but what values were allocated, and in which lines the variables appeared. This is not such a program – it's more a Mini Metro version which simply lists the variable names.

## How to use it

Type the subroutine of Fig. 15.1 which uses line numbers from 60000 onwards, and check it carefully. Save it on cassette, and mark the place on the cassette so that you can find it easily. When you are typing in a long program and you want to check how you have used variables, merge in this subroutine. Now use GOTO60000, and you will see your variable names listed, with the correct type signs (% or $) where appropriate. Keep a note of this listing, which is in the order that the variables have been used. This order gives you a clue as to where in the program each variable has first been mentioned. It's simple, but it's useful!

```
60000 PRINTCHR$(147):PRINT:PRINTTAB(15)"
VARIFINDER":PRINT
60010 VS=PEEK(45)+256*PEEK(46):VF=PEEK(4
7)+256*PEEK(48)
60020 TP$="":KK=0:LL=0:IF VS=VF-14 THENP
RINT:PRINT"END OF LIST":END
60030 IF PEEK(VS+1)>=128THENTP$="$":LL=1
28
60040 IF PEEK(VS)>=128THENTP$="%":KK=128
60050 PRINTCHR$(PEEK(VS)-KK);CHR$(PEEK(V
S+1)-LL);TP$;
60060 PRINTSPC(5);:VS=VS+7:GOTO60020
```

*Fig. 15.1.* The variables lister. Append it and run it to get a list of your variable names.

## How it works

The variable finder works because of the way that the C64 organises its variables. Each time a new variable is allocated, an entry is made in a 'variable list table' (VLT). This VLT is kept in the memory which immediately follows the BASIC program, and it has to be shifted each time you add to or delete anything in the BASIC program. The VLT is not deleted when you NEW a BASIC program. In fact, if you then load in a shorter program, you can make use of variable values that were used in the program that you have deleted! This facility is called 'overlay', and it's one which is oddly neglected by amateur programmers, probably because Commodore make no mention of it in their manual.

Getting back to this subroutine, each variable entry in the VLT uses just seven bytes. The first two of these are used for the name of the variable, and so these are the two that this program will make use of. The remaining five bytes are used to code the value. If the variable is a floating-point number, all five bytes are used to code the value. For an integer number, only two bytes are used, and for a string, just three bytes. Strings are actually stored elsewhere, and the three bytes in the VLT are the string length (one byte) and the address of the first byte of the string (two bytes in the VLT). The first two bytes of the VLT entry, for the variable name, are coded so as to show the variable type. If the variable is a floating-point number (N,Z,BB etc.), then the first two bytes are the ASCII codes for the name. The second byte will be zero if the name uses just one letter. If the variable is a string, then 128 is

added to the ASCII code number for the second letter of the name. If the name consists of only one letter, then the second byte is simply 128. For integer variables, 128 is added to both bytes.

The program starts by finding the start address and end address for the VLT. The start address is stored in locations 45 and 46, and this is obtained and allocated to variable VS. The end address is located in 47 and 48, and this address is allocated to variable VF. Line 60020 then zeros a variable called TP$, which will be used to store each variable name as we read it, and two number variables KK and LL. These variables will not appear in the listing because they were declared after the end-of-VLT address was found. The list is printed only until VS=VF−14. This will prevent the two variables VS and VF themselves from appearing in the listing. Since variables are listed in order of appearance, and this subroutine has been tacked on to the end of a program, these entries should be at the end of the VLT, unless the same names have been used in your own program.

Lines 60030 and 60040 then check for the variable type. If the second character has an ASCII code of 128 or more, then the variable type must be a string, and the variable name of TP$ is set accordingly. If the variable type is an integer, then the first character will have an ASCII code of 128 or more, and TP$ is set to '%'. Line 60050 then prints the variable name and its type, using LL and KK to make the necessary corrections to the code number before using CHR$ to print the name. Line 60060 then prints five spaces, and VS=VS+7 shifts the address in the VLT to the start of the next variable value. GOTO60020 then zeros the variables TP$, KK and LL again, tests for the end of the program, and goes through the procedure with the next variable.

Extending this program so as to list lines in which variables are used is not too difficult, but the process is slow. Each time a variable is found, its complete name can be allocated to another variable. The length of this can be found using LEN, and BASIC starting from address 2049, can be searched for a group of codes of this description. If the BASIC line numbers are held in memory each time, the line number can be printed each time the variable name is found. See the 'linefinder' program for inspiration.

# 16
# The INSTR Routine

Several computers have a BASIC command called INSTR or INSTR$, which is used to find one string buried inside another one. For example, if a string XY$="PERSEVERE", and AB$="SEVER", then you can type something like:

```
PRINT AB$ INSTR XY$
```

and get the result of 4, which is the number of places along in the word PERSEVERE that you will find the first letter of SEVER. For a lot of data processing programs, this can be very useful, and it can even be a great help in quite different applications. For example, suppose that at some stage in a program you wanted a reply of YES or NO. With an old-fashioned BASIC you might test for just Y or N, but you can make much more interesting tests with an INSTR command. Suppose, for example that you made YS$="YESYUPSUREOKYEAHRIGHT", and used whatever answer was made as RP$. Then with RP$ INSTR YS$, you could find if the reply was any one of these ways of saying 'YES'. If the reply is any one of these, the result of INSTR will be a number which is not zero. You could therefore test this by having a line:

```
IF RP$ INSTR YS$ <>0 THEN . . .
```

and this would save the need to have a whole set of tests like:

```
IF RP$="Y"
IF RP$="YES"
IF RP$="YUP"
```

and so on. The INSTR routine can save a lot of typing in a program, and the more experienced you are in programming, the greater the number of uses you are likely to find for it.

The C64, of course, doesn't have such a command. We could add it with machine code, but it's a lot easier just to use a subroutine. This

requires you to make use of set variable names, one for the long string and another one for the shorter string, and it allocates a variable X as the number which is returned. This number, remember, is the position number in the long string at which you will find the start of the short string. If this number is zero, the short string is not contained in the long one.

## Using the subroutine

Type the subroutine in Fig. 16.1, check it, and save it on tape, noting the readings of the tape counter. In your main program, the main,

```
60000 LM=LEN(MN$):LI=LEN(IN$):IF LI>LM T
HENPRINT"IMPOSSIBLE":GOTO60040
60010 FORX=1TOLM:IF IN$=MID$(MN$,X,LI)TH
EN 60030
60020 NEXT:PRINT:PRINT"NOT FOUND":GOTO60
040
60030 PRINT:PRINT"STARTS AT ";X
60040 RETURN
```

*Fig. 16.1.* The INSTR subroutine.

longer, string should be allocated to variable MN$, and the shorter string to IN$. You then call the routine by placing GOSUB60000 in your program. As it stands, the subroutine will print a phrase such as:

STARTS AT 5

but you can omit the printing in lines 60020 and 60030 (put REM in place of each PRINT) so that a value of X is returned without any comments. In this case, X will *not* be zero when the string is not found, and to change this, you need to alter line 60020 to read:

60020 NEXT:X=0:GOTO60040

to get a return of 0 to indicate 'not found'. This amended version is more likely to be useful if you are using INSTR in a YESYUPOK ... type of application. Like any subroutine, you should be prepared to carve this one around so that it does whatever you want of it.

## How it works

Line 60000 finds the lengths of the two strings. If the 'short' string is, in

fact, the longer of the two, the subroutine cannot run, so this is tested, and an error message is issued. If all is well, then line 60010 is executed. This runs through a loop, with X taking values from 1 to the length of the long string. For each value of X, a set of characters of the same size as the short string is sampled, using MID$ to extract the characters from the long string. If this set of characters is identical to the short string, then the program skips to 60030, which prints the value of X. If the loop finishes without the short string being found, the message in line 60020 is printed, and the subroutine returns. Note that X is not made equal to zero when no match is found unless the program is modified as shown above.

# 17
# PRINT AT Made Possible

The C64 can make use of the TAB modifier to a PRINT statement, and the number which follows TAB can be anything in the range of 0 to 255. Numbers greater than 39 will force printing on to lower lines of the screen, but only six lines can be jumped in this way. A lot of machines offer a PRINT AT or PRINT@ command which allows you to place printing anywhere on the screen that you choose. This is done by allocating a number following the command. This makes it possible to print words or graphics anywhere on the screen, and not necessarily in the order of top to bottom or left to right.

One particularly valuable feature of a PRINT AT command is that it makes it possible to have a set of instructions at the top of the screen, and a line used for your input which is wiped and reused at each input, without disturbing the rest of the screen. These effects can be obtained with the C64, but only with a lot of programming effort.

This subroutine allows the C64 to have a simple type of PRINT AT action. It demands that you allocate line and column numbers to two variables, then follow with a normal PRINT statement. The subroutine will move the cursor to the correct part of the screen, and the printing will then take place starting at the cursor position.

