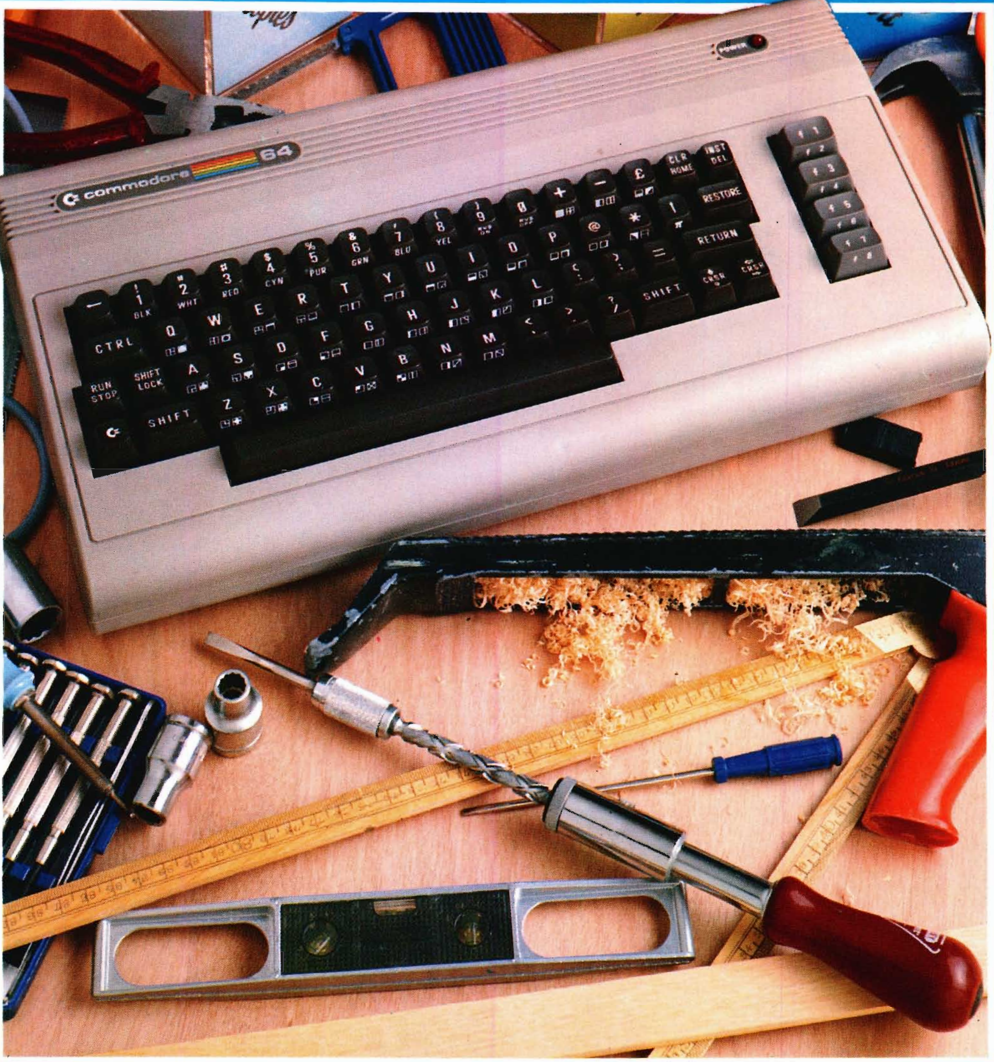
Commodore 64 Assembler Workshop

SHIVA's friendly micro series

Bruce Smith



Commodore 64 Assembler Workshop

Commodore 64 Assembler Workshop

Bruce Smith

Shiva Publishing Limited

SHIVA PUBLISHING LIMITED
64 Welsh Row, Nantwich, Cheshire CW5 5ES, England

ISBN 1 85014 004 9

An interface was used to produce this book from a microcomputer disc, which ensures direct reproduction of error-free program listings.

Typeset and printed by Devon Print Group, Exeter

Contents

Introduction

The *Commodore 64 Assembler Workshop* is aimed at those of you who have been delving into the delights of programming at machine code level. It is a natural progression from *Commodore 64 Assembly Language*, but will be invaluable even if you learned assembler and machine code using any of the other relevant books available. It provides a bench full of useful assembly language routines and utilities programs and examines the techniques involved.

Extensive use of vectored addresses is made throughout the Commodore's operation, allowing modifications to be made to the manner in which the micro operates. Chapter 2 demonstrates how the CHRGET subroutine can be used to allow new RAM-based commands to be added to the already extensive facilities provided by the machine. A short 'wedge' interpreter is provided and the techniques for adding your own commands examined, and to get you going, three commands come supplied with the wedge interpreter: @CLS, @UP and @LOW.

Conversion between ASCII based numerical character strings and their two-byte binary equivalents and vice versa is not straightforward. Such conversions are fully described in Chapters 3 and 4, and working routines are listed.

Any program which handles strings of data must be able to manipulate the strings, whether it is an adventure game or the latest stock control reports. Routines for comparing, copying, deleting and inserting strings are included, and Chapter 6 goes on to show the various ways in which text can be printed to the screen.

Sorting data lists into order is a task which it is often necessary to perform within a program, so the technique of bubble sorting is investigated.

Many other processors provide operations that would be useful to have available when using the 6510. A software stack implementation similar to that found on the 6809 preocessor is produced in Chapter 8, allowing up to eight selected registers to be pushed on to a memory-based stack.

Routines to move, fill and produce a hex and ASCII dump of memory are then examined and the final chapter provides a few hi-resolution graphics utilities to speed you along the way.

Many of the chapters suggest projects for you to undertake at your leisure, while every program has a detailed line-by-line description of its operation. Program listings are provided using BASIC loaders so that they can be used directly as they are. Included in each line is a REM statement giving the mnemonic representation of the instruction should you be using an assembler.

In fact, all the tools for using the *Assembler Workshop* are supplied—assuming of course you have the workbench!

Highbury, November 1984 Bruce Smith

1 Opening the Tool Box

The routines included in this book are designed to make your life that much easier when writing machine code. Quite often, after mastering the delights of the Commodore 64's microprocessor, programmers become frustrated because the techniques involved in, say, converting between ASCII characters and their equivalent binary values are not known. Nor are they readily available in a published form, so the painful process of sitting down armed with pencil and paper and working out the conversion through trial and error begins.

This is just one example of the type of assembler program you will find within these pages. Wherever possible, they are supplied in a form that will make them relocatable, the only addresses requiring alteration being those specified by JSR or JMP.

Each listing is in the form of a BASIC loader program, using a loop to READ and POKE decimal machine code data into memory. This will allow those of you who have not yet splashed out your hard earned cash on a suitable assembler program to get underway. For those lucky ones among you who do have an assembler, each data statement has been followed by a REM line containing the standard mnemonic representation of the instruction (see Appendix 1 for a summary). This can be entered directly and assembled as required.

Although the programs are typeset they have been spooled direct as ASCII files and loaded into my word processor so all should run as they are.

BASIC is used freely to demonstrate the machine code's operation—rather than repeating sections of assembler code, BASIC is often used to shorten the overall listing, and it is left to you to add further sections of assembler from other programs within the book or from your own resources. For example, many programs require you to input a decimal address. In the demonstrations, this is indicated by means of a one-line INPUT statement. In Chapter 3, however, there is a routine for inputting a string of five ASCII decimal characters and converting it into a two-byte binary number. This can be inserted into the assembler text of the program, to go some way to making it a full machine code program available for use as a completely self-contained section of machine code.

WRITING MACHINE CODE

You have an idea that you wish to convert into machine code—so what's the best way to go about it? Firstly, make some brief notes about its operation. Will it use the screen? If so, what mode? Will it require the user to input values from the keyboard? If so, what keys do you use? What will the screen presentation look like? Will you want to use sound?...and so on. Once you have decided on the effects you want, put them down in flowchart form. This need not be the normal flowchart convention of boxes and diamonds—I find it just as easy to write each operation I want the program to perform in a list and then join the flow of these up afterwards.

Quite often, the next step is to write the program in BASIC! This may sound crazy, but it allows you to examine various aspects of the program's operation in more detail. An obvious example of this is obtaining the correct screen layout—you might find after running the routine that the layout does not look particularly good. Finding this out at an early stage will save you a lot of time later, avoiding the need to rewrite the screen layout portion of your machine code—rewriting BASIC is much easier! If you write the BASIC tester as a series of subroutines, it will greatly simplify the process of conversion to machine code. Consider the main loop of such a BASIC tester, which takes the form:

```
10 GOSUB 200  : REM SET UP VARIABLES
20 GOSUB 300  : REM SET UP SCREEN
30              REM LOOP
40 GOSUB 400  : REM INPUT VALUES
50 GOSUB 500  : REM CONVERT AS NEEDED
60 GOSUB 600  : REM DISPLAY VALUES
70 GOSUB 700  : REM DO UPDATE
80 IF TEST=NOTDONE THEN GOTO 30
90 END
```

Each module can be taken in turn, converted into assembler and tested. Once performing correctly the next procedure can be examined. Debugging is made easier because the results of each module are known having used the BASIC tester. The final main loop of the assembler might then look something like this:

```
JSR $C200   : REM SET UP VARIABLES
JSR $C300   : REM SET UP SCREEN
              REM LOOP
JSR $C400   : REM INPUT VALUES
JSR $C500   : REM CONVERT AS NEEDED
JSR $C600   : REM DISPLAY VALUES
JSR $C700   : REM DO UPDATE
BNE LOOP
```

You might be surprised to learn that this technique of testing machine code programs by first using BASIC is employed by many software houses the world over.

DEBUGGING

A word or two about debugging machine code programs that will not perform as you had hoped: if this happens to you, before pulling your hair out and throwing the latest copy of *Machine Code Nuclear Astrophysics Weekly* in the rubbish bin, a check of the following points may reveal the bug!

1. If you are using a commercial assembler, check that your labels have all been declared and correctly assigned. If you are assembling 'by hand', double-check all your branch displacements and JMP and JSR destination addresses. You can normally ascertain exactly where the problem is by examining how much of the program works before the error occurs, rather than checking it all.
2. If your program uses immediate addressing, ensure you have prefixed the mnemonic with a hash (#) to inform the assembler or, if compiling by hand, check that you have used the correct opcode. It is all too easy to assemble the coding for LDA $41 when you really want the coding for LDA #$41.
3. Check that you have set or cleared the Carry flag before subtraction or addition.
4. My favourite now—ensure that you save the result of a subtraction or an addition. The sequence:

```
CLC
LDA $FB
ADC #1
BCC OVER
INC $FC
 OVER
RTS
```

is not much good if you don't save the result of the addition with:

```
STA $FB
```

before the RTS!

5. Does the screen clear to the READY prompt whenever you perform a SYS call, seemingly without executing any of the machine code? The bug that often causes this is due to an extra comma being inserted into a series of DATA statements. For example the DATA line:

```
DATA 169,Ø,,162,255
```

with an extra comma between the Ø and 162, would assemble the following:

```
LDA #$ØØ
BRK
LDX #$FF
```

as the machine has interpreted ',,' as ',Ø,' and assembled the command which has zero as its opcode—BRK!

6. Does the program 'hang up' every time you run it, when you are quite certain that the data statements are correct? This is often caused by a full stop instead of a comma being used between DATA statements, e.g.

```
DATA 169,6,162.5,96
```

Here, if a full stop has been used instead of a comma between the 162 and the 5, the READ command interprets this as a single number, 162.5, rounds it down to 162, and assembles this ignoring the 5 and using the 96 (RTS) as the operand, as follows:

```
LDA #$Ø6
LDX #$6Ø
XXX
```

When executed, the garbage after the last executable instruction results in the system hanging up. This error should not occur if you calculate your loop count correctly, so always double-check this value before running your program.

If none of these errors is the cause of the problem, then I'm afraid you must put your thinking cap on. Well-commented assembler will make debugging very much easier.

2 Commodore Command

One of the disadvantages of using random access memory-based machine code routines as utilities within a BASIC program is that it is left to you, the programmer, to remember just where they are stored, and to use the appropriate SYS call to implement them. This doesn't usually pose any problems if only one or two machine code utilities are present; the problems occur when several are being used. Normally you would need to keep a written list of these next to you, looking up the address of each routine as you need it. Great care must be taken to ensure that the SYS call is made to the correct address, as a mis-typed or wrongly called address can send the machine into an infinite internal loop, for which the only cure is a hard reset, which would destroy all your hard work.

The program offered here provides a useful and exciting solution to the problem, enabling you to add new commands to your Commodore 64's vocabulary. Each of your routines can be given a command name, and the machine code comprising any command will be executed by simply entering its command name. The routine is written so that these new commands can be used either directly from the keyboard or from within programs.

The trick in 'teaching' the Commodore 64 new commands is to get the machine to recognize them. If an unrecognized command is entered at the keyboard, the almost immediate response from the 64 is '?SYNTAX ERROR'. If you have any expansion cartridges you'll know that it is possible to expand the command set, and the *Programmer's Reference Guide* gives a few hints on how to do this, on pages 307 and 308—the method pursued here is by resetting the system CHRGET subroutine.

CHRGET

The CHRGET routine is, in fact, a subroutine which is called by the main BASIC Interpreter. You can think of it as a loop of code, protruding from the machine, into which we can wedge our own bits of code, thereby allowing fundamental changes to be made to the manner in which the Commodore operates. Let's have a look at how the normal CHRGET subroutine (which is located in zero page from $73) operates:

Table 2.1

Address	Machine code	Assembler
$ØØ73	E6 7A	INC $7A
$ØØ75	DØ Ø2	BNE $ØØ79
$ØØ77	E6 7B	INC $7B
$ØØ79	AD xx xx	LDA $xxxx
$ØØ7C	C9 3A	CMP #$3A
$ØØ7E	BØ ØA	BCS $ØØ8A
$ØØ8Ø	C9 2Ø	CMP #$2Ø
$ØØ82	FØ EF	BEQ $ØØ73
$ØØ84	38	SEC
$ØØ85	E9 3Ø	SBC #$3Ø
$ØØ87	38	SEC
$ØØ88	E9 DØ	SBC #$DØ
$ØØ8A	6Ø	RTS

The subroutine begins by incrementing the byte located at $7A. This address forms a vector which holds the address of the interpreter within the BASIC program that is currently being run. If there is no carry over into the high byte, which must therefore itself be incremented, a branch occurs to location $ØØ79. You will notice that the bytes which have just been incremented lie within the subroutine itself. These are signified in the above listing by 'xx xx', because they are being updated continually by the routine. The reason for this should be fairly self-evident: looking at the opcode, we can see that it is LDA, therefore each byte is, in turn, being extracted from the program.

The next two bytes at $ØØ7C perform a compare, CMP #$3A. The operand here, $3A, is the ASCII code for a colon, so CHRGET is checking for a command delimiter. The BCS $ØØ8A will occur if the accumulator contents are greater than $3A, effec tively returning control back to within the BASIC Interpreter ROM. The next line, CMP #$2Ø, checks whether a space has been encountered within the program. If it has, the branch is executed back to $ØØ73 and the code rerun.

The rest of the coding is checking that the byte is a legitimate one—it should be an ASCII character code in the range $3Ø to $39, that is, a numeric code. If it is, the coding will return to the main interpreter with the Carry flag clear. If the accumulator contains less than $3Ø (it could, of course, have ASCII $2Ø in it, as we have already checked for this) then the Carry flag is set.

It is important to understand what is happening here, as we will need to overwrite part of this code to point it in the direction of our own 'wedge' interpreter. This has to perform the 'deleted' tasks before returning to the main interpreter to ensure the smooth and correct running of the Commodore 64.

THE WEDGE OPERATING SYSTEM

To distinguish the Wedge Operating System (WOS) commands from normal commands (and illegal ones!), we must prefix them with a special character—one which is not used by the Commodore 64. The *Programmer's Reference Guide* suggests the use of the the '@' sign, so that's what we will use.

Program 1a lists the coding for the WOS. I have chosen to place it well out of the way, in the free RAM area from 49666 (3C2Ø2) onwards. As we shall see the memory below (bis to 49152 ($CØØØ) is also used by the WOS.