## How to use it

As usual, type the subroutine (Fig. 17.1) and check it. You can

```
60000 KK=1024+LL*40:AH=INT(KK/256):AL=KK
-256*AH
60010 POKE214,LL:POKE209,AL:POKE210,AH
60020 PRINTSPC(CL);
60030 RETURN
```

*Fig. 17.1.* A subroutine for a PRINT AT action.

renumber the lines to suit your own needs. Save the subroutine on a piece of tape, noting the tape counter positions for future use. When you want to make use of the subroutine in a program, merge it with the program, and then modify your program so that it sets up the routine. To do this you have to allocate numbers to LL and CL. LL is the line number, in the range 0 to 23, and CL is the column number, in the range 0 to 39. The combination of these two will get you to any position on the C64 screen. Typically, you might use a line like:

```
LL=3:CL=12:GOSUB60000:PRINT"TRAPPED!"
```

with the variables allocated, the subroutine called, and then the PRINT statement following.

### How it works

The key to this one is how to move the cursor. The cursor position is controlled by the numbers that are contained in several addresses in the memory. Control of vertical position is comparatively easy, but horizontal position is not so simple, which is why SPC has been used instead. Line 60000 forms a screen address number for the start of the line, using the line number LL. This address number then has to be split into high and low bytes, AH and AL respectively, to be used in POKE commands. The address 214 is then poked with the line number directly, and addresses 209, 210 with the low and high bytes of the screen address respectively. These pokes get the cursor to the start of the correct line. Line 60020 then prints a number of spaces corresponding to the value of CL so that the cursor is spaced across the line. Since a semicolon follows the PRINTSPC command, any later printing will be at the position of the cursor.

That's all! Remember that this can be used for graphics characters as well as for items like titles. You can even carry out simple animation with this routine as it stands. Like every other routine in this book, it's up to you to make what you want from it.

# 18
# The Flashterisk

When you want an input, and you use INPUT in your program, the C64 prints a prompt in the form of a question mark. For a lot of purposes, like menu choices or Y/N answers, however, it's neater and quicker to use a GET A$ type of reply. This means that you only have to press the key, ignoring RETURN, and it can make life easier in a program that makes use of a lot of choices. Unfortunately, no computer provides for a question mark to be displayed when GET A$ (or its equivalent) is used. This subroutine supplies just such a prompt for your own GET A$ routines in the form of a flashing asterisk.

The asterisk is the prompt, and the fact that it flashes tends to make you take more care with your reply. A program which uses this subroutine looks a lot more professional, and is much easier to use. A message in the instructions for the program can, if needed, say that the flashterisk will be used when a single-key reply is needed, and a question mark will be displayed when the answer needs several keys to be pressed, followed by RETURN.

## Using the subroutine

This is a very short routine, which is easy to type (Fig. 18.1), but watch for the semicolons. If you omit any of these, the effects will be very odd! Make sure that you save the routine on a piece of tape, and note the tape count for the start and finish. The routine uses high value line numbers so that you can merge it with your own programs, and you

```
60000 GET K$:IF K$<>""THEN RETURN
60010 PRINT"*";:FORKK=1TO500:NEXT
60020 PRINTCHR$(20);:FORKK=1TO500:NEXT:G
OTO60000
```

*Fig. 18.1.* The flashterisk subroutine.

can renumber these lines as you wish. At any part of the program where a one-key choice is needed, print your message, such as:

```
PRESS Y OR N KEY
```

and follow this with GOSUB60000. When this runs, the message will appear, and the flashing asterisk on the next line will indicate that the machine is waiting for the reply. The reply will be allocated to variable K$ when the subroutine returns.

## How it works

It's the essence of simplicity, really. Line 60000 uses a GET K$ step, and if this results in something which is not a blank string, meaning that a key was pressed, then the subroutine returns. If no key has been pressed, however, line 60010 prints an asterisk, and then a time delay is used to ensure that the asterisk remains on screen long enough to see. Line 60020 then backspaces and deletes the asterisk, by using PRINTCHR$(20). The semicolons hold this printing in the same line, and the delete action is followed by another time delay. These time delays determine the flashing rate, and you can experiment with the effect of different numbers, and also with unequal numbers, in these places. Line 60020 then returns to line 60000 so that the key can be tested again. Who could believe that something so simple could look so effective? Remember, too, that you aren't confined to flashing asterisks. You can flash graphics characters just as easily with this routine, and that should open up a lot of interesting possibilities for you.

# 19
# Flashing Your Titles

Several computers have a FLASH command. Typing FLASH1 or FLASHON will cause anything that is printed thereafter to flash as it is displayed on the screen, and FLASH0 or FLASHOFF will mark the end of the flashing text. These commands are, of course, carried out by means of machine code, and doing this for the C64 is a long job, though by no means all that difficult. These commands, in addition, will flash the text wherever it happens to be on the screen, and will continue to flash the text even when it scrolls. For a lot of purposes, though, you don't need to flash any more than a title, and you don't need to have it flashing for all the time that the machine is switched on, or when the screen scrolls. This simple subroutine will flash a title or other piece of text for a specified number of times. As often as not, this is all that you'll need to draw attention to the title.

It doesn't stop at titles, of course. You may want to flash a question which is printed just before an input, or as a warning. For example, if you are entering your name, and the entry is supposed to be surname first, you could have the word SURNAME flashing a few times just before entry. As a warning, you might, in a database program, have a menu choice which would wipe information from a disk. A sensible precaution would be to flash the words ARE YOU SURE? several times before accepting a Y or N answer. As long as you don't overdo it, a flashing facility is decidedly useful.

## How to use it

Type the subroutine (shown in Fig. 19.1) which uses the usual high line numbers. You can, of course, use lower or higher line numbers as you wish. You might, for example, want to use several of these subroutines in one program, so it would make sense to use different line numbers between 60000 and 63999 for each of them. Be particularly careful

```
60000 LL=LEN(T$):BP$="":FOR X=1TOLL:BP$=
BP$+CHR$(20):NEXT
60010 FOR R=1TORP:PRINTTAB(CL);T$;:FORJ=
1TO500:NEXT
60020 PRINTBP$;:FORJ=1TO500:NEXT:NEXT:RE
TURN
```

*Fig. 19.1.* Flashing a title.

about semicolons, because they are all essential to the operation of the program. Remember to save the subroutine on tape before you attempt to use it, and make a note of the tape counter readings so that you can locate the subroutine again when you want to use it.

To make use of the subroutine, you have to assign values to several variables. The title or other phrases which you want to flash will have to be assigned to T$. If it is already assigned to some other variable name, like A$, then simply include the instruction T$=A$ before the subroutine is called. You will also need to assign values to CL and RP. CL carries the TAB number for printing the title, and RP is the number of times that the title will flash before the subroutine returns. Having assigned these variables, use GOSUB60000 (or whatever line number you have used) to call the routine, and watch your title flash!

## How it works

The flashing is done by printing the title, waiting, then printing at the same position a set of spaces which are the same size as the title. This is followed by another pause, and then the process repeats for as many times as are programmed by the value of number variable RP. Line 60000 of the subroutine finds the number of characters in the title as variable LL. The rest of the line then packs a variable BP$ with backspace characters, CHR$(20). This will make a string which, when printed following the title, will backspace over the title and delete it. This string BP$ can be printed like any other string, and it will blot out the title when it is used. In line 60010, then, the title string is printed at the TAB number which has been passed to it as variable CL. A semicolon following the PRINT statement holds the printing cursor in the line, and a time delay is created. Altering this time delay will alter the flashing interval. In line 60020, the backspace string BP$ is printed. This has the effect of wiping out the title, and once again, the use of a semicolon keeps the printing cursor in the same line. Another

time delay is now used before the next repetition. The loop repeats for a number of times that is decided by variable RP.

This is a BASIC subroutine, and the important difference between this and a machine code routine is that the BASIC program cannot continue while the title is flashing. A machine code routine which made use of the IRQ interrupt every fiftieth of a second would allow flashing to continue even while other parts of the program were running. Such a program, however, is rather too long for this book. If your tastes run to machine code programming, however, you might like to tackle it. There are enough clues about using the interrupts in the Utilities section to allow you to draw up your plans!