Program 1a

```
10 REM *** WEDGE OPERATING SYSTEM - WOS ***
20 REM *** WOS INTERPRETER FOR COMMODORE 64 ***
30 :
40 CODE=49666
50 FOR LOOP=0 TO 188
60  READ BYTE
70  POKE CODE+LOOP,BYTE
80 NEXT LOOP
90 :
100 REM ** M/C DATA **
110 DATA  169,0       : REM  LDA #$00
120 DATA  160,192     : REM  LDY #$C0
130 DATA  32,30,171   : REM  JSR $AB1E
140 DATA  169,76      : REM  LDA #$4C
150 DATA  133,124     : REM  STA $7C
160 DATA  169,24      : REM  LDA #$18
170 DATA  133,125     : REM  STA $7D
180 DATA  169,194     : REM  LDA #$C2
190 DATA  133,126     : REM  STA $7E
```

```
200 DATA  108,2,3      : REM  JMP ($0302)
205 ::                   REM WOS STARTS HERE
210 DATA  201,64       : REM  CMP #$40
220 DATA  208,68       : REM  BNE $44
230 DATA  165,157      : REM  LDA $9D
240 DATA  240,40       : REM  BEQ $28
250 DATA  173,0,2      : REM  LDA $0200
260 DATA  201,64       : REM  CMP #$40
270 DATA  208,28       : REM  BNE $1C
280 DATA  32,114,194 : REM  JSR $C272
290 DATA  160,0        : REM  LDY #$00
300 DATA  177,122      : REM  LDA ($7A),Y
310 DATA  201,32       : REM  CMP #$20
320 DATA  240,9        : REM  BEQ $09
330 DATA  230,122      : REM  INC $7A
340 DATA  208,246      : REM  BNE $F6
350 DATA  230,123      : REM  INC $7B
360 DATA  56           : REM  SEC
370 DATA  176,241      : REM  BCS $F1
380 DATA  32,116,164 : REM  JSR $A474
390 DATA  169,0        : REM  LDA #$00
400 DATA  56           : REM  SEC
410 DATA  176,29       : REM  BCS $1D
420 DATA  169,64       : REM  LDA #$40
430 DATA  56           : REM  SEC
440 DATA  176,24       : REM  BCS $18
445 ::                   REM PROGRAM-MODE
450 DATA  32,114,194 : REM  JSR $C272
460 DATA  160,0        : REM  LDY #$00
470 DATA  177,122      : REM  LDA ($7A),Y
480 DATA  201,0        : REM  CMP #$00
490 DATA  240,13       : REM  BEQ $0D
500 DATA  201,58       : REM  CMP #$3A
510 DATA  240,9        : REM  BEQ $09
520 DATA  230,122      : REM  INC $7A
530 DATA  208,242      : REM  BNE $F2
540 DATA  230,123      : REM  INC $7B
550 DATA  56           : REM  SEC
560 DATA  176,237      : REM  BCS $ED
```

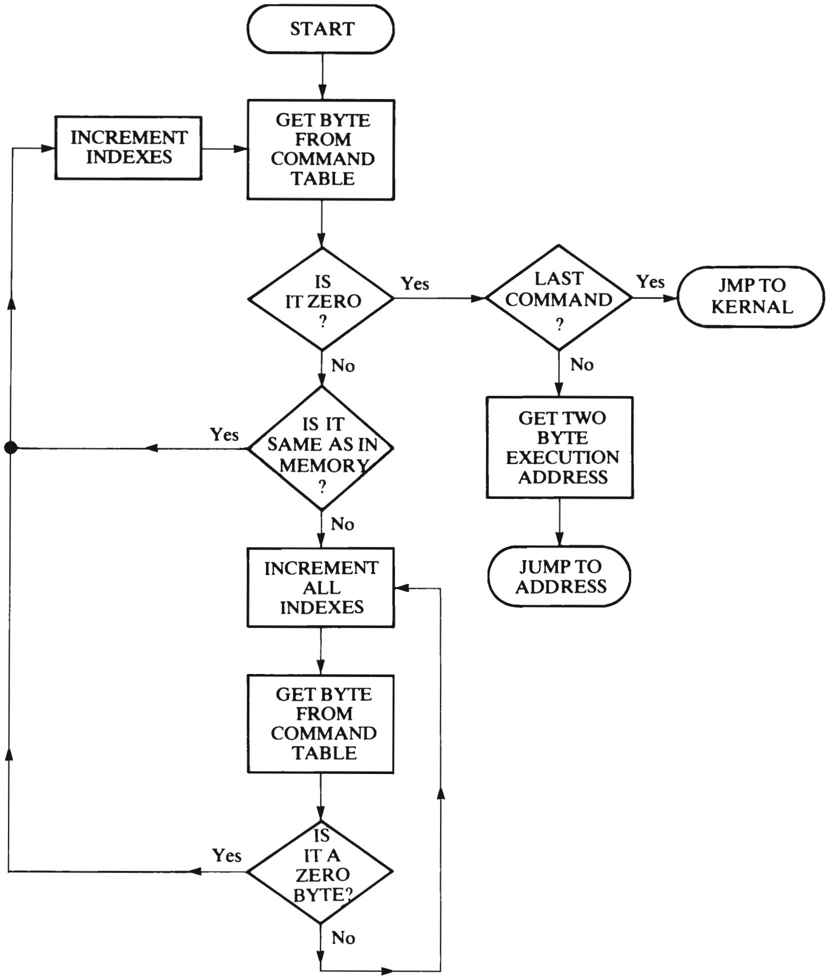


Figure 2.1 The wedge operating system flowchart

```
570 DATA  201,58      : REM  CMP #$3A
580 DATA  176,10      : REM  BCS $0A
590 DATA  201,32      : REM  CMP #$20
600 DATA  240,7       : REM  BEQ $07
610 DATA  56          : REM  SEC
620 DATA  233,48      : REM  SBC #$30
630 DATA  56          : REM  SEC
640 DATA  233,208     : REM  SBC #$D0
650 DATA  96          : REM  RTS
660 DATA  76,115,0    : REM  JMP $0073
665 ::                  REM FIND-EXECUTE
```

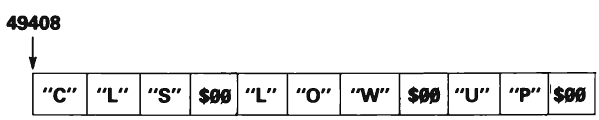
```
670 DATA  169,0       : REM  LDA #$00
680 DATA  133,127     : REM  STA $7F
690 DATA  169,193     : REM  LDA #$C1
700 DATA  133,128     : REM  STA $80
710 DATA  230,122     : REM  INC $7A
720 DATA  208,2       : REM  BNE $02
730 DATA  230,133     : REM  INC $7B
740 DATA  160,0       : REM  LDY #$00
750 DATA  162,0       : REM  LDX #$00
760 DATA  177,127     : REM  LDA ($7F),Y
770 DATA  240,36      : REM  BEQ $24
780 DATA  209,122     : REM  CMP ($7A),Y
790 DATA  208,4       : REM  BNE $04
800 DATA  200         : REM  INY
810 DATA  56          : REM  SEC
820 DATA  176,244     : REM  BCS $F4
830 DATA  177,127     : REM  LDA ($7F),Y
840 DATA  240,4       : REM  BEQ $04
850 DATA  200         : REM  INY
860 DATA  56          : REM  SEC
870 DATA  176,248     : REM  BCS $F8
880 DATA  200         : REM  INY
890 DATA  152         : REM  TYA
900 DATA  24          : REM  CLC
910 DATA  101,127     : REM  ADC $7F
920 DATA  133,127     : REM  STA $7F
930 DATA  169,0       : REM  LDA #$00
940 DATA  101,128     : REM  ADC $80
950 DATA  133,128     : REM  STA $80
960 DATA  160,0       : REM  LDY #$00
970 DATA  232         : REM  INX
980 DATA  232         : REM  INX
990 DATA  56          : REM  SEC
1000 DATA 176,216     : REM  BCS $D8
1010 DATA 189,80,192  : REM  LDA $C050,X
1020 DATA 133,128     : REM  STA $80
1030 DATA 232         : REM  INX
1040 DATA 189,80,192  : REM  LDA $C050,X
1050 DATA 133,129     : REM  STA $81
1060 DATA 108,128,0   : REM  JMP ($0080)
```

```
1065 ::                        REM ILLEGAL
1070 DATA  162,11        : REM  LDX #$0B
1080 DATA  108,0,3       : REM  JMP ($300)
1090 :
1100 REM ** SET UP COMMAND TABLE **
1110 TABLE=49408
1120 FOR LOOP=0 TO 10
1130  READ BYTE
1140  POKE TABLE+LOOP,BYTE
1150 NEXT LOOP
1160 :
1170 REM ** ASCII COMMAND DATA **
1180 DATA 67,76,83,0   : REM CLS
1190 DATA 76,79,87,0   : REM LOW
1200 DATA 85,80,0      : REM UP
```

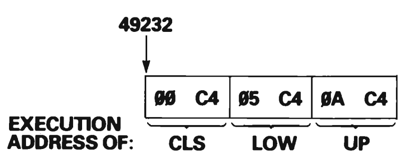
To enable the WOS to identify a wedge command, it needs a complete list to which it can compare the one it is interpreting in the program—this is done with the aid of a command table, which is formed by the program lines from 1100 to 1200. This ASCII table is based at 49408 ($C100) and, as you can see from the listing, three commands are provided: @CLS, @LOW and @UP. Note that the @ is omitted from the front of each command in the table—it is unnecessary at the comparison stage, as by this time it has already been established that it is a WOS command—and that each command is terminated by a zero. A table listing the execution address of each command must also be constructed, but more of this later.

The main program consists of two parts, an initialization routine and the interpreter proper.

The initialization routine is embodied in lines 110 to 200. Its function is to reset the CHRGET subroutine investigated earlier. Lines 110 to 130 issue a heading on the screen indicating that the



The Command Table



The Address Table

Figure 2.2 The Command and Address Tables.

WOS has been initialized. The subroutine at \$AB1E, called by line 130, prints out an ASCII string located at the address given by the index registers. In this instance it is located at \$C000 (49152), and is assembled into memory by the second part of the listing. Lines 140 to 190 poke three bytes into the CHRGET subroutine which effectively assembles the code:

```
JMP $C218
```

The address \$C218 is the address of the start of the WOS interpreter at line 210. Finally, line 200 does an indirect jump through the IMAN vector at \$0302 to perform a warm BASIC start. The CHRGET subroutine, complete with wedge jump, now looks like this:

Table 2.2

Address	Machine code	Assembler
\$0073	E6 7A	INC \$7A
\$0075	D0 02	BNE \$0079
\$0077	E6 7B	INC \$7B
\$0079	AD xx xx	LDA \$xxxx
\$007C	4C 18 C2	JMP \$C218

When the WOS is entered, the byte in the accumulator is checked to see if it is an @ (line 210), signifying a wedge command. If it is not, then a branch to line 570 is performed. As you can see, the code from line 570 to 650 performs the normal function of the CHRGET routine, with control returning to the BASIC Interpreter.

If the byte is an @ , the interpretation continues. The byte at \$9D is located, to detect whether the command is within a program or has been issued in direct mode. A zero indicates that the command has been called from within a program and the branch of line 240 to line 450 is performed. In both instances the interpretation follows similar lines—for descriptive purposes, we will assume program mode and resume the commentary from line 450.

The subroutine at \$C272 is the interpreter proper. Starting at line 665 it locates the command and executes it. The first eight bytes (lines 670 to 700) set up a zero page vector to point to the command table at \$C100. Lines 710 to 730 update the zero page bytes at \$7A and \$7B, which hold the address of the current point within the program. After initializing both index registers, the first byte within the command table is located (lines 74Ø to 76Ø), and compared to the byte within the program, immediately after the @ (line 78Ø). If the comparison fails, the branch to line 83Ø is performed, locating the zero and therefore the next command in the command table. When a comparison is successful (the command is identified) and the terminating zero located by line 77Ø, the branch to line 1Ø1Ø is performed. Lines 1Ø1Ø to 1Ø6Ø locate the execution address of the command from the address table located at $CØ5Ø. The X register is used as an offset into this, being incremented by two each time a command table comparison fails (lines 97Ø and 98Ø). The two address bytes are loaded to form a zero page vector and the machine code is executed via an indirect jump.

On completion of the routine, its terminating RTS returns control to line 46Ø, and the next byte after the command is sought out. When a zero is found, the branch of line 49Ø is performed and the CHRGET routine is completed, control being returned to the BASIC Interpreter.

THE NEW COMMANDS

Program 1b provides the assembly routines to construct the initialization prompts, the machine code for the new commands and the address table:

Program 1b

```
121Ø REM ** TITLE MESSAGE DISPLAYED ON SYS
     49666 **
122Ø HEAD=49152
123Ø FOR LOOP=Ø TO 4Ø
124Ø  READ BYTE
125Ø  POKE HEAD+LOOP,BYTE
126Ø NEXT LOOP
127Ø :
128Ø REM ** ASCII CHARACTER DATA **
129Ø DATA 147,13,32,32,42,42,32,67,54,52,32
13ØØ DATA 69,88,84,69,78,68,69,68,32,83,85
12ØØ DATA 8Ø,69,82,32,66,65,83,73,67,32,86,49
131Ø DATA 46,48,32,42,42,13,Ø
132Ø ::
136Ø REM ** SET UP M/C FOR COMMANDS **
137Ø MC=5Ø176
138Ø FOR LOOP=Ø TO 14
139Ø  READ BYTE
```

```
1400  POKE MC+LOOP,BYTE
1410 NEXT LOOP
1420 :
1430 REM ** COMMAND M/C **
1440 ::                   REM CLS
1450 DATA 169,147       : REM LDA #$93
1460 DATA 76,210,255    : REM JMP $FFD2
1470 ::                   REM LOW
1480 DATA 169,14        : REM LDA #$0E
1490 DATA 76,210,255    : REM JMP $FFD2
1500 ::                   REM UP
1510 DATA 169,142       : REM LDA #$8D
1520 DATA 76,210,255    : REM JMP $FFD2
1530 ::
1540 REM ** SET UP ADDRESS TABLE **
1550 ADDR=49232
1560 FOR LOOP=0 TO 5
1570  READ BYTE
1580  POKE ADDR+LOOP,BYTE
1590 NEXT LOOP
1600 :
1610 REM ** ADDRESS DATA **
1620 DATA 0,196         : REM CLS $C400
1630 DATA 5,196         : REM LOW $C405
1640 DATA 10,196        : REM UP  $C40A
```

Each command's machine code is located from 50176 ($C400). The three new commands and their functions are:

CLS	:	clear screen and home cursor
LOW	:	select lower case character set
UP	:	select upper case character set

Nothing to set the house alight, admittedly, but the techniques involved are more important at present. These are simple to implement and, once understood, enable more useful and complex commands to be added. The code associated with each command is responsible simply for printing its ASCII code. The final section of listing (lines 1540 to 1650) pokes the execution address of each command into memory. The final address points to the code at line 170, and the program jumps to this position if the command is not found within the command table. This code performs an indirect jump to the BASIC Interpreter's error handler.

USING THE WOS

Using the Wedge Operating System is easy: enter the program as shown, run it to assemble the code into memory, and if all goes well, save the program. To initialize the WOS enter:

```
SYS 49666
```

The screen will clear, and the following message be printed across the top of the screen:

```
** C64 EXTENDED SUPER BASIC V1.Ø **
```

The wedge commands are now available for immediate use. Remember that pressing RUN/STOP and RESTORE together will reset the CHRGET routine to its default value making the WOS invisible. To relink it, simply execute the SYS 49666 call again.

Line-by-line

A line-by-line description of the WOS now follows, to enable you to examine its operation in more detail:

line 11Ø : load accumuator with low byte message address
line 12Ø : load accumulator with high byte message address
line 13Ø : print start up message
line 14Ø : reset CHRGET subroutine
line 2ØØ : do a BASIC warm start
line 2Ø5 : main entry for WOS
line 21Ø : is it an '@' and therefore a WOS command?
line 22Ø : no, so branch to line 57Ø to update
line 23Ø : yes, check for direct or program mode
line 24Ø : if zero, then WOS command is within program, so branch to line 45Ø
line 25Ø : else direct mode so get byte from buffer
line 26Ø : recheck that it is a WOS command
line 27Ø : if error, branch to line 41Ø
line 28Ø : find and execute the command else issue appropriate error message
line 29Ø : initialize index
line 3ØØ : get byte from buffer
line 31Ø : is it a space?
line 32Ø : yes, so branch to line 38Ø

line 33Ø : increment low byte of address

line 34Ø : branch back to line 3ØØ if high byte does not need to be updated

line 35Ø : else increment high byte of address

line 36Ø : set Carry flag and do a forced branch back to line 3ØØ

line 38Ø : print 'READY' prompt

line 39Ø : clear accumulator

line 4ØØ : set Carry flag and force a branch to line 5ØØ to update and return

line 42Ø : get '@' into accumulator

line 43Ø : set Carry flag and force a branch to line 57Ø

line 445 : entry point for PROGRAM-MODE

line 45Ø : locate and execute command or print appropriate error message

line 46Ø : clear indexing register

line 47Ø : get byte from program

line 48Ø : is it a Ø and therefore end of line?

line 49Ø : yes, branch to line 5ØØ

line 5ØØ : no, is it the command delimiter ':'?

line 51Ø : yes, branch to line 57Ø

line 52Ø : no, increment low byte of address

line 53Ø : if not zero, branch back to line 47Ø to redo loop

line 54Ø : increment high byte of address

line 55Ø : set Carry flag and force a branch back to line 47Ø.

line 57Ø : is it a command delimiter ':'?

line 58Ø : if greater than or equal to ':' then branch to line 65Ø

line 59Ø : is it a space?

line 6ØØ : yes, so branch to line 65Ø

line 61Ø : set Carry flag

line 62Ø : subtract ASCII base code

line 63Ø : set Carry flag

line 64Ø : subtract token and ASCII set bits

line 65Ø : return to BASIC Interpreter

line 66Ø : jump to CHRGET

line 665 : entry for FIND-EXECUTE subroutine

line 67Ø : seed address of command table ($C1ØØ) into vector at $7F

line 71Ø : increment low byte of command address

line 72Ø : branch over if no carry into high byte line 73Ø : else increment high byte of address

line 74Ø : back together, initialize Y register

line 75Ø : and X register

line 76Ø : get byte from the command table

line 77Ø : if zero byte, then command is identified, branch to line 1Ø1Ø

line 78Ø : is it the same as the byte pointed to in the command table?

line 79Ø : no, branch to line 83Ø

line 8ØØ : increment index

line 82Ø : set Carry flag and force a branch back to line 76Ø

line 83Ø : command not identified—seek out zero byte. Get byte from command table

line 84Ø : if zero, branch to line 88Ø

line 85Ø : increment index

line 86Ø : set Carry flag and force a branch to line 83Ø

line 88Ø : increment index

line 89Ø : transfer into accumulator

line 9ØØ : clear Carry flag

line 91Ø : add to low byte of vector address

line 92Ø : save result

line 93Ø : clear accumulator

line 94Ø : add carry to high byte of vectored address

line 95Ø : and save the result

line 96Ø : initialize index

line 97Ø : add two to X to move onto next address in the

line 98Ø : command address table

line 99Ø : set Carry flag and force a branch to line 76Ø

line 1Ø1Ø : get low byte of command execution address

line 1Ø2Ø : save it in a vector

line 1Ø3Ø : increment index

line 1Ø4Ø : get high byte of command execution address

line 1Ø5Ø : save it in vector

line 1Ø6Ø : jump to vectored address to execute machine code of identified command

line 1Ø65 : entry for ILLEGAL—unrecognized WOS command

line 1Ø7Ø : get error code into X register

line 1Ø8Ø : and jump to error handling routine

3 ASCII to Binary Conversions

An important aspect of interactive machine code is the ability to convert strings of ASCII characters into their hexadecimal equivalents, so that they may be manipulated by the processor. In this chapter we shall examine, with program examples, how this is performed. The routines provide the following conversions:

1. Single ASCII hex characters into binary.
2. Four ASCII hex digits into two hex bytes.
3. Signed ASCII decimal string into two signed hex bytes.

ASCII HEX TO BINARY CONVERSION

This routine converts a hexadecimal ASCII character in the accumulator into its four-bit binary equivalent. For example, if the accumulator contains $37 (that is, ASC"7"), the routine will result in the accumulator holding $7, or ØØØØØ111 binary. Similarly, if the accumulator holds $46 (ASC"F") the routine will return $F, or ØØØØ1111, in the accumulator.

Conversion is quite simple, and Table 3.1 gives some indication of what is required.

Table 3.1

Hex	Binary value	ASCII value	ASCII binary
Ø	ØØØØØØØØ	$3Ø	ØØ11ØØØØ
1	ØØØØØØØ1	$31	ØØ11ØØØ1
2	ØØØØØØ1Ø	$32	ØØ11ØØ1Ø
3	ØØØØØØ11	$33	ØØ11ØØ11
4	ØØØØØ1ØØ	$34	ØØ11Ø1ØØ

Table 3.1 (contd.)