# 20
# Fielded Inputs

If you get a chance to watch a really professional program, perhaps a commercial accounts program running on a machine like the IBM or Advance, one thing that will probably strike you is how the program deals with inputs. Instead of having just the usual "TYPE YOUR NAME" followed by a question mark, these programs indicate just where your name is to be placed, and also show what the limits of size are. This might be done by means of divisions in a box, or a set of chequer squares, each of which is to be replaced by a letter as you type. This sort of thing is called a 'fielded input'. You might, for example, want a date to be entered, so you show something like:

DD/MM/YY

on the screen, and the cursor is over the first 'D'. You are expected to type something like 01/05/84 in response to this, and the format on the screen is a reminder that only this form of reply will be accepted. A fielded input therefore fulfils two functions. It shows the form which a reply is expected to take, and it allows the machine to reject any reply that is not of the correct form. Any program which might be used by non-programmers must be made as clear as possible, so that no user has to wonder just what is required at any input stage. The fielded input is also used when filing on disk is being carried out. If each entry is of a known length (number of characters), then we can use what is termed 'random access' filing, which allows us to find any entry without having to read all of the entries into the computer. For more information on this sort of thing, see my book: *Commodore 64 Disk Systems and Printers.*

You don't need to own an expensive business style computer to program fielded inputs into your programs, however. This subroutine is just one example of what can be done using the modest BASIC of the C64. In this case, the field consists simply of a set of chequer squares, and each letter that you type replaces a chequer. When all of the

chequers have been replaced with letters or spaces, you are allowed to decide whether your reply is correct or if it needs to be changed in any way. This is important, because if the user has filled in the pattern incorrectly, it must be relatively easy to have another go.

## Using the routine

Type the subroutine (Fig. 20.1) taking care over the blank strings and the semicolons. When you have finished checking, record the routine, and note the tape counter numbers so that you can reload the routine

```
60000 PRINTCHR$(147):PRINT:PRINT:PRINT:T
B=INT((39-NR)/2):F$="":B$="":RP$=""
60010 FORJ=1TONR:F$=F$+CHR$(166):B$=B$+C
HR$(157):NEXT
60020 PRINTTAB(TB)F$;B$;:FORJ=1TONR
60030 GET K$:IF K$=""THEN 60030
60040 PRINTK$;:RP$=RP$+K$:NEXT J
60050 PRINT:PRINT"PRESS RETURN IF THIS I
S CORRECT-":PRINT"ELSE SPACEBAR."
60060 GET K$:IF K$=""THEN 60060
60070 IF K$=CHR$(32)THEN 60000
60080 RETURN
```

*Fig. 20.1.* A subroutine for fielded inputs.

when you want it. You can, of course, use whatever line numbers you want – you aren't stuck with 60000 to 60080 as used here. In your main program which calls this routine, you have to provide a quantity for one of the variables. Variable NR is used for the number of squares or chequers that will be printed. This must be a number which will be large enough to allow for the maximum number of letters that the user is likely to type in reply. For a surname, 15 is often acceptable for this number. Having assigned NR, use GOSUB60000 (or whatever line number you have used). The chequer pattern will then appear centred on the screen. As you type letters, chequers will be replaced by the letters until the last chequer disappears. When this happens, you will get the message about pressing RETURN if the entry is correct, spacebar if not. The choice of the spacebar is not ideal if the name that has been typed has been followed by spaces, and you might like to change this. Line 60060 waits for a key to be pressed, and line 60070 checks if this was the spacebar. If you use CHR$(29) in this line in

place of CHR$(32), then the rejection of the entry can be done by using the horizontal cursor shift key rather than the spacebar. For a lot of applications this might be more satisfactory – but don't forget to alter the message in line 60050 as well!

# 21
# Walking the Title

Flashing is one method of drawing attention to a title, but there are others. An obvious one is to have your title in another colour, but a 'walking title' attracts even more attention. A walking title is one which appears at one side of the screen and moves across to the other side, or to wherever you want the title to rest. Like flashing, it's a device that you should use sparingly, because it takes time, and the user would become bored if every title in a long program walked across the screen. More important than the actual walking title is the method that is used. Obviously, this can be used with any string of characters, so it's also a method of making any shape move across the screen. This subroutine, then, can be very useful to you in a lot of different programs.

## How to use it

Type the subroutine in Fig. 21.1. You can, of course, use different line numbers if you want to assemble a lot of subroutines together. As usual, you have to be careful about the semicolons, which are essential to the program. To use the subroutine, you have to allocate the variable name of T$ to the title or other string that you want to walk over the screen. Having done this, use GOSUB60000 (or whatever number you have used) to run the subroutine. This version walks the string from right to left, since this is the direction that is normally used for moving

```
60000 ZZ=LEN(T$):Y=1:FORNN=1TO38
60010 PRINTTAB(38-NN);MID$(T$,1,Y);" ";
60020 FORJJ=1TO80:NEXT
60030 IF Y<ZZTHENY=Y+1
60040 PRINTCHR$(145):NEXT:RETURN
```

*Fig. 21.1.* How to make your title walk.

messages. You can reverse the direction by rewriting the subroutine lines that carry out the movement, as detailed below. You can also alter the speed of walking by altering the number in the time delay of line 60020.

## How it works

In line 60000, the variable name of ZZ is allocated to the length of the title, variable Y is set to 1, and a loop is started. This loops covers a range of numbers less than the number of columns on the screen, and altering the numbers here will alter the starting and finishing positions of the title. In the first line of the loop, 60010, the TAB setting is 38−NN, which for the first pass will be 37. This is close to the right-hand margin. What is printed is MID$(T$,1,Y), and on the first pass this is the first letter of the title. It is followed by a space which on later passes will delete any letter to the right. The existence of this space is the reason why a higher TAB number cannot be used – it would cause a new line to be selected. Line 60020 then causes a time delay, which controls the walking speed, and line 60030 increments Y, but only if Y is less than the number of characters in the title. Line 60040 then prints CHR$(145) to keep the printing in the same line on the next pass through the loop.

On the second pass through the loop, the TAB number will be 36, and the first two characters of the title will be printed. While the value of Y is being incremented, the title will appear as if it were being pulled on a banner from the right-hand side of the screen. When Y has reached its maximum value of ZZ, line 60030 is ignored, and each title is printed one place to the left, with the space at the end of line 60010 wiping out the last letter of the previous printing position. The delay number of 80 has been selected quite carefully. A lower number causes the title to move rather too quickly to recognise, a higher number makes the movement look jerky. By all means experiment with these numbers, however, to get the effect that you want.

You can modify the routine to send the title from left to right. The TAB numbers in line 60010 will now be NN rather than 38−NN. You also want to display the *last* character first. To do this, you have to set Y equal to ZZ in line 60000, and have line 60030 decrement Y until it reaches 1. The MID$ also has to be altered. The line now has to read:

```
PRINTTAB(NN);"";MID$(T$,Y,ZZ−Y+1);
```

and the loop condition will have to be FOR NN=1 TO 39–ZZ. Line 60030 will now read:

```
IF Y>1 THEN Y=Y-1
```

These changes will give left to right walking. I will leave it for you to decide what you have to do to get the title to walk up or down – it's easier!

# 22
# Sort It – Fast!

Somewhere in every data-processing program, you will find the need for a sort routine. 'Sort' in this sense means putting into order, and the usual order is low-to-high for numbers and alphabetical for strings. If you have a list of names and addresses, for example, you may want to present the list in alphabetical order of surnames, or in alphabetical order of town names, depending on your requirements. A subroutine which will sort strings into alphabetical order is therefore very useful. It can be just as easily applied to numbers by substituting the variable names in the subroutine, using number variables rather than string variables.

One problem of string sorting routines, however, is that many of them are slow. This doesn't matter if you are working with only 20 to 50 names, because all sort routines take more or less similar times for so few items. The crunch comes when you have several hundred names to sort, and they are all in random order. The difference between the fastest sort routine and the slowest could then be enough for you to go and get yourself a cup of coffee. The fastest sort routines make use of machine code rather than BASIC, but the machine code versions require a lot of typing, and the speed isn't usually so useful unless you have a list of many thousands of names.

This, then, is a string sort in BASIC. It's the type of sort that is called 'Shell-Metzner' after its originators, and it's acknowledged as being the fastest type of sort for this application that is easily programmable into BASIC. So that you can try it for yourself, I have included a few lines of program (Fig. 22.1) for testing the routine. This program creates a lot of nonsense words at random. The creating process *is* slow, and to judge the speed of the sort, you need to put a STOP into a line 55. When the 'random words' have been created, the screen message will indicate the STOP. You can then type CONT, get your stop-watch ready, and press RETURN as you

```
10 DIM N$(200):MX=200:FORX=1TOMX
20 L=INT(RND(1)*5+2)
30 FOR P=1 TO L:CD=64+INT(RND(1)*26+1)
40 N$(X)=N$(X)+CHR$(CD)
50 NEXT P:NEXT X
60 GOSUB 60000
100 FOR X=1 TO MX:PRINTN$(X);" ";:NEXT
200 STOP
```

*Fig. 22.1.* A program for testing the fast string sort.

start the stop-watch. Stop again whenever you see the screen printing start, and this will give you the time to sort 200 completely random words.