5	ØØØØØ1Ø1	$35	ØØ11Ø1Ø1
6	ØØØØØ11Ø	$36	ØØ11Ø11Ø
7	ØØØØØ111	$37	ØØ11Ø111
8	ØØØØ1ØØØ	$38	ØØ111ØØØ
9	ØØØØ1ØØ1	$39	ØØ111ØØ1
A	ØØØØ1Ø1Ø	$41	Ø1ØØØØØ1
B	ØØØØ1Ø11	$42	Ø1ØØØØ1Ø
C	ØØØØ11ØØ	$43	Ø1ØØØØ11
D	ØØØØ11Ø1	$44	Ø1ØØØ1ØØ
E	ØØØØ111Ø	$45	Ø1ØØØ1Ø1
F	ØØØØ1111	$46	Ø1ØØØ11Ø

The conversion of ASCII characters Ø to 9 is straightforward. All we need to do is mask off the high nibble of the character's ASCII code. For example ASC "1" is $31 or ØØ11ØØØ1 binary—masking the high nibble with AND $OF results in ØØØØØØØ1. Converting ASCII characters A and F is a little less obvious, however. If the high nibble of the code is masked off, then the remaining bits are 9 less than the hex required. For example, the ASCII for the letter 'D' is $44 or Ø1ØØØ1ØØ. Masking the high nibble with AND $OF gives 4, or ØØØØØ1ØØ, and adding 9 to this gives:

```
  ØØØØØ1ØØ
+ ØØØØ1ØØ1
  --------
  ØØØØ11Ø1
```

the binary value for $D.

Program 2

```
 1Ø REM ** CONVERT ASCII CHARACTER IN **
 2Ø REM ** ACCUMULATOR TO BINARY **
 3Ø REM ** REQUIRES 2Ø BYTES OF MEMORY **
 4Ø :
 5Ø CODE=49152
 6Ø FOR LOOP=Ø TO 2Ø
 7Ø  READ BYTE
 8Ø  POKE CODE+LOOP,BYTE
 9Ø NEXT LOOP
1ØØ :
```

```
11Ø REM ** M/C DATA **
12Ø DATA  2Ø1,48        REM  CMP #$3Ø
13Ø DATA  144,15        REM  BCC $ØF
14Ø DATA  2Ø1,58        REM  CMP #$3A
15Ø DATA  144,8         REM  BCC $Ø8
16Ø DATA  233,7         REM  SBC #$Ø7
17Ø DATA  144,7         REM  BCC $Ø7
18Ø DATA  2Ø1,64        REM  CMP #$4Ø
19Ø DATA  176,2         REM  BCS $Ø2
2ØØ ::                  REM ZERO-NINE
21Ø DATA  41,15         REM  AND #$ØF
22Ø ::                  REM RETURN
23Ø DATA  96            REM  RTS
24Ø ::                  REM ILLEGAL
25Ø DATA  56            REM  SEC
26Ø DATA  96            REM  RTS
27Ø :
28Ø :
29Ø REM ** TESTING ROUTINE **
3ØØ TEST=49184
31Ø FOR LOOP=Ø TO 14
32Ø  READ BYTE
33Ø  POKE TEST+LOOP,BYTE
34Ø NEXT LOOP
35Ø :
36Ø REM ** M/C TEST DATA **
37Ø ::                     REM TEST
38Ø DATA  32,228,255       REM  JSR $FFE4
39Ø DATA  24Ø,251          REM  BEQ $FB
4ØØ DATA  32,Ø,192         REM  JSR $CØØØ
41Ø DATA  144,2            REM  BCC $Ø2
42Ø DATA  169,255          REM  LDA #$FF
43Ø ::                     REM OVER
44Ø DATA  133,251          REM  STA $FB
45Ø DATA  96               REM  RTS
46Ø :
47Ø PRINT CHR$(147)
48Ø PRINT"HIT A HEX CHARACTER KEY, AND ITS
    BINARY"
49Ø PRINT"EQUIVALENT VALUE WILL BE PRINTED"
```

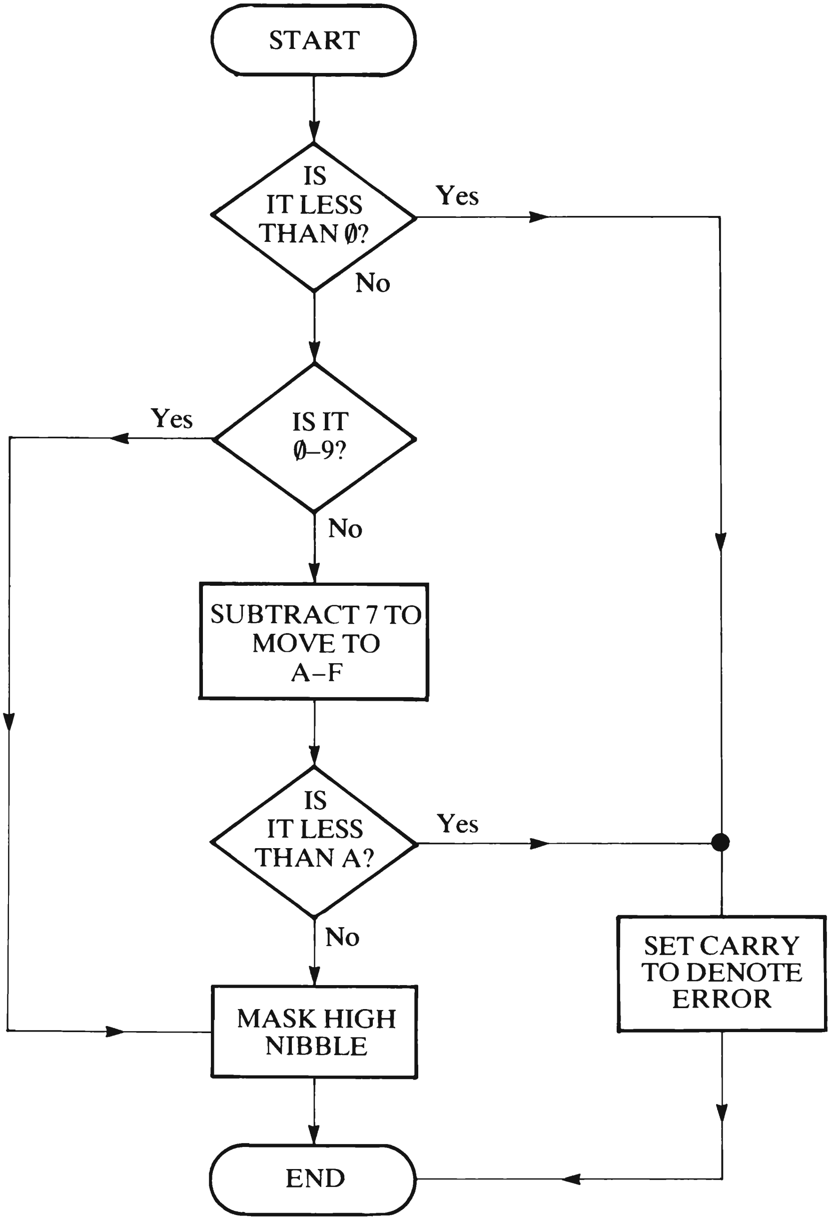


Figure 3.1 Conversion flowchart

```
5ØØ :
51Ø SYS TEST
52Ø :
53Ø PRINT "RESULT = "PEEK(251)
```

Program 2 contains a short demonstration, prompting for a hexadecimal value key to be pressed (i.e. Ø to F) and returning its hexadecimal code. Thus, pressing the 'A' key will produce a result of 41.

The ASCII-BINARY routine begins by checking for the legality of the character, by comparing it with 48 ($3Ø). If the value in the accumulator is less than ASC"Ø", the Carry flag will be cleared, signalling an error. If the character is legal, the contents are then compared with 58 ($3A), which is one greater than the ASCII code for 9. This part of the routine ascertains whether the accumulator's contents are in the range $3Ø to $39. If they are, the Carry flag will be cleared and the branch to ZERO-NINE (lines 15Ø and 12Ø) performed. The high nibble is then masked off to complete the conversion.

If the branch of line 15Ø fails, a legality check for the hex characters A to F is performed. This is done by subtracting 7 from the accumulator's contents, which should bring the value it holds down below 64 ($4Ø), or one less than the ASCII code for the letter 'A'. At this point the Carry flag is set (it was previously set as the branch of the previous line was not performed), and the CMP #$4Ø of line 18Ø clears it if the contents are higher than 64. The routine then masks off the high nibble, leaving the correct binary.

The following example shows how the conversion of ASC"F" to $F works:

Mnemonic	Accumulator	Carry flag
	$46 (ASC"F")	
CMP #$3Ø	$46	1
BCC ILLEGAL		
CMP #$3A	$46	1
BCC ZERO-NINE		
SBC #7	$3F	1
BCC ILLEGAL		
CMP #$4Ø	$3F	Ø
BCS RETURN		
AND $ØF	$ØF	Ø
RTS		

Note that this routine indicates an error by returning with the Carry flag set, so any calls to the conversion routine should always check for this on return. The short test routine does this, and loads the accumulator with $FF to signal the fact.

Using two calls to this routine would allow two-byte hex values to be input and converted into a full eight-byte value. On completion of the first call, the accumulator's contents would need to be shifted into the high nibble.

The coding might look like this:

```
                              : REM WAIT
   JSR GETIN                  : REM GET FIRST CHARACTER
   BEQ WAIT1
   JSR ASCII-BINARY           : REM CONVERT TO BINARY
   BCS REPORT-ERROR           : REM NON-HEX IF C=1
   ASL A                      : REM MOVE INTO HIGHER
                                NIBBLE
   ASL A
   ASL A
   ASL A
   STA HIGH-NIBBLE            : REM SAVE RESULT
                              : REM WAIT2
   JSR GETIN                  : REM GET SECOND CHARACTER
   BEQ WAIT2
   JSR ASCII-BINARY           : REM CONVERT TO BINARY
   BCS REPORT-ERROR           : REM NON-HEX IF C=1
   ORA HIGH-NIBBLE            : REM ADD HIGH NIBBLE
                              : REM ALL BINARY NOW IN
                                ACCUMULATOR
```

Using this routine and entering, say, $FE will return 1111111Ø in the accumulator.

Line-by-line

A line-by-line description of Program 2 follows:

line 12Ø : is it >= than ASC"Ø"?
line 13Ø : no, branch to ILLEGAL
line 14Ø : is it in range Ø–9?
line 15Ø : yes, branch to ZERO-NINE to skip A–F translation.
line 16Ø : move onto ASCII codes for A–F
line 17Ø : branch to ILLEGAL if Carry flag clear
line 18Ø is it higher than ASC"@"?
line 19Ø : no, branch to ILLEGAL
line 2ØØ : entry for ZERO-NINE
line 21Ø : clear high nibble
line 22Ø : entry for RETURN

line 230 : return with binary in accumulator
line 240 : entry for ILLEGAL
line 250 : set Carry flag to denote an error
line 260 : return to BASIC
line 370 : entry for TEST
line 380 : read keyboard
line 390 : if null string, branch to TEST
line 400 : call conversion at $C000
line 410 : if no errors, branch OVER
line 420 : else error, place 255 in accumulator
line 430 : entry for OVER
line 440 : save accumulator in $FB
line 450 : and return to BASIC

FOUR ASCII DIGITS TO HEX

We can use the ASCII-BINARY routine as the main subroutine in a piece of coding which will convert four ASCII digits into a two-byte hexadecimal number, making the routine most useful for inputting two-byte hexadecimal addresses. For example, the routine would convert the ASCII string "CAFE" into a two-byte binary number 11001010 11111110 or $CAFE. Program 3 lists the entire coding:

Program 3

```
10 REM   ** CONVERT FOUR ASCII DIGITS INTO **
20 REM   ** A TWO-BYTE HEXADECIMAL NUMBER **
30 CODE=49152
40 FOR LOOP=0 TO 62
50  READ BYTE
60  POKE CODE+LOOP,BYTE
70 NEXT LOOP
80 :
90 REM   ** M/C DATA **
100 DATA 160,0        : REM   LDY #0
110 DATA 162,251      : REM   LDX #$FB
120 DATA 148,0        : REM   STY $00,X
130 DATA 148,1        : REM   STY $01,X
140 DATA 148,2        : REM   STY $02,X
150 ::                  REM NEXT-CHARACTER
```

```
160 DATA 185,60,3       : REM  LDA $33C,Y
170 DATA 32,42,192      : REM  JSR $C02A
180 DATA 176,21         : REM  BCS $15
190 DATA 10,10          : REM  ASL  A : ASL A
200 DATA 10,10          : REM  ASL  A : ASL A
210 DATA 148,2          : REM  STY $02,X
220 DATA 160,4          : REM  LDY #$04
225 ::                    REM AGAIN
230 DATA 10             : REM  ASL A
240 DATA 54,0           : REM  ROL $00,X
250 DATA 54,1           : REM  ROL $01,X
260 DATA 136            : REM  DEY
270 DATA 208,248        : REM  BNE $F8
280 DATA 180,2          : REM  LDY $02,Y
290 DATA 200            : REM  INY
300 DATA 208,227        : REM  BNE $E3
310 ::                    REM ERROR
320 DATA 181,2          : REM  LDA $02,X
330 DATA 96             : REM  RTS
340 :
350 REM  *** ASCII-BINARY CONVERSION ***
360 DATA 201,48         : REM  CMP #$30
370 DATA 144,15         : REM  BCC $0F
380 DATA 201,58         : REM  CMP #$3A
390 DATA 144,8          : REM  BCC $08
400 DATA 233,7          : REM  SBC $07
410 DATA 144,7          : REM  BCC $07
420 DATA 201,64         : REM  CMP #$40
430 DATA 176,2          : REM  BCS $02
440 ::                    REM ZERO-NINE
450 DATA 41,15          : REM  AND $0F
460 ::                    REM RETURN
470 DATA 96             : REM  RTS
480 ::                    REM ILLEGAL
490 DATA 56             : REM  SEC
500 DATA 96             : REM  RTS
510 :
520 REM  *** SET UP A TEST PROCEDURE ***
530 TEST=49232
540 FOR LOOP=0 TO 34
```

```
550  READ BYTE
560  POKE TEST+LOOP,BYTE
570 NEXT LOOP
580 :
590 REM  ** TEST M/C DATA **
600 DATA 160,0          : REM  LDY #$00
610 DATA 162,4          : REM  LDX #$04
620 ::                    REM OVER
630 DATA 142,52,3       : REM  STX $334
640 DATA 140,53,3       : REM  STY $335
650 ::                    REM INNER
660 DATA 32,228,255     : REM  JSR $FFE4
670 DATA 240,251        : REM  BEQ $FB
680 DATA 174,52,3       : REM  LDX $334
690 DATA 172,53,3       : REM  LDY $335
700 DATA 153,60,3       : REM  STA $33C,Y
710 DATA 32,210,255     : REM  JSR $FFD2
720 DATA 200            : REM  INY
730 DATA 202            : REM  DEX
740 DATA 208,229        : REM  BNE $E5
750 DATA 32,0,192       : REM  JSR $C000
760 DATA 96             : REM  RTS
770 :
780 PRINT CHR$(147)
790 PRINT "INPUT A FOUR DIGIT HEX NUMBER : $";
800 SYS TEST
810 PRINT
820 PRINT "THE FIRST  BYTE WAS :";PEEK(251)
830 PRINT "THE SECOND BYTE WAS :";PEEK(252)
```

The machine code begins by clearing three bytes of zero page RAM pointed to by the contents of the X register (lines 100 to 140). The ASCII characters are accessed one by one from a buffer which may be resident anywhere in memory (line 160), though in this case it is the four bytes at the start of the cassette buffer. Conversion and error-detection are performed (lines 170 and 180) and the four returned bits shifted into the high four bits of the accumulator. The buffer index, which keeps track of the character position in the buffer, is saved in the third of the three bytes cleared.

The loop between lines 250 and 300 is responsible for moving the four bits through the two zero page bytes which hold the final result. In fact, with the accumulator, the whole process of the loop is to perform the operation of a 24-bit shift register. Figure 3.2

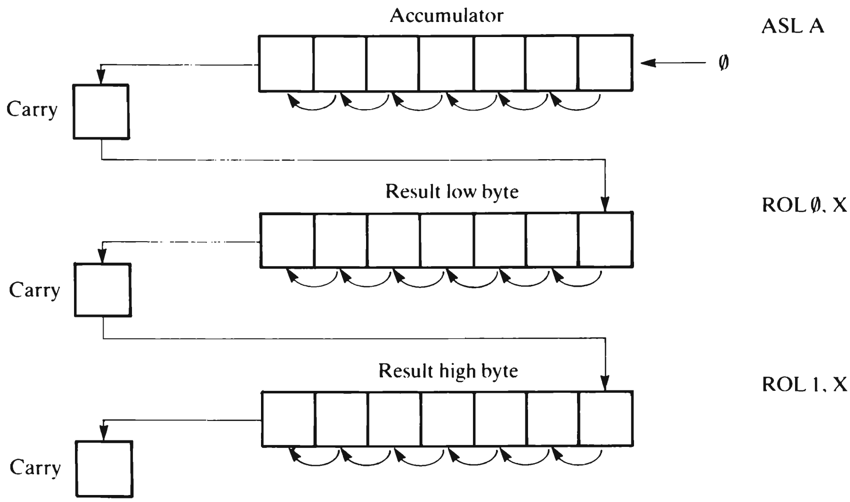


Figure 3.2 Movement of bits through a 3-byte shift register

illustrates the procedure.

The ASL A instruction shuffles the bits in the accumulator one bit to the left, with the dislodged bit 7 moving across into the Carry flag bit. This carry bit is then rotated into bit Ø of the result address low byte, which in turn rotates its bit 7 into the Carry flag. The next ROL instruction repeats this movement on the high byte. The net effect of all this is that as the process is executed four times, the returned conversions are shifted through the result address to reside in the correct place, as Figure 3.3 illustrates.

	1, X	Ø, X	Accumulator
Entry	ØØØØØØØØ	ØØØØØØØØ	1111ØØØØ
1st pass	ØØØØØØØØ	ØØØØ1111	ØØØØØØØØ
2nd pass	ØØØØØØØØ	1111ØØØØ	ØØØØØØØØ
3rd pass	ØØØØ1111	ØØØØØØØØ	ØØØØØØØØ
4th pass	1111ØØØØ	ØØØØØØØØ	ØØØØØØØØ

Figure 3.3 A 24-bit shift register, showing passage of the bits in the number $FØØØ

Error-checking is provided for, the routine aborting when it encounters an illegal hex character, leaving the accumulator containing the index into the buffer, pointing to the illicit value. In fact, this method is used to complete the execution of the conversion-rotate loop, using a RETURN character placed at the end of the

ASCII hex string.

The test routine (lines 590 to 800) prompts for four hex-based characters to be input. These are placed in the buffer (line 610) and printed to the VDU. On completion of the input, the address-binary routine is called, and the result placed in the first two bytes of the user area, for printing or manipulation purposes.

Line-by-line

A line-by-line decription of Program 3 follows:

`line 100` : clear indexing register
`line 110` : get byte destination
`line 120` : clear three bytes
`line 150` : entry for NEXT-CHARACTER
`line 160` : get character from buffer
`line 170` : call ASCII-BINARY to convert
`line 180` : branch to ERROR if Carry flag is set
`line 190` : move low nibble into high nibble
`line 210` : save index into buffer
`line 220` : moving four bits
`line 225` : entry for AGAIN
`line 230` : move bit 7 into Carry flag
`line 240` : move carry into bit 0 and bit 7 into Carry flag
`line 250` : move carry into bit 0 and bit 7 into Carry flag
`line 260` : decrement bit count
`line 270` : and do until four bits done
`line 280` : restore index into buffer
`line 290` : increment it to point to next character
`line 300` : do branch to NEXT-CHARACTER
`line 310` : entry for ERROR
`line 320` : get illegal character
`line 330` : return to calling routine

CONVERT DECIMAL ASCII STRING TO BINARY

This routine takes a signed decimal string of ASCII characters and transforms it into a two-byte hexadecimal number. For example, entering −32,678 will return the value $8000, where $8000 is its signed binary equivalent. Entry requirements to the conversion routine are obtained by the BASIC text in lines 880 to 940. Note that in addition to obtaining the characters for insertion into the string buffer, the number of characters for conversion is required, this being placed in the first byte of the buffer.

Program 4

```
 1Ø REM ** DECIMAL ASCII TO BINARY **
 2Ø REM ** READ & POKE M/C DATA **
 3Ø CODE=49152
 4Ø FOR LOOP=Ø TO 155
 5Ø  READ BYTE
 6Ø  POKE CODE+LOOP,BYTE
 7Ø NEXT LOOP
 8Ø :
 9Ø REM ** M/C DATA **
1ØØ :
11Ø DATA  174,6Ø,3        REM  LDX $33C
12Ø DATA  2Ø8,3           REM  BEQ $Ø3
125 DATA  76,154,192 :    REM  JMP $CØ9A
13Ø DATA  16Ø,Ø           REM  LDY #Ø
14Ø DATA  14Ø,55,3        REM  STY $337
15Ø DATA  14Ø,53,3        REM  STY $335
16Ø DATA  15Ø,54,3        REM  STY $336
17Ø DATA  2ØØ             REM  INY
18Ø DATA  14Ø,52,3        REM  STY $334
19Ø DATA  185,6Ø,3        REM  LDA $33C,Y
2ØØ DATA  2Ø1,45          REM  CMP #$2D
21Ø DATA  2Ø8,14          REM  BNE $ØE
22Ø DATA  169,255         REM  LDA #&FF
23Ø DATA  141,55,3        REM  STA $337
24Ø DATA  238,52,3        REM  INC $334
25Ø DATA  2Ø2             REM  DEX
26Ø DATA  24Ø,113         REM  BEQ $71
27Ø DATA  76,54,192       REM  JMP $CØ36
28Ø ::                    REM POSITIVE
29Ø DATA  2Ø1,43          REM  CMP #$2B
3ØØ DATA  2Ø8,12          REM  BNE $Ø6
31Ø DATA  238,52,3        REM  INC $334
32Ø DATA  2Ø2             REM  DEX
33Ø DATA  24Ø,1ØØ         REM  BEQ $64
```

```
340 ::                    REM CONVERT-CHARACTER
350 DATA  172,52,3        REM  LDY $334
360 DATA  185,60,3        REM  LDA $33C,Y
370 ::                    REM CHECK-LEGALITY
380 DATA  201,58          REM  CMP #$3A
390 DATA  16,90           REM  BPL $5A
400 DATA  201,48          REM  CMP #$30
410 DATA  48,86           REM  BMI $56
420 DATA  72              REM  PHA
430 DATA  14,53,3         REM  ASL $335
440 DATA  46,54,3         REM  ROL $336
450 DATA  173,53,3        REM  LDA $335
460 DATA  172,53,3        REM  LDY $336
470 DATA  14,53,3         REM  ASL $335
480 DATA  46,54,3         REM  ROL $336
490 DATA  14,53,3         REM  ASL $335
500 DATA  46,54,3         REM  ROL $336
510 DATA  24              REM  CLC
520 DATA  109,53,3        REM  ADC $335
530 DATA  141,53,3        REM  STA $335
540 DATA  152             REM  TYA
550 DATA  109,54,3        REM  ADC $336
560 DATA  141,54,3        REM  STA $336
570 DATA  56              REM  SEC
580 DATA  104             REM  PLA
590 DATA  233,48          REM  SBC #$30
600 DATA  24              REM  CLC
610 DATA  109,53,3        REM  ADC $335
620 DATA  141,53,3        REM  STA $335
630 DATA  144,3           REM  BCC $03
640 DATA  238,54,3        REM  INC $336
650 ::                    REM NO-CARRY
660 DATA  238,52,3        REM  INC $334
670 DATA  202             REM  DEX
680 DATA  208,181         REM  BNE $B5
690 DATA  173,55,3        REM  LDA $337
700 DATA  16,17           REM  BPL $11
710 DATA  56              REM  SEC
720 DATA  169,0           REM  LDA #0
730 DATA  237,53,3        REM  SBC $335
```

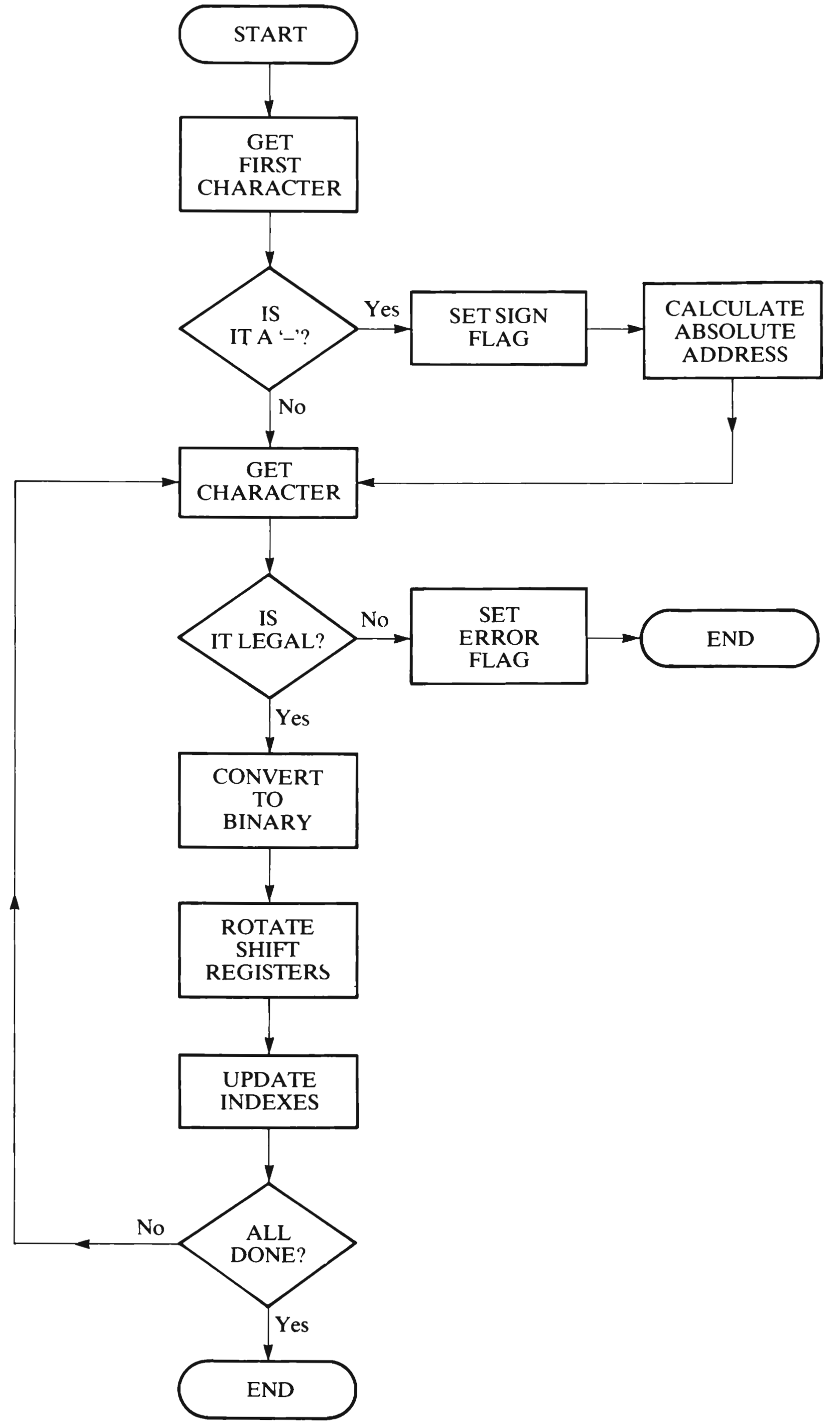


Figure 3.4 ASCII string to binary conversion flowchart

```
740 DATA  141,53,3         REM  STA $335
750 DATA  169,0            REM  LDA #0
760 DATA  237,54,3         REM  SBC $336
770 DATA  141,54,3         REM  STA $336
780 ::                     REM NO-COMPLEMENT
790 DATA  24               REM  CLC
800 DATA  144,1            REM  BCC $1
810 ::                     REM ERROR
820 DATA  56               REM  SEC
830 ::                     REM FINISH
840 DATA  96               REM  RTS
850 :
860 REM ** SET UP SCREEN AND GET NUMBER **
870 PRINT CHR$(147)
880 INPUT"NUMBER FOR CONVERSION";A$
890 FOR LOOP=1 TO LEN(A$)
900  TEMP$=MID$(A$,LOOP,1)
910  B=ASC(TEMP$)
920  POKE 828+LOOP,B
930 NEXT LOOP
940 POKE 828,LEN(A$)
950 :
960 SYS CODE
970 :
980 PRINT"THE TWO BYTES ARE AS FOLLOWS"
990 PRINT"LOW  BYTE ";PEEK(821)
1000 PRINT"HIGH BYTE ";PEEK(822)
```

Bytes are designated as follows:

820	($334)	:	string index
821	($335)		current count
823	($336)	:	sign flag
828	($33C)		length of string
829	($33D)	:	start of character string

The machine code begins by obtaining the character count from the X register. An error is signalled if this count is zero, otherwise the program progresses, clearing the sign flag (used to signal positive or negative values) and result destination bytes at 'current' (lines 130 to 160). Location $70 is used to hold the string index, pointing to the next character for conversion. This byte is initially loaded with 1 so that it skips over the count byte in the buffer.

The first byte of the string is tested for a '+' or '−' sign, the former being an optional item in the string, and the sign flag is set accordingly (lines 190 to 230). The CONVERT-CHARACTER loop starts by testing the character about to be manipulated to ensure it is a decimal value, i.e. 0 to 9 inclusive. Converting the byte into binary form is achieved by multiplying the byte by 10. This multiplication is readily available using four arithmetic shifts and an addition: 2 * 2 * 2 + 2 = 10.

Because we are dealing with a two-byte result, the arithmetic shift must be performed on the two bytes, allowing bits to be transferred from one byte to the other. This is performed by using an ASL followed by a ROL. As figure 3.5 illustrates, this acts exactly like a 16-bit ASL. The first pass through this character-conversion loop has little effect, as it is operating on characters already converted, of which there are none first time round!

Lines 570 to 620 carry out the conversion of ASCII to binary and store the result. This is performed, as we know from earlier examples, by masking off the high nibble. Another technique for doing this is simply to subtract the ASCII code for '0': $30.

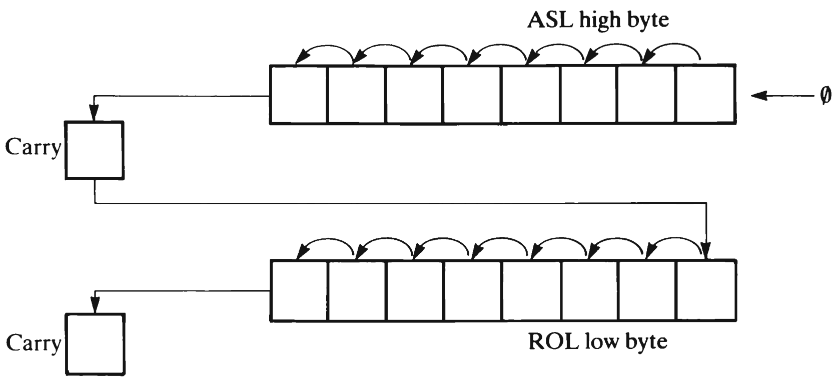


Figure 3.5 A16-bit arithmetic shift

Once all the characters have been processed, the sign flag at $334 (820) is checked for a negative value. If this is indicated (lines 690 and 700), the value of current is subtracted from zero, thereby converting the absolute value into a signed negative byte (lines 710 to 770). The Carry flag is used to indicate any error conditions—if it is set an error occurred, and the string index at $334 points to the illegal character.

Line-by-line

A line-by-line description of Program 4 now follows:

line 11Ø : get length of string
line 12Ø : branch if not zero
line 125 : else jump to ERROR
line 13Ø : clear Y register
line 14Ø : sign flag
line 15Ø : and store bytes
line 17Ø : increment Y
line 18Ø : set index to first ASCII character
line 19Ø : get first character
line 2ØØ : is it a minus sign?
line 21Ø : no, branch to POSITIVE
line 22Ø : yes, get negative byte
line 23Ø : and set the sign flag
line 24Ø : move to next character
line 25Ø : decrement length counter
line 26Ø : branch to ERROR if zero
line 27Ø : else jump to CONVERT-CHARACTER
line 28Ø : entry for POSITIVE
line 29Ø : is first character a +?
line 3ØØ : no, branch to CHECK-LEGALITY
line 31Ø : yes, move to next character
line 32Ø : decrement length counter
line 33Ø : branch to ERROR if zero
line 34Ø : entry for CONVERT-CHARACTER
line 35Ø : restore index
line 36Ø : get character from buffer
line 37Ø : entry for CHECK-LEGALITY
line 38Ø : is it <= ASC"9"?
line 39Ø : no, it's bigger, branch to ERROR
line 4ØØ : is it >= ASC"Ø"?
line 41Ø : no, branch to ERROR
line 42Ø : save code on stack
line 43Ø : multiply both bytes by two
line 45Ø : save low byte

line 46Ø : save high byte
line 47Ø : multiply by two again (now *4)
line 49Ø : and again (now *8)
line 51Ø : clear Carry flag
line 52Ø : add low byte *2
line 53Ø : and save result
line 54Ø : transfer high byte *2
line 55Ø : and add to *8 high byte
line 56Ø : save it. Now *1Ø
line 57Ø : set Carry flag
line 58Ø : restore ASCII code from stack
line 59Ø : convert ASCII to binary
line 6ØØ : clear Carry flag
line 61Ø : add it to low byte current
line 62Ø : save result
line 63Ø : branch if NO-CARRY
line 64Ø : else increment high byte
line 65Ø : entry for NO-CARRY
line 66Ø : move index on to next byte
line 67Ø : decrement length counter
line 68Ø : branch to CONVERT-CHARACTER if not finished
line 69Ø : completed so get sign flag
line 7ØØ : if clear branch to NO-COMPLEMENT
line 71Ø : else set Carry flag
line 72Ø : clear accumulator
line 73Ø : and obtain two's complement
line 74Ø : save low byte result
line 75Ø : clear accumulator
line 76Ø : subtract high byte from Ø
line 77Ø : and save result
line 78Ø : entry for NO-COMPLEMENT
line 79Ø : clear Carry flag
line 8ØØ : and force branch to FINISH
line 81Ø : entry for ERROR
line 82Ø : set Carry flag to denote error
line 83Ø : entry for FINISH
line 84Ø : return to BASIC

4 Binary to Hex ASCII

This chapter complements the previous one and illustrates how memory-based hex values can be converted into their ASCII representation. The routines provide the following conversions:

1. Print accumulator as two ASCII hex characters.
2. Print two hex bytes as four ASCII hex characters.
3. Print two-byte signed binary number as signed decimal number.

PRINT ACCUMULATOR

To convert an eight-bit binary number into its ASCII hex equivalent characters, the procedure described in Chapter 3 must be reversed. However, because text is printed on the screen from left to right, we must deal with the high nibble of the byte first. Program 5 uses the hexprint routine to print the hexadecimal value of any key pressed at the keyboard.

Program 5

```
10 REM ** PRINT ACCUMULATOR AS A HEX NUMBER **
20 :
30 CODE=49152
40 FOR LOOP=0 TO 21
50  READ BYTE
60  POKE CODE+LOOP,BYTE
70 NEXT LOOP
80 :
90 REM ** M/C DATA **
100 :
```

```
11Ø DATA 72                 REM  PHA
12Ø DATA 74, 74             REM  ASL  A : ASL A
13Ø DATA 74, 74             REM  ASL  A : ASL A
14Ø DATA 32,9,192           REM  JSR $CØØ9
15Ø DATA 1Ø4                REM  PLA
16Ø ::                      REM FIRST $CØØ9
17Ø DATA 41,15              REM  AND #$ØF
18Ø DATA 2Ø1,1Ø             REM  CMP #$ØA
19Ø DATA 144,Ø2             REM  BCC $Ø2
2ØØ DATA 1Ø5,6              REM  ADC #$Ø6
21Ø ::                      REM OVER
22Ø DATA 1Ø5,48             REM  ADC #$3Ø
23Ø DATA 76,21Ø,255         REM  JMP $FFD2
24Ø :
25Ø REM ** SET UP DEMO AT 828 **
26Ø REM LDA $FB  : JMP $CØØØ
27Ø POKE 828,165 : POKE 829,251
28Ø POKE 83Ø,76  : POKE 831,Ø   : POKE 832,192
29Ø PRINT CHR$(147)
3ØØ PRINT "HIT ANY KEY AND ITS HEX VALUE IN"
31Ø PRINT "ASCII WILL BE DISPLAYED"
32Ø GET A$: IF A$="" THEN GOTO 32Ø
33Ø A=ASC(A$)
34Ø POKE 251,A
35Ø :
36Ø SYS 828
37Ø REM CALL 'SYS CODE' TO USE DIRECTLY
```

The hexprint routine is embedded between lines 11Ø and 23Ø. The accumulator's contents are first pushed on to the hardware stack. This procedure is necessary as it will have to be restored before the second pass, which calculates the ASCII code for the second character. The first pass through the routine sets about moving the upper nibble of the accumulator byte into the lower nibble (lines 12Ø and 13Ø). The FIRST subroutine ensures that the high nibble is cleared by logically ANDing it with $ØF. This is, of course, surplus to requirement on the first pass, but is needed on the second pass to isolate the low nibble. Comparing the accumulator's contents with 1Ø will ascertain whether the value is in the range Ø to 9 or A to F. If the Carry flag is clear, it falls in the lower range (Ø to 9) and simply setting bits 4 and 5, by adding $3Ø, will give the required ASCII code. A further 7 must be added to skip non-hex ASCII codes to arrive at the ASCII codes for A to F ($41 to $46). You may have noticed that line 200 does not add 7 but in fact adds one less, 6. This is because, for this section of coding to be executed, the carry must have been set, and the 6510 addition opcode references the Carry flag in addition. Therefore, the addition performed is: accumulator + 6 + 1.