## Using the routine

Type this routine (Fig. 22.2) carefully, because it's not easy to tell where a mistake might be. If you find that a list of words hasn't been sorted, or that the program stays in a loop, you will have to examine each part of the subroutine carefully. Remember to record it on tape before you attempt to use it. If you don't have a program which can read a list of words from tape or disk, then use the random-word generator in Fig. 22.1. The result after sorting should be a stream of nonsense words in strictly alphabetical order.

```
60000 NT=2
60005 NT=2*NT
60010 IF NT<MX/2 THEN 60005
60020 FOR J1=1 TO MX-NT
60030 FOR J2=J1 TO 1 STEP -NT
60040 IF N$(J2)>N$(J2+NT)THEN 60080
60050 NEXT J1:NT=INT(NT/2)
60060 IF NT>0 THEN 60020
60070 RETURN
60080 T$=N$(J2):N$(J2)=N$(J2+NT):N$(J2+N
T)=T$
60090 NEXT J2:GOTO60050
```

*Fig. 22.2.* The string sorting subroutine.

## How it works

All sort routines work by comparing one string with another, and swapping them round if they are in the wrong order. The simpler types of sorts compare strings which are adjacent, like the 4th with the 5th, 5th with 6th and so on. When a sort like this (a 'bubble sort') is used on a long list, it takes a lot of time because the machine has to keep repeating the comparisons. For example, if the 4th item was swapped with the 5th and the 5th with the 6th, you may then find that another swap of the (new) 4th and (new) 5th is needed. The computer can cope with this only by running down the list over and over again until a run produces no swaps. This can mean a very large number of loops for a long list, and this is what makes the bubble sort so very slow.

All of the faster sorts rely on the principle of selecting items for comparison that are *not* adjacent. The differences between these quicker sorts are mainly in the way that is used to select the items. The program that we have used here starts by calculating a 'separation' number in lines 60000 to 60010. If MX is 200, then the process of doubling will arrive at the number of 128 for NT when the subroutine starts. The main loop then starts in line 60020, and will be from 1 to 200−128 (=72) with the test figures. This is a definite improvement on using the whole set! The inner loop uses J1 TO 1 STEP −128, and since J1 will never be more than 72 in this example, it can run only once this time round.

The comparison in line 60040 is between item J2 and J2+NT. This makes the first comparison between items 1 and 129, when J2=J1=1. If the items are out of order, then they are swapped in line 60080, and the NEXT J2 is selected – which in this case won't cause another loop because of the STEP figure of −128. The program then returns to line 60050 for NEXT J1, so that item 2 is compared with item 130, and the loop continues in this way until the whole of the J1 set is complete.

At this point, the last part of line 60050 is carried out. In our example, this makes a new value of NT equal to 64, and the loops are carried out again. Because of the smaller value of NT, there is a chance now that the inner loop using J2 will run more than once. As the program moves on, the interval number NT becomes smaller until the final run will sort out any items that are next to each other but in the wrong order. After this stage, NT is reduced to zero, and the program ends.

A description like this is necessarily brief, and the best way of seeing how the program works is to construct an example on paper,

using perhaps 20 items, and going through the motions of the program until you see what is happening at each pass of the loop. Apart from anything else, it'll give you a lot of respect for the people who designed the routine in the first place!

# 23
# Search for It!

Once again, this is for database designers, because database work is what computers are mainly used for if you exclude games. When you have a long list of items in the memory of the computer, it's useful to be able to call up an item when you want it by typing one name or perhaps even part of a name. The simplest way of doing this is to go down the items one by one, comparing each item with what has been typed, and stopping when the correct item has been found. If the list is in alphabetical order, however, a finding program of this type is quite unnecessary. All that is needed is to use the list like an index. You look up the middle item, and decide if it's the one you want. If it is not, then you decide whether your item is in the first half or the second half. You throw away the half you don't need, and repeat the process. In this way, an item can be found after only a few comparisons, often only 8 to 11 steps, rather than by a search through the whole list. If your list is of several thousands names, you'll be glad! A search of this type is often called a 'binary' search, because the size of the list that is being searched is halved after each unsuccessful search.

## How to use it

Type the subroutine of Fig. 23.1, using different line numbers if you want to. Save your subroutine on tape after checking it carefully. You need to have the number of items on the list allocated to variable name MX. The subroutine is set up for use with a complete name, and I have assumed that your list would store other information as well. You could also have 'linked lists'. For example, one list, N$, might contain names and another one, AD$, addresses. By using the searching routine to get the name, you also get a value of N2, the number of the name on the list. This number can then be used to get

```
61000 INPUT"PLEASE TYPE NAME...";A$
61010 J=MX+1:N1=J
61020 N2=INT((N1+1)/2)
61030 IF N2=0 OR N2>MX THEN 61090
61040 IF A$<N$(N2) THEN 61080
61050 IF A$=N$(N2) THEN 61100
61060 N1=N2+J:N3=N2
61070 GOTO 61020
61080 N1=N2+N3:GOTO 61020
61090 PRINTA$;" IS NOT IN THE LIST"
61100 RETURN
```

*Fig. 23.1.* The searching subroutine, for use with a sorted list.

AD$(N2), which would be the address. The routine will tell you if a name cannot be found.

You can alter the routine so as to look for a name that starts with a given set of letters, A$. You would then have to add LL=LEN(A$) to get the length of this set of letters, and the comparison steps like line 61040 would then have to read:

IF A$<LEFT$(N$(N2),LL) THEN 61080

with a similar change to line 61050. You could also use the INSTR routine in conjunction with this search operation. Remember, however that this search operation can work only on lists that *have already been sorted into alphabetical order*. Any attempt to use the routine on an unsorted list is doomed to failure. Because of this, you will very probably want to use this subroutine together with the sorting subroutine (No. 22) and the line numbers have therefore been made different.

## How it works

Line 61000 obtains the name that you want to look for in the list. Line 61010 finds a number, J, which is one more than the number of items in the list. This number of items should have been assigned to variable MX before the subroutine was called. Variable N1 is then set to equal J. In line 61020, a new variable N2 is defined as the nearest whole number less than half of N1. Line 61030 tests this number to find if the list has been searched without finding the item.

Line 61040 then compares your request A$ with N$(N2), the halfway item. If A$<N$(N2) then A$ is in the first half of the list, not the second, and control passes to line 61080. This creates a new value of

N1, equal in this instance to N2, because N3 has not yet been used. If your list had 100 items, for example, and you found that your choice was less than the 50th item, then line 61080 would make N1=50. The GOTO61020 then makes the program search the first 50 items rather than the whole list, by looking at the 25th item and making the same sort of comparison.

If the comparison in line 61040 fails (A$ not less than the midway item), then another test is needed in line 61050. This is for equality. If this test is not made, then the item can *never* be identified! If the items are not identical, then A$ must belong to the second half of the list. Line 61060 makes N1 equal to N2+J, and N3 equal to N2. For a list of 100 items, for example, with N2=50, this would make N1 equal to 151 and N3=50. When the program moves back to line 61020, the new value of N2 will be 75, so we are searching the second half of the list. Line 61080 will then make use of the value of N3 if the item is in the first half of this piece of the list.

Once again, if you want to see in detail how this program works, then you will have to go over it on paper, making up a list and seeing what steps are needed to find an item in any part of the list. If you have never used a binary search routine like this before, you will be amazed at how quickly a name can be found in a sorted list.

# 24
# It's the Greatest!

Yes, this is yet another routine which is applied to lists, but this time it's for lists of numbers. You very often have a list of numbers stored as an array, and you want to find the maximum size of number. You may, for example, want to draw a graph or a bar-chart, and you must find the maximum size of number that you will need to display. Having found this you can make your scales so that this number will be represented by the top-most graph point or the longest bar. First of all, though, you need to find the largest number. It's a simple subroutine, but a surprising number of programmers in BASIC seem to find the task difficult.

## How to use it

Type the routine (Fig. 24.1) as usual, with whatever line numbers you want to use. Save it on tape for future use – it's a very simple and straightforward routine, but once is enough for typing any program! Your list of numbers should be in an array called A, and with the number of items equal to TP. When these variables have been allocated, then calling GOSUB60000 (or whatever number you have used for the start of the subroutine) will search for the maximum number. The list does not need to be in order, of course, and the maximum size is returned to your main program as variable MX.

```
60000 MX=0:FORJ=1 TO TP
60010 IF A(J)>MX THEN MX=A(J)
60020 NEXT:RETURN
```

*Fig. 24.1.* Finding the maximum in a list.

Line 60000 starts with MX set to zero. The loop then takes limits of 1 to TP, the whole number of items in the array. In line 60010, the value of each item is compared with MX, and the two are exchanged if the array item is greater than MX. For the first pass through the loop, the exchange will always take place unless the array item is also zero. In this way, MX will be equal to the largest number in the array when the loop is complete in line 60020. You can, of course, adapt this to find the *smallest* item in an array. You can do this by setting MX equal to the first item, and using the < sign in place of the > sign in line 60010.

# 25
# Point to It!

Have you ever thought that your menus were getting boring. I don't mean the meat and two veg. type of menu, but the sort of list of items that always ends with the instruction:

TYPE NUMBER TO SELECT

Such a menu needs to be followed by a test to make sure that the number which you selected was within the correct limits, and it gets rather boring if you have to make use of it over and over again. It can also take quite a lot of time if you have to type a number, then press RETURN, for each choice. This subroutine makes use of quite a different type of menu. The items are displayed and numbered, certainly, but the numbers are not needed for the choice and you can omit them if you want. The difference between this and the ordinary menu is that this one displays an arrow next to the first item. You can move this arrow up and down the menu items by using the vertical cursor key (with or without SHIFT). When the arrow points to the item that you want to use, you simply press the spacebar – and your item is selected! There's no need to test numbers, because the subroutine arranges things so that the arrow will always point to an item, and the correct number for that item will always be selected. Magic!

## How to use it

To start with, you have to do some planning, and you will have to read the 'how it works' section as well. Refer to Fig. 25.1 for the BASIC routine. You must choose how many items of menu you will have. The example shows 5, but you could have up to about 20 – the limit is set by the number of lines that you can print on the screen without scrolling. Having chosen your number of items, you must

```
60000 PRINTCHR$(147):PRINT:PRINT
60010 AD=1104:N=1:CL=55376
60020 PRINTTAB(2)"1. FIRST."
60030 PRINTTAB(2)"2. SECOND."
60040 PRINTTAB(2)"3. THIRD."
60050 PRINTTAB(2)"4. FOURTH."
60060 PRINTTAB(2)"5. FIFTH."
60070 REM USE ZZ ITEMS HERE
60200 PRINT:PRINT
60210 POKEAD+40*N,62:POKECL+40*N,7
60220 GET K$:IF K$=""THEN60220
60230 V=ASC(K$):POKEAD+40*N,32
60240 IF V=17THEN N=N+1
60250 IF V=145THEN N=N-1
60260 IF N=ZZ+1THEN N=1
60270 IF N=0 THEN N=ZZ
60280 IF V<>32 THEN 60210
60290 RETURN
```

*Fig. 25.1.* A visual menu which needs no numbers to be typed, nor any mugtraps.

allocate this number to the variable ZZ, and you will have to write your own menu lines between line number 60020 and 60200. The lines need not be numbered, but the subroutine will allocate a number to variable N. This will be 1 if you have selected the first line, 2 for the second and so on. Your main routine must therefore contain lines which will select subroutines depending on the value that N will have after this subroutine returns. The usual sort of thing is ON N GOSUB 1000,2000,3000... and so on. Now type your version of the subroutine with your own menu, and save it on tape. Check it carefully, because it relies on POKE commands, and any of these can cause trouble if you get numbers wrong. Call the subroutine with GOSUB60000 (or whatever number you have used in your own version). You will see your menu items printed, with the arrow against the first one. Move the arrow with the vertical cursor key until it points to the correct menu item, press the spacebar, and watch your program work!

## How it works

This subroutine depends on poking into the screen memory. Because of that, you can't alter it without some knowledge of what is involved.

The explanation here will be centred around the example, with five items, but for your own routine, you will have to amend numbers accordingly. The example routine would be called after ZZ was set to 5.

Line 60000 clears the screen and prints two blank lines. If you want to print just one blank line, or none, then you will also have to alter the AD number in line 60010. This AD number represents a position on the screen which will be to the left of the first line of printed menu. This corresponds to N=1, so this value is set, and the variable CL is set to 55376. This is the corresponding position in the colour screen map, which must be poked at the same time as the screen position memory if the arrow is to be visible. Lines 60020 to 60060 then print the items of the menu, which have been numbered in this example so that you can see how the number N corresponds to them. Line 60200 then prints two more blank lines which you might want to omit, or use for a reminder (MOVE ARROW, PRESS SPACEBAR WHEN READY).

The real action starts in line 60210. The pokes in this line place an arrow-head at the left-hand side of the first menu line. Two pokes are needed, one for the position memory and one for the colour memory. The POKE statement uses a formula which includes N, and each value of N will select a different line for the arrow. Line 60220 gets a key in the usual way, and the ASCII code for this key is found in line 60230. At the same time, the arrow-head is also deleted by poking the space character, 32, into the screen memory address that was selected earlier. Lines 60240 to 60250 then test the key, which should be the vertical cursor key, with or without SHIFT, or the spacebar. If the down-cursor has been pressed, then N is incremented; if the up-cursor has been pressed, then N is decremented. We can't allow N to be less than 1 or more than 5, though, so lines 60260, 60270 attend to this. The effect of these lines is to cause a 'wrap-round'. If the arrow-head is on the last line and you attempt to move it down, it will shift to the first line. If it is on the first line and you try to move it up, it will move to the last line. This makes it unnecessary to have any 'mugtrap' tests in the way that you do for an ordinary menu.

Line 60280 then checks for the spacebar having been pressed. If it was not, then control passes back to line 60210, which will print the arrow at whatever position is represented by the value of N. If, on the other hand, the spacebar was pressed, the subroutine returns to the calling program with the value of N representing the line of the menu that was chosen. You can test the subroutine with the line:

```
10 ZZ=5:GOSUB60000:PRINTN
```

# 26
# Sprite Editor

If you have tried to generate sprites on the C 64, you'll know how insufferably boring the whole job is. You have to draw your sprite shape on a 24 × 21 grid by filling in squares, then work out a set of numbers for each line. You end up with a total of 63 numbers, each of which has to be copied correctly into DATA lines to be read by your sprite-creating program. After creating one sprite, you usually don't feel like creating another, and so yet another good idea goes out the window. What would be decidedly useful would be a program which allowed you to get to these DATA lines a lot more easily.

This is such a program. Like the utility which created DATA lines for machine code bytes, this program will generate DATA lines for you. In this case, however, the DATA lines are the DATA for a sprite, a shape which you have drawn by using the keyboard controls, watching the result on the screen. Because the normal size of sprite is too small to work with, the screen image for this program is of a mammoth sprite which just about covers the whole screen. You can alter the pattern as you choose, and only when you decide that it's right will the DATA lines be created. You can then save the DATA lines for later renumbering and incorporation into your own sprite programs.

## How to use it

Unlike the short routines that have been featured in this book so far, this is a medium-length program, Fig. 26.1. If you feel that it's too much to type in one go, then by all means type half, and then save it on tape to continue at another sitting. The important thing is to make sure that each line is correct before you press RETURN on it. In a program of this size, looking for errors is not easy, particularly when you didn't design the program in the first place. Note, however, that

# Part Three
# Programs

The last two items in this book are not short utilities nor subroutines, but full-blown programs. They do, however, serve a subsidiary role in allowing you to make more creative programs of your own, rather than being useful in their own right. One of the main weaknesses of the small BASIC of the C 64 is that it allows no commands for the sprite graphics nor for sound creation. All the work of sprite graphics and sound has to be done by means of POKE instructions. This is very unsatisfactory, because one POKE looks very much like another when you read a program, and it's always a long and complicated task to set up the pokes. These two programs are designed to take a little of the weight from your shoulders when you work with sprites or sound. The first program relieves you of having to write out and type the sixty-three numbers that are needed to specify the shape of a sprite. The second program enables you to hear the effect of changing the POKE numbers for a sound, and will print out the addresses and POKE numbers that created the sound. Neither of these is fully comprehensive – the sprite program generates only the shapes, and does not attempt to provide BASIC words to control them. The sound program deals only with musical notes, one channel at a time, with no attempt to control noise, pulses or sound effects. It's only by limiting the aims like this that the program can be made short enough to be a reasonable prospect for typing!