The JMP of line 230 will return the program back to line 150. Remember, FIRST was called with a JSR, so the RTS from completion of the CHROUT call returns control here. The accumulator is restored and the process repeated for the second ASCII digit.

A short test routine is established in lines 250 to 340. This requests you to hit a key, the value of which is placed in a free zero page byte. The 'hand-POKEd' routine at 828 is called by line 360, and puts the key's value into the accumulator before performing a jump to the main routine.

The following example illustrates the program's operation, assuming the accumulator holds the value 01001111, $4F:

Mnemonic	Accumulator	Carry flag
	$4F	
LSR A	$27	1
LSR A	$13	1
LSR A	$09	1
LSR A	$04	1
JSR FIRST		
AND #$0F	$04	1
CMP #$0A	$04	0
BCC OVER		
OVER		
ADC #$30	$34 (ASC"4")	0
JMP CHROUT		
PLA	$4F	0
AND #$0F	$0F	0
CMP #$0A		

Line-by-line

A line-by-line description of Program 5 follows:

`line 110` : save accumulator on stack
`line 120` : move high nibble into low nibble

line 140 : call FIRST subroutine
line 150 : restore accumulator
line 160 : entry for FIRST
line 170 : ensure only low nibble set
line 180 : is it < 10?
line 190 : yes, branch to OVER
line 200 : no, add 7, value $A to $F
line 210 : entry for OVER
line 220 : add 48 to convert to ASCII code
line 230 : and print, returning to line 140 or BASIC

PRINT A HEXADECIMAL ADDRESS

The hexprint routine can be extended to enable two zero page bytes to be printed out in hexadecimal form. This is an especially important procedure when writing machine based utilities, such as a hex dump or disassembler. The revamped program is listed below:

Program 6

```
 10 REM ** PRINT TWO HEX BYTES AS **
 20 REM ** A TWO-BYTE ADDRESS **
 30 CODE=49152
 40 FOR LOOP=0 TO 34
 50  READ BYTE
 60  POKE CODE+LOOP,BYTE
 70 NEXT LOOP
 80 :
 90 REM ** M/C DATA **
100 REM ** CALL WITH $FB,$FC HOLDING BYTES **
110 ::                      REM ADDRESS-PRINT
120 DATA  162,251       : REM  LDX #$FB
130 DATA  181,1         : REM  LDA $01,X
140 DATA  32,13,192     : REM  JSR $C00D
150 DATA  181,0         : REM  LDA $00,X
160 DATA  32,13,192     : REM  JSR $C00D
170 DATA  96            : REM  RTS
180 ::                      REM HEXPRINT
190 DATA  72            : REM  PHA
200 DATA  74,74         : REM  LSR A : LSR A
```

```
210 DATA  74,74          : REM  LSR A : LSR A
220 DATA  32,22,192      : REM  JSR  $C016
230 DATA  104            : REM  PLA
240 ::                     REM FIRST
250 DATA  41,15          : REM  AND #$0F
260 DATA  201,10         : REM  CMP #$0A
270 DATA  144,2          : REM  BCC $02
280 DATA  105,6          : REM  ADC #$06
290 ::                     REM OVER
300 DATA  105,48         : REM  ADC #$30
310 DATA  76,210,255 : REM  JMP $FFD2
```

Zero paged indexed addressing is used to access the two bytes, the crucial location being given in the X register, which acts as the index for the high byte, LDA $01,X (line130), and the low byte, LDA $00,X (line 150). The all-important address in this instance is $FB (line 130), so the bytes accessed by ADDRESS-PRINT are $FB ($FB+0) and $FC ($FB+1). Using this method, various addresses can be housed within zero page and any one reached simply by seeding the X register with the location value.

Project

Adapt Program 6 to accept a five character decimal number from the keyboard, printing its hexadecimal value on the screen. Remember—no BASIC, and the input routine must be able to accept numbers in the range 0 to 65!

BINARY SIGNED NUMBER TO SIGNED ASCII DECIMAL STRING

This conversion utility takes a two-byte hexadecimal number and converts it into its equivalent decimal based ASCII character string. For example, if the two-byte value is $7FFF, the decimal string is 32,767, $7FFF being 32,767 in decimal. The coding uses signed binary values so that if the most significant bit is set, a negative value is interpreted. This is relayed in the string with a minus sign. This means that the routine can handle values in the range 32,767 to −32,768. When using the routine, remember that the two's complement representation is used, so that a hex value of $FFFF is converted to the string −1, and $8000 returns the character string −32,767.

The two address bytes are located at $334 and $335 and the string buffer from $FB onwards. The length of the string buffer will vary, but its maximum length will not exceed six digits, so this number of bytes should be reserved.

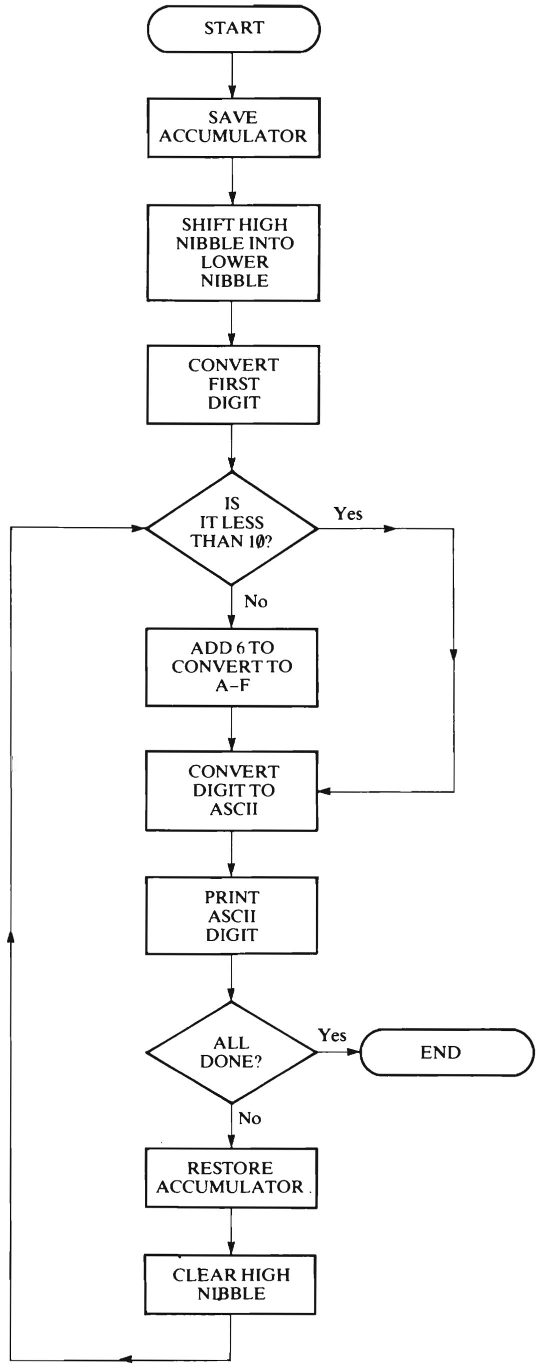


Figure 4.1 Hex to ASCII conversion flowchart

Program 7

```
1Ø REM ** BINARY SIGNED NUMBER CONVERSION **
2Ø REM ** INTO SIGNED DECIMAL ASCII STRING **
3Ø CODE=49152 : OUTPUT=493Ø1
4Ø FOR LOOP=Ø TO 163
5Ø  READ BYTE
6Ø  POKE CODE+LOOP,BYTE
7Ø NEXT LOOP
8Ø :
9Ø REM ** M/C DATA **
1ØØ DATA  16Ø,Ø        : REM  LDY #$ØØ
11Ø DATA  152          : REM  TYA
12Ø DATA  133,251      : REM  STA $FB
13Ø DATA  133,252      : REM  STA $FC
14Ø DATA  133,253      : REM  STA $FD
15Ø DATA  133,254      : REM  STA $FE
16Ø DATA  133,255      : REM  STA $FF
17Ø DATA  173,53,3     : REM  LDA $335
18Ø DATA  141,56,3     : REM  STA $338
19Ø DATA  16,15        : REM  BPL $ØF
2ØØ DATA  56           : REM  SEC
21Ø DATA  152          : REM  TYA
22Ø DATA  237,52,3     : REM  SBC $334
23Ø DATA  141,52,3       REM  STA $334
24Ø DATA  152          : REM  TYA
25Ø DATA  237,53,3     : REM  SBC $335
26Ø DATA  141,53,3     : REM  STA $335
27Ø ::                   REM CONVERSION
28Ø DATA  169,Ø        : REM  LDA #$ØØ
29Ø DATA  141,54,3     : REM  STA $336
3ØØ DATA  141,55,3     : REM  STA $337
31Ø DATA  24           : REM  CLC
32Ø DATA  162,16       : REM  LDX #$1Ø
33Ø ::                   REM LOOP
34Ø DATA  46,52,3      : REM  ROL $334
35Ø DATA  46,53,3      : REM  ROL $335
36Ø DATA  46,54,3      : REM  ROL $336
37Ø DATA  46,55,3      : REM  ROL $337
38Ø DATA  56           : REM  SEC
```

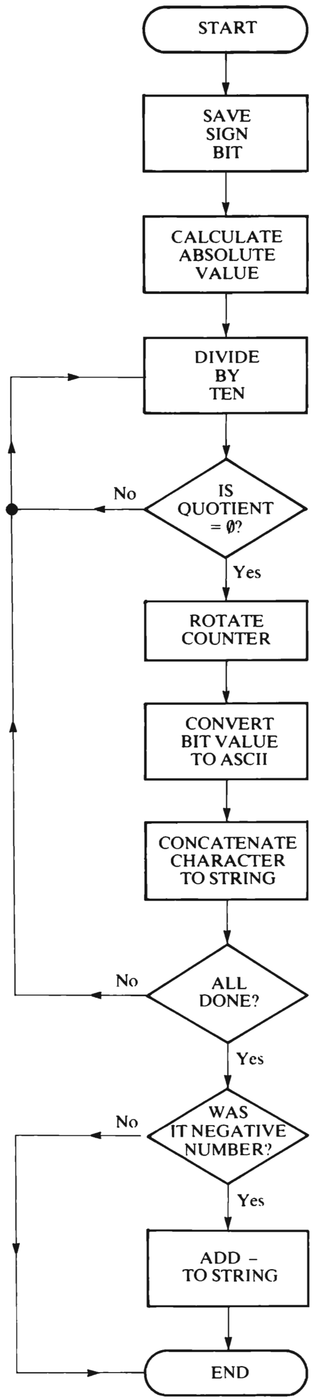


Figure 4.2 Binary to ASCII string conversion flowchart

```
390 DATA  173,54,3     : REM  LDA $336
400 DATA  233,10       : REM  SBC #$0A
410 DATA  168          : REM  TAY
420 DATA  173,55,3     : REM  LDA $337
430 DATA  233,0        : REM  SBC #$00
440 DATA  144,6        : REM  BCC $06
450 DATA  140,54,3     : REM  STY $336
460 DATA  141,55,3     : REM  STA $337
470 ::                   REM LESS-THAN
480 DATA  202          : REM  DEX
490 DATA  208,221      : REM  BNE $DD
500 DATA  46,52,3      : REM  ROL $334
510 DATA  46,53,3      : REM  ROL $335
520 ::                   REM ADD-ASCII
530 DATA  24           : REM  CLC
540 DATA  173,54,3     : REM  LDA $336
550 DATA  105,48       : REM  ADC #$30
560 DATA  32,116,192   : REM  JSR $C074
570 DATA  173,52,3     : REM  LDA $334
580 DATA  13,53,3      : REM  ORA $335
590 DATA  208,187      : REM  BNE $BB
600 ::                   REM FINISHED
610 DATA  173,56,3     : REM  LDA $338
620 DATA  16,5         : REM  BPL $05
630 DATA  169,45       : REM  LDA #$2D
640 DATA  32,116,192   : REM  JSR $C074
650 ::                   REM POSITIVE
660 DATA  96           : REM  RTS
670 REM  SUBROUTINE TO FORM ASCII CHARACTER
    STRING IN $FB
680 ::                   REM CONCATENATE
690 DATA  72           : REM  PHA
700 DATA  160,0        : REM  LDY #$00
710 DATA  185,251,0    : REM  LDA $00FB,Y
720 DATA  168          : REM  TAY
730 DATA  240,11       : REM  BEQ $0B
740 ::                   REM SHUFFLE-ALONG
750 DATA  185,251,0    : REM  LDA $00FB,Y
760 DATA  200          : REM  INY
770 DATA  153,251,0    : REM  STA $00FB,Y
```

```
780 DATA  136,136      : REM  DEY : DEY
790 DATA  208,245      : REM  BNE $F5
800 ::                   REM ZERO-FINISH
810 DATA  104          : REM  PLA
820 DATA  160,1        : REM  LDY #$01
830 DATA  153,251,0    : REM  STA $00FB,Y
840 DATA  136          : REM  DEY
850 DATA  182,251      : REM  LDX $FB,Y
860 DATA  232          : REM  INX
870 DATA  150,251      : REM  STX $FB,Y
880 DATA  96           : REM  RTS
890 REM STRING PRINTING ROUTINE
900 ::                   REM STRING-PRINT
910 DATA  166,251      : REM  LDX $FB
920 DATA  160,1        : REM  LDY #$01
930 ::                   REM PRINT-LOOP
940 DATA  185,251,0    : REM  LDA $FB,Y
950 DATA  32,210,255   : REM  JSR $FFD2
960 DATA  200          : REM  INY
970 DATA  202          : REM  DEX
980 DATA  208,246      : REM  BNE $F6
990 DATA  96           : REM  RTS
1000 :
1010 REM ** GET IN A HEX NUMBER **
1020 PRINT CHR$(147) : PRINT
1030 PRINT"INPUT A HEX NUMBER :$";
1040 GOSUB 2000
1050 POKE 820,LOW      : REM LOW BYTE HEX
     NUMBER
1060 GOSUB 2000
1070 POKE 821,HIGH     : REM HIGH BYTE HEX
     NUMBER
1080 :
1090 SYS CODE          : REM CALL CONVERSION
1100 :
1110 PRINT"ITS DECIMAL EQUIVALENT IS :";
1120 SYS OUTPUT
1130 END
1140 :
1999 REM ** HEX INPUT CONTROL **
```

```
2000 GOSUB 2500
2010 F=NUM : PRINT Z$;
2020 GOSUB 2500
2030 S=NUM : PRINT Z$;
2040 HIGH=F*16+S
2050 GOSUB 2500
2060 F=NUM : PRINT Z$;
2070 GOSUB 2500
2080 S=NUM : PRINT Z$
2090 LOW=F*16+S
2100 RETURN
2200 :
2499 REM ** GET HEX ROUTINE **
2500 GET Z$
2510 IF Z$="" THEN GOTO 2500
2520 IF Z$>"F" THEN GOTO 2500
2530 IF Z$="A" THEN NUM=10: RETURN
2540 IF Z$="B" THEN NUM=11: RETURN
2550 IF Z$="C" THEN NUM=12: RETURN
2560 IF Z$="D" THEN NUM=13: RETURN
2570 IF Z$="E" THEN NUM=14: RETURN
2580 IF Z$="F" THEN NUM=15: RETURN
2590 NUM=VAL(Z$): RETURN
```

Functional bytes:

251-255	($FB-$FF)	:	ASCII string buffer
820-821	($334-$335)	:	binary address for conversion
822-823	($336-$337)	:	temporary storage
824	($338)	:	sign flag

To demonstrate the routine's workings, the program first prompts for a hexadecimal number using the BASIC hex loader subroutine at line 2000. This is evaluated and placed at BINARY-ADDRESS by lines 1050 and 1070.

The program proper begins by clearing the string buffer area (lines 100 to 160), an important procedure which ensures no illicit characters find their way into the ASCII string. The sign of the number is tested by loading the high byte of the address byte into the accumulator and saving its value in the sign flag byte. This process will condition the Negative flag. If it is set, a negative number is interpreted and the plus branch to CONVERSION (line

19Ø) fails. The next seven operations obtain the absolute value of the two-byte number by subtracting it from itself and the set carry bit. Thus $FFFF will result in an absolute value of 1 and $8ØØØ an absolute value of 32,678.

The two flows of the program rejoin at line 28Ø, where the two temporary bytes are cleared. These bytes are used in conjunction with the binary address bytes to form a 32-bit shift register, allowing bits to flow from the low byte address to the high byte of temporary.

The loop of lines 34Ø to 51Ø performs the conversion, by successively dividing through by ten until the quotient has a value of zero. By this time the binary equivalent of this ASCII character being processed will have been placed in the temporary byte. To produce this, the loop needs sixteen iterations so the X register is used to count these out. Converting the binary to hex involves simply adding $3Ø or ASC"Ø" to it (lines 53Ø to 55Ø).

Because it may not be immediately clear what is happening, Table 4.1 shows the values of the accumulator and four associated bytes after each of the 16 passes of the loop, when converting $FFFF into its absolute ASCII value of 1. It should be clear from this how the bits shuffle their way through the four byte 'register'.