the listing which is shown in this book is an *exact* copy of the printout of the tested working program, so if you type with 100% accuracy, you will get exactly the program that I designed and tested.

```
100 PRINTCHR$(147):PRINTTAB(13)"SPRITEBU
ILDER":PRINT
110 PRINT"USE CSR KEYS TO MOVE, SPACEBAR
 TO PLACE":PRINT"DOT, DEL TO REMOVE."
120 PRINT"PRESS RETURN WHEN READY."
130 FORJ=1TO2000:NEXT:PRINTCHR$(147):PRI
NT:FORJ=9TO30:LL$=LL$+CHR$(163):NEXT
140 PRINTTAB(8)CHR$(111);LL$;TAB(31)CHR$
(112):FORJ=9TO30:UL$=UL$+CHR$(164):NEXT
150 FORN=2TO20:PRINTTAB(8)CHR$(165);TAB(
31)CHR$(167):NEXT
160 PRINTTAB(8)CHR$(108);UL$;TAB(31)CHR$
(186)
170 AD=1112:LN=0:POKE53281,1
190 POKEAD+LN,32
200 GET A$
210 IF A$=CHR$(17)THEN AD=AD+40
220 IF A$=CHR$(29)THEN LN=LN+1
230 IF A$=CHR$(145)THEN AD=AD-40
240 IF A$=CHR$(157)THEN LN=LN-1
250 IF AD<1112 THEN AD=1912
260 IF AD>1919 THEN AD=1112
270 IF LN<0 THEN LN=23
280 IF LN>23 THEN LN=0
290 IF A$=CHR$(13)THEN 350
300 IF A$=CHR$(20)THEN LN=LN+1:GOTO200
310 IF A$=CHR$(32)THEN POKE AD+LN,102:LN
=LN+1:GOTO450
320 FOR J=1TO50:NEXT
330 POKE AD+LN,102:FORJ=1TO100:NEXT
340 GOTO190
350 AD=1112:LN=0:DIM N(63):FORJ=1TO63:N(
J)=0:NEXT:J=0
355 POKE214,23:POKE209,152:POKE210,7:PRI
NT"PLEASE WAIT"
360 FOR NN=0TO20:FORQ=0TO2:J=J+1
370 RESTORE:FOR LN=0 TO 7:READ ML
380 IF PEEK(AD+8*Q+LN)=102 THEN N(J)=N(J
)+ML
390 NEXT LN:NEXT Q
400 AD=AD+40:NEXT NN
410 GOTO600
```

```
450 IF LN>23THEN LN=23:AD=AD+40
460 IF LN<0THEN LN=0:AD=AD+40
470 GOTO200
500 DATA 128,64,32,16,8,4,2,1
600 A1%=PEEK(43)+256*PEEK(44)
610 T%=PEEK(A1%+2)+256*PEEK(A1%+3)
620 IF T%<>1000 THEN A1%=PEEK(A1%)+256*P
EEK(A1%+1):GOTO610
630 Q%=A1%+4
640 Z%=0:FOR J=1TO63
650 L$=STR$(N(J)):LL=LEN(L$)
660 FOR X=2TOLL:Y%=ASC(MID$(L$,X,1)):Z%=
Z%+1
670 POKE Q%+Z%,Y%:NEXT X:Z%=Z%+1:POKEQ%+
Z%,44
680 IF Z%<62 THEN NEXT J
690 IF J>=63 THEN 730
700 POKE Q%+Z%,32:FOR D=1TO3:IF PEEK(Q%+
Z%+D)=38THEN POKE Q%+Z%+D,32
710 NEXT D
720 Q%=Q%+71:Z%=0:NEXT J
730 PRINTCHR$(147):PRINT:PRINT"YOUR SPRI
TE DATA NUMBERS ARE NOW IN "
740 PRINT"DATA LINES 1000 TO 1030. DELET
E ANY
750 PRINT"REMAINING & SIGNS FOR YOURSELF
."
760 POKE53281,2:END
1000 DATA&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1010 DATA&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1020 DATA&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
1030 DATA&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
```

*Fig. 26.1.* The sprite editor program. This creates DATA lines for you to use.

When the program runs, you will see an outline on the screen. This represents the shape of a giant sprite, and the pieces around the edge are in a different colour so as to make it easier to recognise the edge. This does not affect the final colour of the sprite, which you control when you make use of the DATA. There is a block cursor flashing, and this can be moved by means of the cursor arrow keys at the bottom right-hand side of the C 64 keyboard. Moving the block to

the left will delete any block in the way, and the DEL key can also be used to do this. If you want a block to remain in a position, you press the spacebar. In this way, you can move a block to any part of the sprite outline and press the spacebar where you want a part blocked in. The blocks make up your sprite shape, and if you step back a few paces, you will get a better idea of what it will look like in its final form. Since you can delete blocks as well as put them in place, the whole shape can be edited as you please. You don't have to accept the shape until it's just as you want it. When it is just right, pressing the RETURN key will fix the shape, and create the DATA lines 1000 to 1030. Depending on what shape you have chosen, you may find that only two or three lines are needed, and you can delete the others. By deleting the rest of the program, you can save the DATA lines by themselves. You will also have to delete unwanted & signs from any incomplete DATA line, and any comma at the end of the line. It's a good idea to keep a sketch in a separate notebook of each shape that you create, along with notes of the tape position indicator readings for the DATA lines. In this way, you can build up a library of useful sprites and MERGE them in with any sprite program whenever you want. You must remember, though that *you need to load in the sprite generator program afresh for each sprite you want to create*. This is because the DATA lines are altered when the sprite is created, and fresh new DATA lines are needed for each attempt.

## How it works

Lines 100 to 120 print a brief reminder of how the program is used. This is followed by a time delay to give you time to read the instructions, and then the screen is cleared again in line 130. Following this, the top border of the sprite-rectangle is created as string LL$. Line 140 then prints the left-hand top edge, the top border and the right-hand top edge of the rectangle. UL$ is the bottom edge, and line 150 prints the sides, followed by line 160 which prints the bottom edge and corners.

Having created the rectangle which represents the sprite outline, the starting address of the top left-hand corner of this screen position is then allocated to variable AD, the line position number LN is set to zero, and the background colour is set to colour 1. Line 190 then pokes a blank space into the first address, the top left-hand corner of the rectangle, and line 200 is a GET to find which key, if any, is pressed. Lines 210 to 310 then deal with the effect of whatever key

happens to be pressed at this time. These lines form a large loop which extends from line 190 to line 340.

Lines 210 to 240 deal with the effect of the cursor keys. A vertical cursor key will alter the AD number by 40, plus or minus. A horizontal cursor key will change LN by one. The address at which the block cursor will appear will be AD+LN, so that AD has been used to select the vertical position and LN to select horizontal position. The next four lines are concerned with making sure that the cursor does not stray out of the boundaries of the rectangle. If the AD number becomes too low, the address is changed to 1912, which shifts the cursor from the top of the rectangle to the bottom. If the AD number indicates that the cursor would fall under the bottom line of the rectangle, the AD number is changed to 1112 to shift the cursor to the top. Similarly, the LN number is not allowed to exceed 23 or become less than 0.

Line 290 tests for the RETURN key. If this has been pressed, the program jumps out of this loop to the section that will read the blocks and create the DATA lines. Line 300 tests for the action of the DEL key, and simply moves the cursor to the next position, leaving no remaining block at the original position. Line 310 then tests for the spacebar. This causes the block to be left permanently in place, and it also shifts the position of the cursor one place to the right. The GOTO450 following this adds a piece of program which tests LN. If LN has reached an edge, then AD is changed rather then LN. By doing this, when a block is fixed at one edge, the cursor moves down rather than across to the other edge. Assuming that you create your sprite as you create your programs, from the top down, this makes it less likely that you will blank out a block at the left-hand side of your shape when you paint one in on the right-hand side. Line 320 then introduces a short time delay, and line 330 prints a block in place, followed by another time delay. The loop ends in line 340 with the GOTO190 command. The net effect if you press no keys is that you see a flashing block at some position on the sprite rectangle. When the spacebar is pressed, the flashing block moves to the next position, and a steady block remains behind.

The next part of the program deals with the conversion of the shape on the screen into DATA lines for a sprite. This uses many of the routines which were described earlier in section No. 3, and I won't go over them in detail again. Line 350 sets the starting address AD back to 1112 and LN to zero. A number array N(J) is also dimensioned and cleared. Line 355 then positions the print cursor down and prints the PLEASE WAIT message. The sprite shape is

then read into the array, using the loop which starts in line 360. This must take 21 lines, of three sections per line, and make up a number to represent eight block positions in each section. In each section, then, the decimal position number is read, and is added to the number N(J) if a block occupies the position given by PEEK(AD+8*Q+LN). When eight blocks have been read in this way, the next section is read, followed by the next line until all 63 numbers have been filled into the array. Where a blank exists, the number in the array will be 0, otherwise, it will be a number which represents the sprite shape. At the end of this routine in line 410, then, the array contains all the DATA numbers, and these then have to be converted into ASCII codes to be poked into DATA lines.

From line 600 on, the routine is virtually the same as that of the machine code to DATA program of Fig. 3.1. Each number in the array is read, converted to a string, and then poked as ASCII codes into a DATA line, followed by a comma. When one line is nearly full, the ampersands (&) at the end are erased, and a new DATA line is taken. On average, all four DATA lines will not be needed unless the sprite is unusually big. When the creation of the DATA lines is complete, the program prints a brief message about the DATA lines, and ends, leaving you with your sprite data. You can then erase the rest of the program, and record just these DATA lines, ready to use in your next sprite masterpiece!

# 27
# Music Editor Program

Producing music on the C 64 is, by any standard, very hard work compared to so many other computers. To start with, there are no BASIC sound commands, so each sound action must be programmed by poking numbers into memory. Another infuriating point is that you cannot PEEK the same memory addresses to find what has just been poked there, because the machine clears these addresses almost as fast as you can poke them. In common with all other computer sound systems, too, it's very hard to relate what you hear to what you program.