Table 4.1

Iteration	Accumulator	$334	$335	$336	$337
1	ØØ	Ø1	ØØ	ØØ	ØØ
2	FF	Ø2	ØØ	ØØ	ØØ
3	FF	Ø4	ØØ	ØØ	ØØ
4	FF	Ø8	ØØ	ØØ	ØØ
5	FF	1Ø	ØØ	ØØ	ØØ
6	FF	2Ø	ØØ	ØØ	ØØ
7	FF	4Ø	ØØ	ØØ	ØØ
8	FF	8Ø	ØØ	ØØ	ØØ
9	FF	ØØ	ØØ	Ø1	ØØ
1Ø	FF	ØØ	ØØ	Ø1	ØØ
11	FF	ØØ	ØØ	Ø1	ØØ
12	FF	ØØ	ØØ	Ø1	ØØ
13	FF	ØØ	ØØ	Ø1	ØØ
14	FF	ØØ	ØØ	Ø1	ØØ
15	FF	ØØ	ØØ	Ø1	ØØ
16	FF	ØØ	ØØ	Ø1	ØØ

All that is now required is for this character to be added to the string buffer. This concatenation is completed by the code of lines 69Ø to 88Ø. This began by obtaining the buffer index, which contains the current number of characters already concatenated. This is stored in the first byte of the buffer, $FB in this instance. It is then moved across into the accumulator. Next, lines 75Ø to 79Ø move any characters present in the buffer up memory one byte, thereby opening up a gap of one byte into which the newly formed character can be placed (lines 81Ø to 87Ø). The buffer index is also incremented and restored at this point, before an RTS is made back to the main body of the program.

End of program operation is tested for by logically ORing the contents of the high and low bytes of the address. If the result is zero, all bits have been rotated and dealt with, in which case the sign flag byte is tested to ascertain whether a minus sign need be placed at the start of the ASCII string (lines 6ØØ to 66Ø).

Line-by-line

A line-by-line description of Program 7 follows:

`line 1ØØ`	:	clear Y register
`line 11Ø`	:	and accumulator
`line 12Ø`	:	and then the five buffer bytes
`line 17Ø`	:	get high byte for conversion
`line 18Ø`	:	save in sign flag
`line 19Ø`	:	if positive branch to CONVERSION
`line 2ØØ`	:	else set Carry flag
`line 21Ø`	:	clear accumulator
`line 22Ø`	:	obtain absolute value of low byte
`line 23Ø`	:	and save
`line 24Ø`	:	clear accumulator
`line 25Ø`	:	obtain absolute value of high byte
`line 26Ø`	:	and save
`line 27Ø`	:	entry for CONVERSION
`line 28Ø`	:	clear accumulator
`line 29Ø`	:	clear temporary storage bytes
`line 31Ø`	:	clear Carry flag
`line 32Ø`	:	sixteen bits to process
`line 33Ø`	:	entry for LOOP
`line 34Ø`	:	move bit 7 into Carry flag
`line 35Ø`	:	and on into bit Ø
`line 36Ø`	:	move bit 7 into Carry flag

line 37Ø : and on into bit Ø
line 38Ø : set Carry flag
line 39Ø : get low byte of temp
line 4ØØ : subtract 1Ø
line 41Ø : save result in Y
line 42Ø : get high byte of temporary
line 43Ø : subtract carry bit
line 44Ø : branch to LESS-THAN if divisor>dividend
line 45Ø : else save result of operation in temporary
line 47Ø : entry for LESS-THAN
line 48Ø : decrement bit count
line 49Ø : branch to LOOP until 16 bits done
line 5ØØ : rotate bit 7 into Carry flag
line 51Ø : and on into bit Ø
line 52Ø : entry for ADD-ASCII
line 53Ø : clear Carry flag
line 54Ø : get low byte from temporary
line 55Ø : convert into ASCII character
line 56Ø : concatenate on to string in buffer
line 57Ø : get low byte of binary number
line 58Ø : OR with high byte. If Ø then all done
line 59Ø : if not finished branch to CONVERSION
line 6ØØ : entry for FINISHED
line 61Ø : get sign
line 62Ø : if N = Ø branch to POSITIVE
line 63Ø : otherwise get ASC"−"
line 64Ø : and add it to final string
line 65Ø : entry for POSITIVE
line 66Ø : back to BASIC
line 68Ø : entry for CONCATENATE, $CØ74
line 69Ø : save accumulator
line 7ØØ : initialize index
line 71Ø : and get buffer length
line 72Ø : move it into Y for indexing
line 73Ø : if Ø branch to ZERO-LENGTH
line 74Ø : entry for SHUFFLE-ALONG
line 75Ø : get character from buffer
line 76Ø : increment index
line 77Ø : save character one byte along
line 78Ø : restore original address minus one

line 79Ø : branch to SHUFFLE-ALONG until completed
line 8ØØ : entry for ZERO-FINISH
line 81Ø : restore accumulator
line 82Ø : index past length byte
line 83Ø : add character to buffer
line 84Ø : decrement index
line 85Ø : get length byte
line 86Ø : increment it
line 87Ø : save it
line 88Ø : back to calling routine
line 9ØØ : entry for OUTPUT
line 91Ø : get length of string as counter
line 92Ø : set index to first character
line 93Ø : entry for PRINT-LOOP
line 94Ø : get character
line 95Ø : print it
line 96Ø : increment index
line 97Ø : decrement count
line 98Ø : branch to PRINT-LOOP until all done
line 99Ø : back to BASIC

110 INPUT"WHICH DIRECTION

5 String Manipulation

In this chapter we will look at how ASCII character strings can be manipulated using machine code routines to perform the following operations:

1. Compare two strings.
2. Concatenate one string onto another.
3. Copy a substring from within a main string.
4. Insert a substring into a main string.

These types of routines are essential if you intend to write any programs that manipulate data and information. Adventure games are a typical example of this kind of program.

COMPARING STRINGS

String comparison is normally performed after the computer user has input some information from the keyboard. In BASIC this might be written as:

```
1ØØ A$="MOVE LEFT"
11Ø INPUT"WHICH DIRECTION ?"; B$
12Ø IF A$=B$ THEN PRINT "CORRECT!"
```

We do not always wish to test for equality, however. In BASIC, we are able to test for unlike items using the NOT operators ‘<>’. Thus, line 12Ø could have been written as:

```
12Ø IF A$ <> B$ PRINT "WRONG!"
```

At other times, we may wish to test which of two strings has a greater length, and this is possible in BASIC using the LEN statement:

```
210 IF LEN(A$) > LEN(B$) THEN PRINT "FIRST"
```

Program 8 gives the assembler and BASIC listing for the string comparison routine, which puts all the functions described above at your disposal whenever the program is used. The Status register holds these answers in the Zero and Carry flags. The Zero flag is used to signal equality: if it is set (Z=1), the two strings compared were identical; if it is cleared (Z=0) they were dissimilar.

The Carry flag returns information as to which of the two strings was the longer: if it is set (C=1), they were identical in length or the first string was the larger. The actual indication required here is evaluated in conjunction with the Zero flag. If Z=0 and C=1, then a longer string rather than an equal-length string is indicated, but if the Carry flag is returned clear (C=0), then the second string was longer than the first.

Program 8

```
10 REM ** STRING COMPARISON ROUTINE **
20 CODE=49152
30 TEST=49184
40 FOR LOOP=0 TO 41
50  READ BYTE
60  POKE CODE+LOOP,BYTE
70 NEXT LOOP
80 :
90 REM ** M/C DATA **
100 DATA  173,52,3   : REM  LDA $334
110 DATA  205,53,3   : REM  CMP $335
120 DATA  144,3      : REM  BCC $03
130 DATA  174,53,3   : REM  LDX $335
140 ::                 REM COMPARE-STRING
150 DATA  240,12     : REM  BEQ $0C
160 DATA  160,0      : REM  LDY #$00
170 ::                 REM COMPARE-BYTES
180 DATA  177,251    : REM  LDA ($FB),Y
190 DATA  209,253    : REM  CMP ($FD),Y
200 DATA  208,10     : REM  BNE $0A
210 DATA  200        : REM  INY
220 DATA  202        : REM  DEX
230 DATA  208,246    : REM  BNE $F6
240 ::                 REM CONDITION-FLAGS
250 DATA  173,52,3   : REM  LDA $334
```

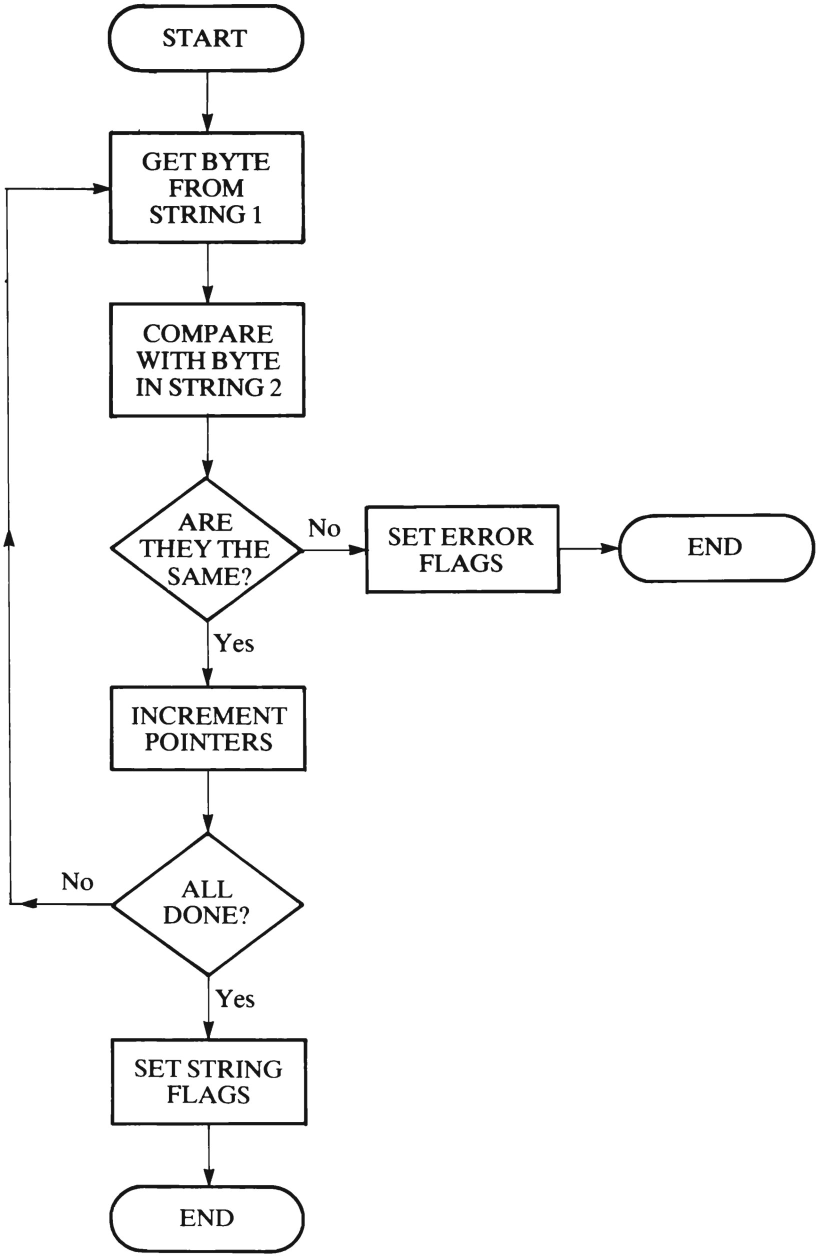


Figure 5.1 Compare strings flowchart

```
260 DATA  205,53,3   : REM  CMP $335
270 ::                 REM FINISH
280 DATA  96         : REM  RTS
290 :
300 ::                 REM TEST ROUTINE
310 DATA  32,0,192   : REM  JSR $C000
```

```
320 DATA  8          : REM  PHP
330 DATA  104        : REM  PLA
340 DATA  41,3         REM  AND #$03
350 DATA  133,251    : REM  STA $FB
360 DATA  96         : REM  RTS
370:
380 REM ** SET UP STRINGS FOR COMPARISON **
390 PRINT CHR$(147)
400 INPUT "FIRST  STRING :";A$
410 FOR LOOP=1 TO LEN(A$)
420 TEMP$=MID$(A$,LOOP,1)
430 A=ASC(TEMP$)
440 POKE 50432+LOOP-1,A
450 NEXT LOOP
460 :
470 INPUT "SECOND STRING :";B$
480 FOR LOOP=1 TO LEN(B$)
490 TEMP$=MID$(B$,LOOP,1)
500 B=ASC(TEMP$)
510 POKE 50688+LOOP-1,B
520 NEXT LOOP
530 :
540 POKE 251,0 : POKE 252,197
550 POKE 253,0 : POKE 254,198
560 POKE 820,LEN(A$)   POKE 821,LEN(B$)
570 :
580 SYS TEST
590 :
600 PRINT "RESULT IS : ";PEEK(251)
```

Bytes reserved:

251–252	($FB–$FC)	:	address of first string
253–254	($FD–$FE)	:	address of second string
820	($334)	:	length of first string
821	($335)	:	length of second string

Once run, the BASIC text of lines 380 to 520 calls for two strings to be input. These are stored in memory from $C500 and $C600. Note that the routine cannot handle strings greater than 256 characters in length (though it could of course be expanded to do so). The length of each string is also required by the routine, so this is ascertained and stored in the appropriate zero page bytes at $334 and $335 (line 56Ø).

To allow the string buffers to be fully relocatable, the string addresses are held in two zero page vectors (lines 54Ø and 55Ø).

String comparison proper starts by evaluating the length bytes to find out if they are the same length. If they are not equal, then the strings cannot be identical. However, as the routine returns information about the lengths of the strings it is still completed—in this case the program compares bytes through the length of the smaller of the two strings.

Byte comparison is performed by lines 17Ø to 19Ø, using post-indexed indirect addressing. On the first non-equal characters the main loop is exited to FINISH. Assuming the entire comparison works, and the X register, which holds the working string length, has been decremented to zero, the length bytes (lines 25Ø and 26Ø) are compared to condition the Zero and Carry flags before the routine completes.

The short test routine returns the Zero and Carry flag values and prints them out, indicating the following results:

Returned	Z	C	Result
Ø	Ø	Ø	Strings <> and string 1 larger
1	Ø	1	Strings <> and string 2 larger
3	1	1	Strings =

Line-by-line

A line-by-line description of Program 8 follows:

line 1ØØ : get length of first string
line 11Ø : is it the same length as the second string?
line 12Ø : no, it's longer, so branch to COMPARE-STRING
line 13Ø : yes, so get length of second string
line 14Ø : entry for COMPARE-STRING
line 15Ø : if zero, branch to CONDITION-FLAGS
line 16Ø : initialize indexing register
line 17Ø : entry for COMPARE-BYTES
line 18Ø : get character from first string
line 19Ø : compare to same character in second string
line 2ØØ : if dissimilar, branch to FINISH
line 21Ø : increment index

line 22Ø : decrement string counter
line 23Ø : branch back to COMPARE-BYTES until zero
line 24Ø : entry for CONDITION-FLAG
line 25Ø : get length of first string
line 26Ø : compare with length of the second string
line 27Ø : entry for finish
line 28Ø : back to calling routine
line 3ØØ : entry for TEST routine
line 31Ø : push status onto stack
line 32Ø : pull into accumulator
line 33Ø : save Z and C
line 34Ø : save at location $FB
line 35Ø : back to BASIC

STRINGS UNITE

Strings may be joined together by a process called 'concatenation'. In BASIC the addition operator '+' performs this function. Thus the program:

```
1ØØ A$="REM"
11Ø B$="ARK"
12Ø C$=A$+B$
```

assigns the string 'REMARK' to the string C$. If line 12Ø were rewritten as:

```
12Ø C$=B$+A$
```

the resultant value assigned to C$ would be 'ARKREM'. We can see from this that one string is simply tagged on to the end of the other, overwriting the former's RETURN character, but preserving the latter's.

This process of concatenation can be performed quite readily as Program 9 illustrates. However, the actual BASIC equivalent of the operation we are performing here is:

```
A$=A$+B$
```

In other words, we are adding the second string on to the first string, rather than summing the two to give a separate final string, although this is possible with slight modifications to the assembler text.

Program 9

```
10 REM ** STRING CONCATENATION **
20 CODE=49152
30 FOR LOOP=0 TO 96
40  READ BYTE
50  POKE CODE+LOOP,BYTE
60 NEXT LOOP
70 :
80 REM ** M/C DATA **
90 ::                      REM STRING-CONCATENATION
100 DATA  173,52,3    : REM  LDA $334
110 DATA  141,54,3    : REM  STA $336
120 DATA  169,0       : REM  LDA #$00
130 DATA  141,55,3    : REM  STA $337
140 DATA  24          : REM  CLC
150 DATA  173,53,3    : REM  LDA $335
160 DATA  109,52,3    : REM  ADC $334
170 DATA  176,3       : REM  BCS $03
180 DATA  76,45,192   : REM  JMP $C02D
190 ::                  REM TOO-LONG
200 DATA  169,255     : REM  LDA #$FF
210 DATA  141,57,3    : REM  STA $339
220 DATA  56          : REM  SEC
230 DATA  237,52,3    : REM  SBC $334
240 DATA  144,51      : REM  BCC $33
250 DATA  141,56,3    : REM  STA $338
260 DATA  169,255     : REM  LDA #$FF
270 DATA  141,52,3    : REM  STA $334
280 DATA  76,59,192   : REM  JMP $C03B
290 ::                  REM GOOD-LENGTH
300 DATA  141,52,3    : REM  STA $334
310 DATA  169,0       : REM  LDA #$00
320 DATA  141,57,3    : REM  STA $339
330 DATA  173,53,3    : REM  LDA $335
340 DATA  141,56,3    : REM  STA $338
350 ::                  REM CONCATENATION
360 DATA  173,56,3    : REM  LDA $338
370 DATA  240,21      : REM  BEQ $15
380 ::                  REM LOOP
```

```
390 DATA  172,55,3    : REM  LDY $337
400 DATA  177,253     : REM  LDA ($FD),Y
410 DATA  172,54,3    : REM  LDY $336
420 DATA  145,251     : REM  STA ($FB),Y
430 DATA  238,54,3    : REM  INC $336
440 DATA  238,55,3    : REM  INC $337
450 DATA  206,56,3    : REM  DEC $338
460 DATA  208,235     : REM  BNE $EB
470 ::                  REM FINISHED
480 DATA  172,52,3    : REM  LDY $334
490 DATA  169,13      : REM  LDA #$0D
500 DATA  145,251     : REM  STA ($FB),Y
510 DATA  173,57,3    : REM  LDA $339
520 DATA  106         : REM  ROR A
530 DATA  96          : REM  RTS
540 :
600 PRINT CHR$(147)
610 INPUT "FIRST  STRING ";A$
620 INPUT "SECOND STRING ";B$
630 :
640 F=49664           : REM $C200
650 S=49920           : REM $C300
660 :
670 FOR LOOP=1 TO LEN(A$)
680  TEMP$=MID$(A$,LOOP,1)
690  A=ASC(TEMP$)
700  POKE F+LOOP-1,A
710 NEXT LOOP
720 :
730 FOR LOOP=1 TO LEN(B$)
740  TEMP$=MID$(B$,LOOP,1)
750  B=ASC(TEMP$)
760  POKE S+LOOP-1,B
770 NEXT LOOP
780 :
790 POKE 251,0    POKE 252,194
800 POKE 253,0  : POKE 254,195
810 POKE 820,LEN(A$)
```