One way out of the problem is a sound editor. The sound system of the C 64 is a very good one, and remarkable effects are possible, if you can just find out how. It's so complicated in fact, that a program which could help you with every possible type of sound would be very long, certainly longer than you would want to type from a listing. This program is therefore a compromise – it sets up the sound signals for one channel at a time, using two types of waveform only, and ignoring the sound effects that can be obtained from more advanced use of the sound pokes. That's why I have called it a music editor rather than a sound editor. It allows you to put in figures for a musical note, and change your entries as you please, hearing the effect on the sound all the time. When you have got your note just as you want it, you can signal this to the program, and then see the poke numbers and addresses printed out on the screen. For music purposes, you would probably want to use this to set up sounds for up to three 'instruments'. You would want to experiment with waveforms, attack, decay, sustain and release numbers for each channel until you got the effect you wanted. For the music, then, you would note the pokes for the waveforms, and ADSR settings, and simply alter the pitch numbers and delay times for each note. You can also, obviously, use this as the skeleton for a much larger sound editor which will let you hear the effects of many more of the C 64 sound POKE commands.

```
10 PRINTCHR$(147):PRINTTAB(14)"MUSIC EDI
TOR":PRINT
20 PRINTTAB(1)"THIS PROGRAM ALLOWS YOU T
O TEST THE"
30 PRINT"EFFECT OF POKING NUMBERS INTO T
HE SOUND"
40 PRINT"REGISTERS. AS YOU ENTER THE NUM
BERS YOU"
50 PRINT"CAN HEAR THE SOUND CHANGE, AND
YOU CAN"
60 PRINT"EDIT IT TO SUIT WHAT YOU NEED.
PRESS
70 PRINT"PRESS THE SPACEBAR WHEN YOU ARE
 "
80 PRINT"SATISFIED WITH THE SOUND, AND T
HE POKE"
90 PRINT"VALUES WILL BE LISTED FOR YOU."
110 PRINT"THE PROGRAM STARTS WITH A MIDD
LE-C NOTE."
130 PRINT:PRINTTAB(8)"PRESS RETURN TO PR
OCEED":INPUT K$
140 BA=54272:FORN=BATOBA+24:POKEN,0:NEXT
1000 PRINTCHR$(147):PRINT:PRINTTAB(14)"S
IMPLE SOUND":PRINT
1010 PRINT"WHICH CHANNEL, PLEASE (1,2,3)
 ";:INPUT CH
1020 IF CH<1 OR CH>3 THENPRINT"MISTAKE":
GOTO1010
1030 VB=BA+7*(CH-1)
1040 HI=16:LO=195:WF=33:AD=2:SR=246:VL=1
5:DL=1000:GOSUB10100:GOSUB30000
1050 GOSUB10100
1060 MN=1:MX=255:F=0
1070 PRINT:PRINT"TRY DIFFERENT NOTE(COAR
SE) ";MN;" - ";MX:GOSUB10000
1080 GOSUB10010
1090 IF F=0 THEN HI=RR:GOTO1050
1100 F=0:MN=1:MX=255
1110 GOSUB10100
1120 PRINT:PRINT"TRY DIFFERENT NOTE(FINE
) ";MN;" - ";MX:GOSUB10000
1130 GOSUB10010
1140 IF F=0THENLO=RR:GOTO1110
1150 F=0:MN=0:MX=15
1160 GOSUB10100
```

```
1170 PRINT:PRINT"TRY DIFFERENT ATTACK ";
MN;" - ";MX:D=2:GOSUB10000
1180 GOSUB10010
1190 IF F=0THENA=RR*16:AD=A+D:GOTO1160
1200 F=0:MN=0:MX=15
1210 GOSUB10100
1220 PRINT:PRINT"TRY DIFFERENT DECAY ";M
N;" - ";MX:GOSUB10000
1230 GOSUB10010
1240 IF F=0 THEND=RR:AD=A+D:GOTO1210
1250 F=0:MN=0:MX=15
1260 GOSUB10100
1270 PRINT:PRINT"TRY DIFFERENT SUSTAIN "
;MN;" - ";MX:GOSUB10000
1280 GOSUB10010
1290 IF F=0 THENS=16*RR:SR=S+6:GOTO1260
1300 F=0:MN=0:MX=15
1310 GOSUB10100
1320 PRINT:PRINT"TRY DIFFERENT RELEASE "
;MN;" - ";MX:GOSUB10000
1330 GOSUB10010
1340 IF F=0 THENR=RR:SR=S+R:GOTO1310
1350 F=0:MN=0:MX=10000
1360 GOSUB10100
1370 PRINT:PRINT"TRY DIFFERENT DURATION
";MN;" - ";MX:GOSUB10000
1380 GOSUB10010
1390 IF F=0THEN DL=RR:GOTO1360
1400 F=0:MN=0:MX=15
1410 GOSUB 10100
1420 PRINT:PRINT"TRY DIFFERENT VOLUME ";
MN;" - ";MX:GOSUB10000
1430 GOSUB10010
1440 IF F=0THEN VL=RR:GOTO1410
1450 F=0:MN=0:MX=262:VT=VL
1460 GOSUB10100
1470 PRINT:PRINT"TRY FILTER VALUE ";MN;"
 - ";MX:GOSUB10000
1480 GOSUB10010
1490 IF F=0 THEN GOSUB 20000
1500 PRINT:PRINT"NOW FILTER TYPE- "
1510 PRINT"TYPE H,B,L OR HL ":GOSUB 1000
0:PRINT"N FOR NO FILTER"
1520 INPUT FT$ :VL=VT:POKEBA+23,7
1530 IF FT$="H"THEN VL=VL+128:GOTO1590
1540 IF FT$="L"THEN VL=VL+32:GOTO1590
```

```
1550 IF FT$="B"THEN VL=VL+64:GOTO1590
1560 IF FT$="HL"THEN VL=VL+160:GOTO1590
1570 IF FT$="X"THEN 1600
1580 IF FT$="N" THEN POKE BA+23,0:GOTO16
00
1590 GOTO1460
1600 PRINTCHR$(147):PRINT:PRINT"NOW YOU
PROBABLY WANT TO TRY AGAIN."
1610 PRINT"PRESS Y TO EDIT THIS SOUND AG
AIN"
1620 PRINT"PRESS N IF YOU WANT TO SEE TH
E POKE":PRINT"NUMBERS DISPLAYED."
1630 GET A$:IF A$=""THEN 1630
1640 IF A$="Y"THEN 1050
1650 IF A$="N"THEN 1670
1660 PRINT"INCORRECT-Y OR N ONLY":GOTO16
30
1670 PRINTCHR$(147):PRINTTAB(11)"CHANNEL
 ";CH;" SOUND."
1680 PRINT:PRINTTAB(2)"POKE"VB","LO
1690 PRINTTAB(2)"POKE"VB+1","HI
1700 PRINTTAB(2)"POKE"VB+4","WF
1710 PRINTTAB(2)"POKE"VB+5","AD
1720 PRINTTAB(2)"POKE"VB+6","SR
1730 PRINTTAB(2)"POKE"BA+21","FL
1740 PRINTTAB(2)"POKE"BA+22","FH
1750 IF FT$<>"N"THENPRINTTAB(2)"POKE"BA+
23","7
1760 PRINTTAB(2)"POKE"BA+24","VL
1770 PRINTTAB(2)"DELAY COUNT IS"DL
1780 FORN=BA TO BA+24:POKEN,0:NEXT
1790 END
10000 PRINTTAB(3)"(ENTER X TO GET NEXT C
HOICE)":RETURN
10010 INPUT R$:IF R$="X" THEN F=1:GOTO10
030
10020 RR=VAL(R$):IF RR<MN OR RR>MX THEN
PRINT"MISTAKE":GOTO10010
10030 RETURN
10100 POKEVB,LO:POKEVB+1,HI:POKEVB+5,AD
10110 POKEVB+6,SR:POKEVB+4,WF:POKEBA+24,
VL
10120 FOR D=1TODL:NEXT
10130 TR=WF AND 254
```

```
10140 POKE BA+4,TR:POKEBA+11,TR:POKEBA+1
8,TR
10150 RETURN
20000 FH=INT(RR/256):FL=RR-256*FH
20010 POKE BA+21,FL:POKEBA+22,FH
20020 RETURN
30000 PRINT:PRINT"NOW CHOOSE WAVEFORM"
30010 PRINT"FOR MUSIC, USE T OR S ONLY":
GOSUB10000
30020 INPUT"YOUR CHOICE - ";R$
30030 IF R$="T"THEN WF=17:GOTO30080
30040 IF R$="S"THEN WF=33:GOTO30080
30050 IF R$="X"THEN 30070
30060 PRINT"INCORRECT- T OR S ONLY":GOTO
30020
30070 RETURN
30080 GOSUB10100:GOTO30000
```

*Fig. 27.1.* The music note editor. You can listen to the note as you make changes in the sound control numbers. The POKEs will be listed when you choose to do so.

### How to use it

Unless your typing is particularly fast and accurate, don't try to enter all of this program (Fig. 27.1) in one session. Your concentration starts to wander after about an hour of concentrated typing at one- or two-finger rates, and when that happens, all sorts of mistakes are liable to creep in. It's better to catch the mistakes as you make them than to have to hunt through a program of this length afterwards looking for errors. As you type the program, save each section on tape, so that your tape contains a progressively longer chunk. Do this until the program is complete, and then make another recording just for safety. Mark the tape or note the tape counter settings for the start and end of the program.

When the program runs, you will be presented with a title and brief instructions. The principle is that the program starts a sound playing for you. It's a Middle C, with settings of registers which you will probably want to change. The screen then shows you what variable is being changed, and allows you to make the change. As each change is made, you will hear the effect on the sound. Each change is an INPUT step, so that you have to press RETURN to enter the quantity. To start the program, you have to press RETURN. You are then asked to specify which channel you want to use, 1,2, or 3. When

you have made this choice, the sound will then be heard – remember to turn up the volume control of your TV receiver. If you are using a monitor, you will have to make use of the sound output socket, connected to a separate amplifier and speaker.

The next choice that you are offered is of waveform. There are only two waveforms allowed, triangle (T) or sawtooth (S). The pulse waveform has been omitted, because choosing it would require another pair of pokes to be made (the pulse width pokes). Your reply must therefore be T, S or X. X is the choice that you make throughout this program if you want to keep things as they are, making no change. When you make a choice, you will hear the effect, and you get a chance to make another change to the *same* item. You can therefore experiment with what each sound command does, and leave it only when you are satisfied. The program also gives you the opportunity to go over all the commands again when you have run through all of the possible changes.

The next item that is presented, when you have typed X to the waveform change query, is of pitch. Now the C 64 uses two pitch numbers, a coarse pitch and a fine pitch. Altering the coarse pitch number makes much more difference to the note than altering the fine pitch number. If you want musically correct pitch notes, the C 64 manual gives a table of coarse and fine numbers to use. This program also gives you this choice, but with the usual advantage that you can hear the effect of your change. As usual, you get as many shots at changing the number as you like, and you have to press the 'X' key to get to the next choice. This is of the 'fine' pitch number, and your selection here will complete your choice of pitch for the note.

The next set of choices are more interesting, because they are the ones whose effect is least easy to predict. You are, in turn, asked to enter attack, decay, sustain and release figures. The range of numbers that you are allowed to use is shown on the screen, and as always, you can hear the effect. Don't be deceived into thinking that some changes have no effect. The trouble with sound commands is that a lot of them are interactive, so you don't hear the full effect until all of the quantities have been entered. The effect of the sustain number, for example, depends a lot on the time of note that is chosen. This is why you get a chance to go over all of the quantities again after the set has been completed. In each of these entry steps, you are allowed as many changes as you like, until you press the 'X' key to proceed to the next quantity.

When the attack, decay, sustain and release figures have been entered, the next quantity is duration. Now this is a difficult one to

fix. If you make the duration of a note very long, you will often mask the effect of ADSR, because the sustain and release effects can be heard only if the delay is comparatively short, and the note is properly terminated. You will find, for example, that with a large sustain setting, the note continues for quite a long time after you might expect the delay loop to have ended it. This is true only if the correct poke is made at the end of the delay, and there's more about that in the 'How it works' section. You are asked to choose your volume level after the delay number has been fixed.

The next things to settle are the filter settings. The C 64 sound system allows you to simulate the effect of filtering the sound waveform, so that only a range of wave frequencies will get to the loudspeaker. This can sometimes have noticeable effects on the sound - it may even make it inaudible! The C 64 manual does not mention the pokes for filtering, and they are not the simplest to carry out. In this program, you can choose whether to have a filter or not. You must first choose a filter value number, however. Having done this, if you don't want to use filtering, type N in reply to the menu question. if you *do* decide to try out the effect, you have to specify high-pass (H), low-pass (L), band-pass (B) or high- and low-pass (HL). Try the effect of the different filters on your note, because the program encourages you to experiment. You do *not*, however, hear the effect of picking the filter number until you have picked the filter type. These filter quantities recycle, as usual, for as long as you want until you enter an 'X' or an 'N'.

When you have finished with all of the sound pokes, you are offered the chance to go round again. There may be several adjustments to make, and it's useful to be able to hear what these do when all of the other numbers have been poked. If you don't want to go round again, you can answer 'N' to the question, and the screen will clear. You will then see what addresses have been poked, and with what quantities. You can then note these quantities for later use.

## How it works

It may look massive, but the program is in fact not all that difficult to understand, because so much is done by subroutines. The preliminaries take up lines 10 to 130, following which all the sound addresses are poked with zero to clear out any old effects. The next title follows in line 1000, and you then get to choose your channel in lines 1010 to 1030. Line 1030 selects the correct range of addresses in

the sound chip to poke. Line 1040 sets up the numbers for a Middle C note, and then GOSUB10100 pokes the quantities into the sound registers. This subroutine is a straightforward sound subroutine, but note lines 10130 and 10140. The waveform number consists of two important parts, one of which selects the waveform while the other acts as a switch for the ADSR system. Subtracting 1 from the usual waveform number has the effect of allowing the AD part to end, so that the sustain-release section can run. if the waveform number is so changed after the time delay, then the correct ADSR effect will be heard. This removal of 1 is carried out by the command in line 10130. If you know enough about machine code to understand the binary AND effect, you will know what has been done here. If you don't understand it, please trust me!

The next part of line 1040 then calls subroutine 30000. This allows a choice of waveform from the two that are on offer. From now on, all of the choices are going to be of numbers, and so a different set of subroutines will handle them. The first of these can act as a model for all the others, so we'll take a close look at lines 1060 to 1090. If you can see what is being done in this set of lines, you'll understand the rest, because they all use the same methods.

In line 1060, three variables are set. MN is the minimum number that can be used for the command, and MX is the maximum. F is a 'flag' variable which will be used if there is to be no change in the quantity. These variables have to be set for each quantity that is to be poked. Line 1070 then shows the range of numbers, and uses GOSUB10000. In this subroutine, the message about entering X to go to the next item is printed. Following this, the GOSUB10010 allows your input and tests it to make sure that it is within the permitted range. If you take the 'X' option, then flag F is made equal to 1, and line 1090 does not finish. If a new item has been entered, F=0, and the test in line 1090 cause the program to recycle round this section.

All of the other entry routines work in this same way. The main differences are that the range of numbers change, and in some cases, the numbers are multiplied by 16 before being used. In the AD and SR sections, for example, the numbers consist of two parts which are added before being poked into memory. The same subroutines are used for each entry routine, and this continues until the questions in lines 1600 to 1620. A 'Y' entered here will cause a return to line 1050, a 'N' to line 1670. At line 1670, the quantities are displayed on the screen, printing the memory numbers and poke quantities in approximate order of address value. That's it! I'll leave you to expand it so as to take in pulses, noise and sound effects!

# Appendix A

## Codes which are stored in address 197 ($C5)

(Keys in row order)

| Key | Code | Key | Code |
|---|---|---|---|
| ← | 57 | I | 33 |
| 1 | 56 | O | 38 |
| 2 | 59 | P | 41 |
| 3 | 8 | @ | 46 |
| 4 | 11 | * | 49 |
| 5 | 16 | ↑ | 54 |
| 6 | 19 | A | 10 |
| 7 | 24 | S | 13 |
| 8 | 27 | D | 18 |
| 9 | 32 | F | 21 |
| 0 | 35 | G | 26 |
| + | 40 | H | 29 |
| − | 43 | J | 34 |
| £ | 48 | K | 37 |
| CLR, HOME | 51 | L | 42 |
| INST, DEL | 0 | [, : | 45 |
| Q | 62 | ], ; | 50 |
| W | 9 | = | 53 |
| E | 14 | RETURN | 1 |
| R | 17 | Z | 12 |
| T | 22 | X | 23 |
| Y | 25 | C | 20 |
| U | 30 | V | 31 |

| Key | Code | Key | Code |
|---|---|---|---|
| B | 28 | ↑ | 7 |
| N | 39 | ↓ | |
| M | 36 | ← | 2 |
| < | 47 | → | |
| , | | SPC | 60 |
| > | 44 | F1 | 4 |
| . | | F3 | 5 |
| ? | 55 | F5 | 6 |
| / | | F7 | 3 |

## ENHANCE YOUR COMMODORE 64!

**This book helps to put your Commodore 64 on a par with much more expensive machines. The collection of tried and tested routines – including programs in machine code where appropriate – will improve its versatility, overcome its limitations and make it easier, more rewarding and interesting to use. It is also the way to extend your repertoire, as you get to know and put into practice the commands used in advanced versions of BASIC. All the routines are short, easy to type in and simple to modify.**

**You will be delighted as your Commodore achieves far more sophisticated capabilities.**

*The Author*

Ian Sinclair has regularly contributed to journals such as *Personal Computer World, Computing Today, Electronics and Computing Monthly, Hobby Electronics* and *Electronics Today International*. He has written over fifty books on aspects of electronics and computing, mainly aimed at the beginner.




Front cover illustration by Jeff Ridge

COLLINS
Printed in Great Britain
0 00 383012 8


**£6.95 net**