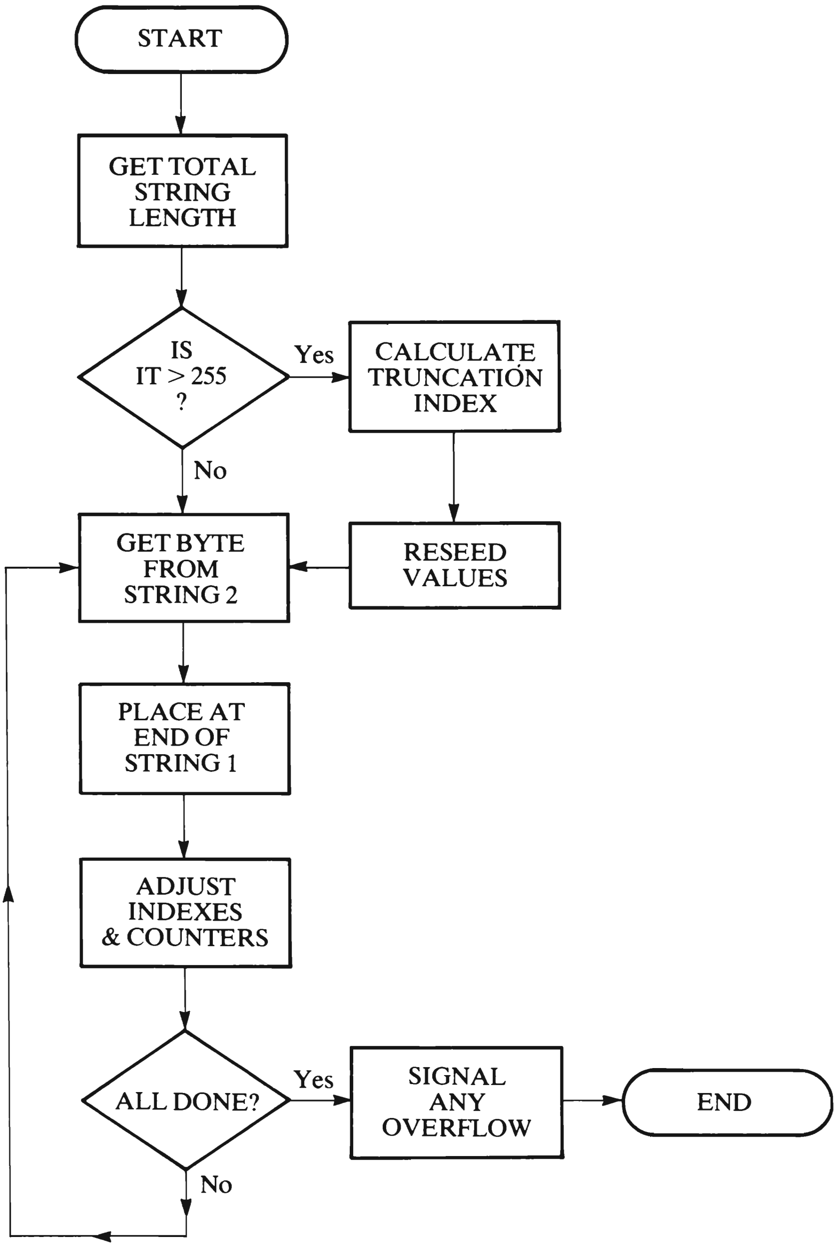


Figure 5.2 Concatenate strings flowchart

```
820 POKE 821,LEN(B$)
830 :
840 SYS CODE
850 :
```

```
860 REM *** PRINT OUT FINAL STRING ***
870 PRINT "FINAL STRING IS :";
880 LOOP=0
890 REM ** REPEAT **
900 BYTE=PEEK(F+LOOP)
910 PRINT CHR$(BYTE);
920 LOOP=LOOP+1
930 IF BYTE=13 THEN END
940 GOTO 900
```

This program allows a final string of 256 characters in length to be manipulated. Therefore, as the program stands, the combined lengths of the two strings should not exceed this length. If they do, then only as many characters as space allows will be concatenated on to the first string, leaving the second string truncated. The Carry flag is used to signal whether any truncation has taken place, being set if it has and cleared otherwise. As with the string comparison routine, the string buffers are accessed via two zero page vectors (lines 790 and 800) and two bytes are reserved to hold the length of each string. A further two bytes are used to save index values.

The first nine machine code operations (lines 100 to 180) determine the final length of the string, by adding the length of the first string to that of the second string. A sum greater than 256 is signalled in the Carry flag and the branch of line 170 is performed, in which case the number of characters which can be inserted into the first string buffer is ascertained. The overflow indicator is loaded with $FF if a truncation occurs; otherwise it is cleared with $00.

The concatenating loop is held between lines 350 and 460. This simply moves a byte from the vectored address plus the index of the second string and places it at the end of the first string, as pointed to by the first string index byte. This process is reiterated until the value of 'count' has reached zero. Lines 480 and 500 place a RETURN character at the end of the string to facilitate printing from BASIC or machine code. The Overflow flag is loaded into the accumulator and bit 7 rotated across into the Carry flag, thereby signalling whether truncation has occurred. Lines 610 to 770 hold the BASIC test routine that reads in and then pokes the character strings into memory at $C200 and $C300. After the SYS call (line 840), the final BASIC routine prints the concatenated string from memory.

Project

Adapt the program to perform the BASIC equivalent of C$=A$+B$ or C$=B$+A$ on request.

Line-by-line

A line-by-line description of Program 9 now follows:

line 1ØØ	:	get first string's length
line 11Ø	:	string one's index
line 12Ø	:	clear accumulator
line 13Ø	:	set string two's index to zero
line 14Ø	:	clear Carry flag
line 15Ø	:	get second string's length
line 16Ø	:	and add to length of first string
line 17Ø	:	branch to TOO-LONG if total greater than 256 bytes
line 18Ø	:	otherwise jump to GOOD-LENGTH
line 19Ø	:	entry for TOO-LONG
line 2ØØ	:	load accumulator with 255
line 21Ø	:	and store to indicate overflow
line 22Ø	:	set Carry flag and subtract
line 23Ø	:	string one's length from maximum length
line 24Ø	:	branch to FINISH if first string is greater than 256 bytes in length
line 25Ø	:	save current count
line 26Ø	:	restore maximum length
line 27Ø	:	store in string one's length
line 28Ø	:	jump to concatenation routine
line 29Ø	:	entry for GOOD-LENGTH
line 3ØØ	:	save accumulator in string one's length
line 31Ø	:	load with Ø to clear
line 32Ø	:	overflow indicator
line 33Ø	:	get string two's length
line 34Ø	:	save in count
line 35Ø	:	entry for CONCATENATION
line 36Ø	:	get count value
line 37Ø	:	if zero, then finish
line 38Ø	:	entry for LOOP
line 39Ø	:	get index for string two
line 4ØØ	:	and get character from second string
line 41Ø	:	get string one's index
line 42Ø	:	and place character into first string
line 43Ø	:	increment first string's index
line 44Ø	:	increment second string's index
line 45Ø	:	decrement count

line 46Ø : branch to LOOP until count=Ø
line 47Ø : entry for FINISHED
line 48Ø : get final length of first string
line 49Ø : load accumulator with ASCII return
line 5ØØ : place at end of string
line 51Ø : get overflow indicator
line 52Ø : and move it into Carry flag
line 53Ø : back to calling routine

COPY-CAT

String manipulation routines must include a method of copying substrings of characters from anywhere within a string of characters. In BASIC, three such commands are provided. They are MID$, LEFT$ and RIGHT$, although with the first of these, any point in a string can be accessed. The following shows the sort of thing possible in BASIC:

```
1ØØ A$="CONCATENATE"
11Ø B$=MID$(A$,Ø,3)
12Ø PRINT B$
```

Running this will output the string 'CON'. What the code has done is to take the three characters from the first character in the Main$. Program 10 produces the same type of operation from machine code.

Program 10

```
 1Ø REM ** COPY A SUBSTRING FROM WITHIN **
 2Ø REM ** A MAIN ASCII STRING **
 3Ø CODE=49152
 4Ø MAIN=5Ø432        : REM $C5ØØ
 5Ø SUB=5Ø688         : REM $C6ØØ
 6Ø REM ** READ AND POKE M/C DATA **
 7Ø FOR LOOP=Ø TO 123
 8Ø  READ BYTE
 9Ø  POKE CODE+LOOP,BYTE
1ØØ NEXT LOOP
11Ø :
12Ø REM ** M/C DATA **
13Ø DATA  16Ø,Ø       : REM  LDY #$ØØ
14Ø DATA  14Ø,52,3    : REM  STY $334
```

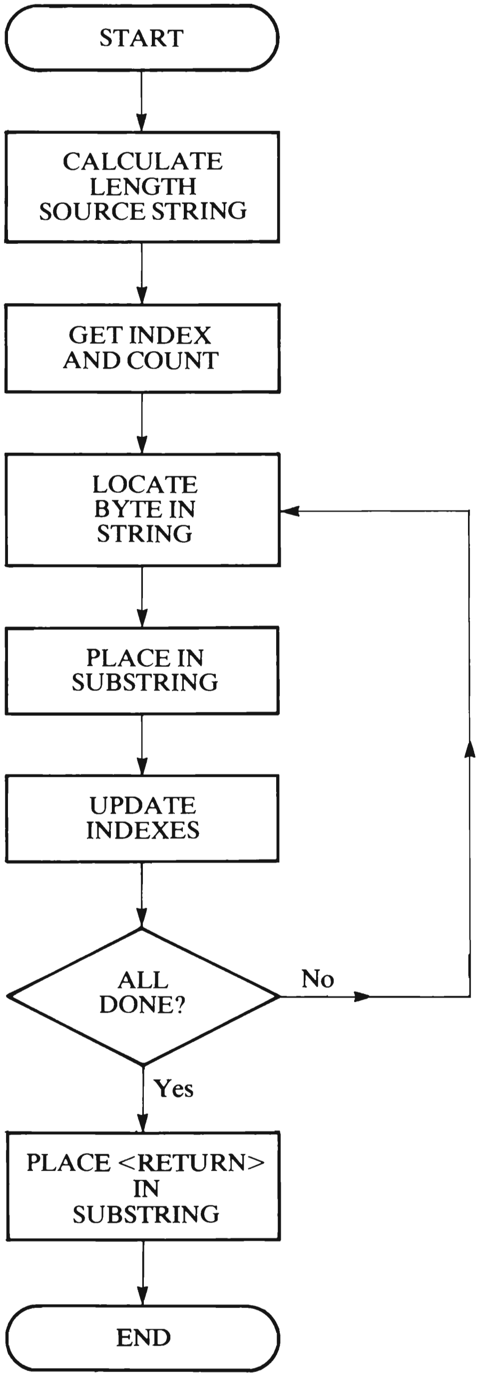


Figure 5.3 Copy string flowchart

```
150 DATA  140,56,3   : REM  STY $338
160 DATA  173,54,3   : REM  LDA $336
170 DATA  240,98     : REM  BEQ $62
```

```
180 DATA  173,53,3   : REM  LDA $335
190 DATA  205,55,3   : REM  CMP $337
200 DATA  144,93     : REM  BCC $5D
210 DATA  24         : REM  CLC
220 DATA  173,55,3   : REM  LDA $337
230 DATA  109,54,3   : REM  ADC $336
240 DATA  176,9      : REM  BCS $09
250 DATA  170        : REM  TAX
260 DATA  202        : REM  DEX
270 DATA  236,53,3   : REM  CPX $335
280 DATA  144,20     : REM  BCC $14
290 DATA  240,18     : REM  BEQ $12
300 ::                 REM TRUNCATION
310 DATA  56         : REM  SEC
320 DATA  173,53,3   : REM  LDA $335
330 DATA  237,55,3   : REM  SBC $337
340 DATA  141,54,3   : REM  STA $336
350 DATA  238,54,3   : REM  INC $336
360 DATA  169,255    : REM  LDA #$FF
370 DATA  141,56,3   : REM  STA $338
380 ::                 REM GREATER-EQUAL
390 DATA  173,54,3   : REM  LDA $336
400 DATA  201,255    : REM  CMP #$FF
410 DATA  144,10     : REM  BCC $0A
420 DATA  240,8      : REM  BEQ $08
430 DATA  169,255    : REM  LDA #$FF
440 DATA  141,54,3   : REM  STA $336
450 DATA  141,56,3   : REM  STA $338
460 ::                 REM COPY-SUBSTRING
470 DATA  174,54,3   : REM  LDX $336
480 DATA  240,35     : REM  BEQ $23
490 DATA  169,0      : REM  LDA #$00
500 DATA  141,52,3   : REM  STA $334
510 ::                 REM LOOP
520 DATA  172,55,3   : REM  LDY $337
530 DATA  177,251    : REM  LDA ($FB),Y
540 DATA  172,52,3   : REM  LDY $334
550 DATA  145,253    : REM  STA ($FD),Y
560 DATA  238,55,3   : REM  INC $337
570 DATA  238,52,3   : REM  INC $334
```

```
580 DATA  202         : REM  DEX
590 DATA  208,237     : REM  BNE $ED
600 DATA  206,52,3    : REM  DEC $334
610 DATA  173,56,3    : REM  LDA $338
620 DATA  208,3       : REM  BNE $03
630 ::                  REM FINISH
640 DATA  24          : REM  CLC
650 DATA  144,1       : REM  BCC $01
655 ::                  REM ERROR
660 DATA  56          : REM  SEC
670 ::                  REM OUT
680 DATA  169,13      : REM  LDA #$0D
690 DATA  172,52,3    : REM  LDY $334
700 DATA  200         : REM  INY
710 DATA  145,253     : REM  STA ($FD),Y
720 DATA  96          : REM  RTS
730 :
740 REM ** SET UP MAIN STRING **
750 PRINT CHR$(147)
760 ::                  REM ERROR
770 INPUT "MAIN STRING ";B$
780 FOR LOOP=1 TO LEN(B$)
790  TEMP$=MID$(B$,LOOP,1)
800  B=ASC(TEMP$)
810  POKE MAIN+LOOP-1,B
820 NEXT LOOP
830 :
840 INPUT"INDEX INTO STRING ";X
850 INPUT"NUMBER OF BYTES TO COPY ";Y
860 :
870 REM ** SET UP BYTES FOR M/C **
880 POKE 251,0 : POKE 252,197
                      : REM $C500 VECTOR
890 POKE 253,0 : POKE 254,198
                      : REM $C600 VECTOR
900 POKE 821,LEN(B$)
910 POKE 822,Y
920 POKE 823,X
930 :
940 SYS CODE
```

```
 950 :
 960 REM ** READ COPIED SUBSTRING **
 970  FOR LOOP=1 TO Y
 980  Z=PEEK(SUB+LOOP-1)
 990  PRINT CHR$(Z);
1000 NEXT LOOP
```

Bytes are designated as follows:

251–252	($FB–$FC)	:	main string vector
253–254	($FD–$FE)	:	substring vector
820	($334)	:	length of substring
821	($335)	:	length of main string
822	($336)	:	number of bytes to be copied
823	($337)	:	index into main string
824	($338)	:	error flag

Once again, a few lines of BASIC demonstrate the operation of the routine, requesting the source string, starting index and length of substring, or rather the number of bytes to be copied into the substring from the starting index. The main string is in a buffer located at $C500 and the substring is copied into its own buffer at $C600. As always, these addresses may be changed to suit user needs, as they are vectored through zero page (lines 880 and 890).

Error-checking is allowed, as the Carry flag is set on exit if an error has occurred. Normally, an error will occur only if the starting index is beyond the length of the source string, or the number of bytes to be copied from the main string is zero. If the number of bytes requested in the length exceeds the number left from the indexed position to the end of the main string, then only the bytes available will be copied to the substring buffer.

On entry to the routine, error-checking is performed (lines 160 to 240) and if any are found, the program exits. Lines 300 to 370 perform a truncation if the number of bytes to be copied exceeds those available. The COPY-SUBSTRING loop (lines 460 to 590) copies each string byte from the vectored address in the main string to the substring buffer. Each time a character is copied, the substring length byte is incremented. On completion of this loop, controlled by the X register, the error flag is restored and the Carry flag conditioned accordingly (lines 610 to 660). Finally (lines 690 to 730), an ASCII RETURN character is placed at the end of the substring.

The following example shows the resultant substrings produced from the main string 'CONCATENATE' for different indexes. Figure 5.4 illustrates the index value for each of the main string's characters.

Index	Length	Substring
Ø	3	CON
3	3	CAT
4	3	ATE

String	C	O	N	C	A	T	E	N	A	T	E
Index	Ø	1	2	3	4	5	6	7	8	9	1Ø

Figure 5.4 String Index

Line-by-line

A line-by-line description of Program 10 follows:

`line 13Ø` : initialize Y register
`line 14Ø` : clear substring length
`line 15Ø` : and error flag
`line 16Ø` : get substring length
`line 17Ø` : if a null string, branch to FINISH
`line 18Ø` : get main string's length
`line 19Ø` : compare it with index byte
`line 2ØØ` : branch to ERROR if index is greater
`line 21Ø` : clear the Carry flag
`line 22Ø` : get index
`line 23Ø` : add it to substring length
`line 24Ø` : branch to TRUNCATION if result is greater than 255
`line 25Ø` : move index across into X register
`line 26Ø` : decrement it by one
`line 27Ø` : compare result with string length
`line 28Ø` : branch to GREATER-EQUAL if result is
`line 29Ø` : greater than or equal to string length
`line 3ØØ` : entry for TRUNCATION
`line 31Ø` : set the Carry flag
`line 32Ø` : get string length
`line 33Ø` : subtract the index from it
`line 34Ø` : save the new length
`line 35Ø` : and increment it by one

line 36Ø : denote an error by
line 37Ø : setting the error flag
line 38Ø : entry for GREATER-EQUAL
line 39Ø : get length into accumultor
line 4ØØ : compare with maximum length
line 41Ø : branch if count is
line 42Ø : greater or equal to maximum length
line 43Ø : put maximum length in accumulator
line 44Ø : store in bytes to copy
line 45Ø : and also in error flag
line 46Ø : entry for COPY-SUBSTRING
line 47Ø : get the index position
line 48Ø : branch to ERROR if zero
line 49Ø : clear accumulator
line 5ØØ : and substring length
line 51Ø : entry for LOOP
line 52Ø : get main string index into Y register
line 53Ø : get character from main string
line 54Ø : get substring index
line 55Ø : copy character into substring
line 56Ø : increment main string index
line 57Ø : increment substring index
line 58Ø : decrement bytes to move counter
line 59Ø : branch to LOOP if still bytes to be copied
line 6ØØ : decrement final substring count
line 61Ø : get error flag into accumulator
line 62Ø : branch to ERROR if not zero
line 63Ø : FINISH entry
line 64Ø : clear Carry flag as no error
line 65Ø : branch to OUT
line 655 : entry for ERROR
line 66Ø : set Carry flag to indicate error
line 67Ø : entry for OUT
line 68Ø : place RETURN in accumlator
line 69Ø : get substring index into Y
line 7ØØ : increment Y
line 71Ø : place RETURN at end of substring
line 72Ø : return to BASIC.

INSERTION

This final routine provides the facility for inserting a string within the body of another string, allowing textual material—for example, in word processing applications—to be manipulated. If the main string held 'ELIZABETH OKAY', this routine could be called to insert the string 'RULES', so that the final string would read 'ELIZABETH RULES OKAY'. As with the COPY routine, the position of the insertion is pointed to by an index byte, and the Carry flag is set if an error is detected—that is, if an index of Ø or a null substring is specified.

The maximum length of the final string is 256 characters. If the insertion of the substring would cause this length to be exceeded, the substring is truncated to the length given by (256 minus length of main string) and only these characters are inserted.

As always, a BASIC primer demonstrates the routine's use. The string buffers are held at $C5ØØ and $C6ØØ and in this instance they are accessed directly, although there is no reason why vectored addresses could not be used.

Program 11

```
 1Ø REM ** INSERT ONE ASCII STRING **
 2Ø REM ** INTO ANOTHER ASCII STRING **
 3Ø MAIN=5Ø432          : REM $C5ØØ
 4Ø SUB=5Ø688           : REM $C6ØØ
 5Ø CODE=49152
 6Ø REM ** READ AND POKE DATA **
 7Ø FOR LOOP=Ø TO 141
 8Ø  READ BYTE
 9Ø  POKE LOOP+CODE,BYTE
1ØØ NEXT LOOP
11Ø :
12Ø REM ** M/C DATA **
13Ø DATA  16Ø,Ø        : REM  LDY #Ø
14Ø DATA  14Ø,53,3     : REM  STY $335
15Ø DATA  165,252      : REM  LDA $FC
16Ø DATA  2Ø8,3        : REM  BNE $Ø3
17Ø DATA  76,137,192   : REM  JMP $CØ89
18Ø ::                   REM ZERO-LENGTH
19Ø DATA  165,253      : REM  LDA $FD
2ØØ DATA  24Ø,124      : REM  BEQ $7C
21Ø DATA  ::             REM CHECK
```

```
220 DATA  24           : REM  CLC
230 DATA  165,252      : REM  LDA $FC
240 DATA  101,251      : REM  ADC $FB
250 DATA  176,6        : REM  BCS $06
260 DATA  201,255      : REM  CMP #$FF
270 DATA  240,18       : REM  BEQ $12
280 DATA  144,16       : REM  BCC $10
290 ::                   REM CUT-OFF
300 DATA  169,255      : REM  LDA #$FF
310 DATA  56           : REM  SEC
320 DATA  229,251      : REM  SBC $FB
330 DATA  240,104      : REM  BEQ $68
340 DATA  144,102      : REM  BCC $66
350 DATA  133,252      : REM  STA $FC
360 DATA  169,255      : REM  LDA #$FF
370 DATA  141,53,3     : REM  STA $335
380 ::                   REM CALC-LENGTH
390 DATA  165,251      : REM  LDA $FB
400 DATA  197,253      : REM  CMP $FD
410 DATA  176,20       : REM  BCS $14
420 DATA  166,251      : REM  LDX $FB
430 DATA  232          : REM  INX
440 DATA  134,253      : REM  STX $FD
450 DATA  169,255      : REM  LDA #$FF
460 DATA  141,53,3     : REM  STA $335
470 DATA  24           : REM  CLC
480 DATA  165,251      : REM  LDA $FB
490 DATA  101,252      : REM  ADC $FC
500 DATA  133,251      : REM  STA $FB
510 DATA  76,109,192   : REM  JMP $C06D
520 ::                   REM NO-PROBLEMS
530 DATA  56           : REM  SEC
540 DATA  165,251      : REM  LDA $FB
550 DATA  229,253      : REM  SBC $FD
560 DATA  170          : REM  TAX
570 DATA  232          : REM  INX
580 DATA  165,251      : REM  LDA $FB
590 DATA  133,254      : REM  STA $FE
600 DATA  24           : REM  CLC
610 DATA  101,252      : REM  ADC $FB
```

```
620 DATA  133,251      : REM  STA $FB
630 DATA  141,52,3     : REM  STA $334
640 ::                   REM MAKE-SPACE
650 DATA  164,254      : REM  LDY $FE
660 DATA  185,0,197    : REM  LDA $C500,Y
670 DATA  172,52,3     : REM  LDY $334
680 DATA  153,0,197    : REM  STA $C500,Y
690 DATA  206,52,3     : REM  DEC $334
700 DATA  198,254      : REM  DEC $FE
710 DATA  202          : REM  DEX
720 DATA  208,237      : REM  BNE $ED
730 ::                   REM INSERT-SUBSTRING
740 DATA  169,0        : REM  LDA #$00
750 DATA  133,254      : REM  STA $FE
760 DATA  166,252      : REM  LDX $FC
770 ::                   REM TRANSFER
780 DATA  164,254      : REM  LDY $FE
790 DATA  185,0,198    : REM  LDA $C600,Y
800 DATA  164,253      : REM  LDY $FE
810 DATA  153,0,197    : REM  STA $C500,Y
820 DATA  230,253      : REM  INC $FD
830 DATA  230,254      : REM  INC $FE
840 DATA  202          : REM  DEX
850 DATA  208,239      : REM  BNE $EF
860 DATA  173,53,3     : REM  LDA $335
870 DATA  208,3        : REM  BNE $03
880 ::                   REM GOOD
890 DATA  24           : REM  CLC
900 DATA  144,1        : REM  BCC $01
910 ::                   REM ERROR
920 DATA  56           : REM  SEC
930 ::                   REM FINISH
940 DATA  96           : REM  RTS
950 :
960 REM ** GET MAIN STRING AND STORE AT
    $C500 **
970 PRINT CHR$(147)
980 INPUT"MAIN STRING";B$
990 FOR LOOP=1 TO LEN(B$)
1000  TEMP$=MID$(B$,LOOP,1)
```

```
1010  B=ASC(TEMP$)
1020  POKE MAIN+LOOP-1,B
1030 NEXT LOOP
1040 :
1050 REM ** GET SUBSTRING AND STORE AT $C600 **
1060 INPUT"SUB STRING";C$
1070 FOR LOOP=1 TO LEN(C$)
1080  TEMP$=MID$(C$,LOOP,1)
1090  B=ASC(TEMP$)
1100  POKE SUB+LOOP-1,B
1110 NEXT LOOP
1120 :
1130 REM ** GET INSERTION INDEX **
1140 INPUT"INSERTION INDEX"; X
1150 :
1160 REM ** POKE VALUES INTO ZERO PAGE **
1170 POKE 251,LEN(B$)
1180 POKE 252,LEN(C$)
1190 POKE 253,X
1200 :
1210 SYS CODE
1220 :
1230 REM ** READ FINAL STRING **
1240 COUNT=LEN(B$)+LEN(C$)-1
1250 FOR LOOP=0 TO COUNT
1260  Z=PEEK(MAIN+LOOP)
1270  PRINT CHR$(Z);
1280 NEXT LOOP
```

The program begins by checking the length bytes to ensure that no null strings are present (lines 150 to 200) and then sums the two lengths to obtain the final length. If the addition results in the Carry flag being set (line 250), the total length will exceed 256 bytes and, as a result, the inserted substring will be truncated (lines 310 to 390).

If the insertion index is greater than the length of the string, the substring is actually concatenated on to the end of the main string. This evaluation is performed through lines 400 to 530. Before inserting the substring, all characters to the left of the index must be shuffled up through memory to make space for it. These calculations are carried out in lines 550 to 650, ready for the shuffling process (lines 660 to 740). Inserting the substring now involves simply copying it from its buffer into the space opened up for it (lines 75Ø to 87Ø), the X register being used as the characters-moved counter.

Finally, the error flag is restored and the Carry flag conditioned to signal any errors.

Line-by-line

A line-by-line description of Program 11 follows:

line 13Ø : clear indexing register
line 14Ø : clear error flag
line 15Ø : get substring length
line 16Ø : branch to ZERO-LENGTH if Z=Ø
line 17Ø : otherwise carry on
line 18Ø : entry for ZERO-LENGTH
line 19Ø : get offset
line 2ØØ : branch to ERROR if Z=1
line 21Ø : entry for CHECK
line 22Ø : clear Carry flag
line 23Ø : get substring length
line 24Ø : add it to main string length
line 25Ø : branch to CUT-OFF if greater than 256
line 26Ø : is it maximum length?
line 27Ø : branch to CALC-LENGTH if
line 28Ø : it is equal to or greater than
line 29Ø : entry for CUT-OFF
line 3ØØ : get the maximum length allowed
line 31Ø : set Carry flag
line 32Ø : subtract length of string
line 33Ø : branch to ERROR if
line 34Ø : length is equal to or greater than string
line 35Ø : save characters free
line 36Ø : set error flag
line 38Ø : entry for CALC-LENGTH
line 39Ø : get main string length
line 4ØØ : is offset within string?
line 41Ø : branch to NO-PROBLEMS if it is
line 42Ø : else place substring
line 43Ø : at end of main string
line 44Ø : save X in offset
line 45Ø : and flag the error

line 46Ø : in error flag byte
line 47Ø : clear Carry flag
line 48Ø : get length of string
line 49Ø : calculate total length
line 5ØØ : and save result
line 51Ø : jump to INSERT-SUBSTRING
line 52Ø : entry for NO-PROBLEMS
line 53Ø : set Carry flag
line 54Ø : get length of substring
line 55Ø : subtract offset
line 56Ø : move index into X
line 57Ø : increment index
line 58Ø : get length
line 59Ø : save in source
line 6ØØ : clear Carry flag
line 61Ø : find total length
line 62Ø : save result
line 63Ø : and for index
line 64Ø : entry for MAKE-SPACE
line 65Ø : get source index
line 66Ø : get byte from main
line 67Ø : get offset into string
line 68Ø : move byte along
line 69Ø : decrement both indexes
line 71Ø : decrement counter
line 72Ø : branch to MAKE-SPACE until done
line 73Ø : entry for INSERT-SUBSTRING
line 74Ø : clear accumulator
line 75Ø : and source
line 76Ø : get counter
line 77Ø : entry for TRANSFER
line 78Ø : get index
line 79Ø : get byte from substring
line 8ØØ : get offset into main string
line 81Ø : and place byte in main
line 82Ø : increment both indexes
line 84Ø : do until substring inserted
line 85Ø : branch to TRANSFER
line 86Ø : get error flag
line 87Ø : branch to ERROR

line 88Ø : entry for GOOD
line 89Ø : signal no error
line 9ØØ : branch to FINISH
line 91Ø : entry for ERROR
line 92Ø : denote error
line 93Ø : entry for FINISH
line 94Ø : return to calling routine

6 Printing Print!

Every machine code program sooner or later requires text to be printed on to the screen. In most instances, this is a fairly simple process and often involves merely indexing into an ASCII string table and printing the characters, using one of the Operating System calls, until either a RETURN character or zero byte is encountered. Program 12 uses this method.

Program 12

```
 1Ø REM ** PRINT STRING FROM MEMORY **
 2Ø CODE=49152
 3Ø FOR LOOP=Ø TO 13
 4Ø  READ BYTE
 5Ø  POKE CODE+LOOP,BYTE
 6Ø NEXT LOOP
 7Ø :
 8Ø REM ** M/C DATA **
 9Ø ::                   REM STRING-PRINT
1ØØ DATA  162,Ø        : REM  LDX #$ØØ
11Ø ::                   REM NEXT-CHARACTER
12Ø DATA  189,Ø,197    : REM  LDA $C5ØØ,X
13Ø DATA  32,21Ø,255   : REM  JSR $FFD2
14Ø DATA  232          : REM  INX
15Ø DATA  2Ø1,13       : REM  CMP #$ØD
16Ø DATA  2Ø8,245      : REM  BNE $F5
17Ø DATA  96           : REM  RTS
18Ø :
19Ø REM ** GET STRING TO BE PRINTED **
2ØØ STRING=5Ø432
```

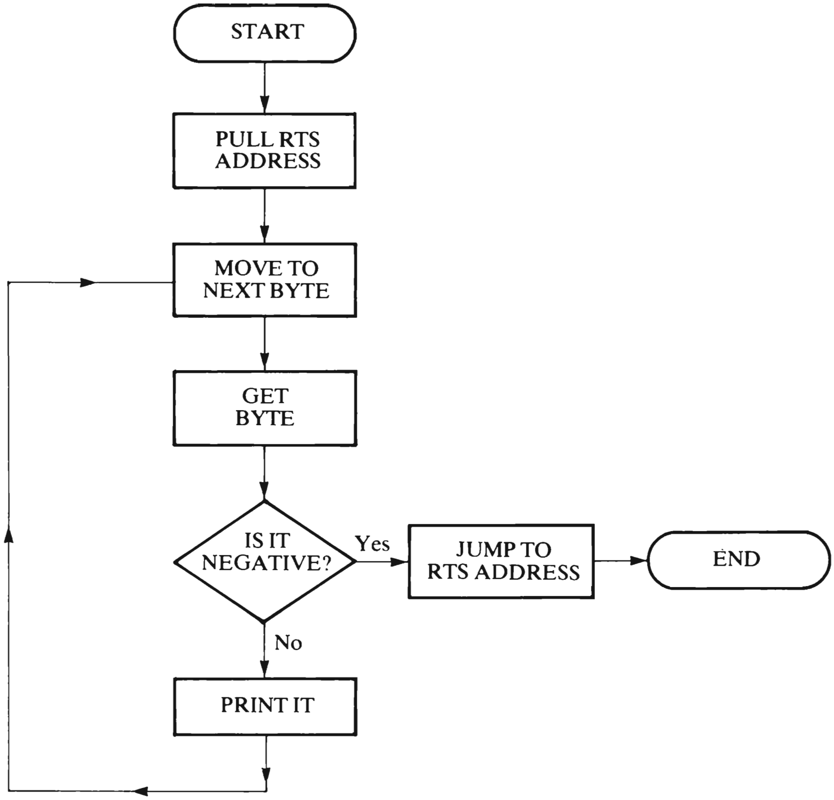


Figure 6.1 Printing embedded code flowchart

```
210 PRINT CHR$(147)
220 INPUT "INPUT STRING :";A$
230 FOR LOOP=1 TO LEN(A$)
240 TEMP$=MID$(A$,LOOP,1)
250 B=ASC(TEMP$)
260 POKE STRING+LOOP-1,B
270 NEXT LOOP
280 PRINT:PRINT
290 PRINT"YOUR STRING WAS AS FOLLOWS :";
300 SYS CODE
```

Here, a string buffer is located at \$C500 (50432) and the requirement for printing the string is that it must be terminated with an ASCII RETURN character, \$0D. The program begins by initializing an index, the X register (line 100), and loading the byte at \$C500+X into the accumulator. This is printed using the Kernal's CHROUT routine, the index is incremented and then the accumulator's contents are compared to see whether the character just output was a RETURN (line 150). If not, the loop branches back and the next character is sought.

Program 13 shows how several strings may be printed to the screen using a loop similar to that described above. The number of strings for printing may be variable, the desired number being passed into the routine via the Y register. The string data has been entered using the DATA statement. If a large amount of string data is to be stored, and the amount to be printed at any one time varied, a vectored address should be used to access the table. Positioning of the text on the screen can be performed by embedding the relative number of RETURNs and spaces into the DATA, or more neatly by using the Kernal's PLOT routine to set the X and Y tab co-ordinates.

Program 13

```
 1Ø REM ** PRINT Y NUMBER OF STRINGS **
 2Ø CODE=49152
 3Ø FOR LOOP=Ø TO 18
 4Ø READ BYTE
 5Ø  POKE CODE+LOOP,BYTE
 6Ø NEXT LOOP
 7Ø :
 8Ø REM ** M/C DATA **
 9Ø DATA  162,Ø         : REM  LDX #$ØØ
1ØØ DATA  16Ø,4         : REM  LDY #$Ø4
11Ø ::                    REM NEXT-CHARACTER
12Ø DATA  189,Ø,197     : REM  LDA $C5ØØ,X
13Ø DATA  32,21Ø,255    : REM  JSR $FFD2
14Ø DATA  232           : REM  INX
15Ø DATA  2Ø1,13        : REM  CMP #$ØD
16Ø DATA  2Ø8,245       : REM  BNE $F5
17Ø DATA  136           : REM  DEY
18Ø DATA  2Ø8,242       : REM  BNE $F2
19Ø DATA  96            : REM  RTS
2ØØ :
21Ø REM ** SET UP FOUR SIMPLE STRINGS **
22Ø STRING=5Ø432
23Ø FOR LOOP=Ø TO 31
24Ø READ BYTE
25Ø POKE STRING+LOOP,BYTE
26Ø NEXT
27Ø :
28Ø REM ** ASCII DATA **
```

```
290 DATA 32,65,65,65,65,65,65,13
300 DATA 32,32,66,66,66,66,66,13
310 DATA 32,32,32,67,67,67,67,13
320 DATA 32,32,32,32,68,68,68,13
```

The final program in this chapter shows the way I find easiest to store and print character strings, stowing them directly within the machine code. The two main advantages of this method are that the string is inserted directly at the point it is needed, avoiding the need to calculate indexes into look-up tables, and that because it manipulates its own address it is fully relocatable.

Program 14

```
 10 REM ** ASCII STRING OUTPUT ROUTINE **
 20 CODE=49152
 30 FOR LOOP=0 TO 26
 40  READ BYTE
 50  POKE CODE+LOOP,BYTE
 60 NEXT LOOP
 70 :
 80 REM ** M/C DATA **
 90 DATA  104          : REM  PLA
100 DATA  133,251      : REM  STA $FB
110 DATA  104          : REM  PLA
120 DATA  133,252      : REM  STA $FC
130 ::                   REM REPEAT
140 DATA  160,0        : REM  LDY #$0
150 DATA  230,251      : REM  INC $FB
160 DATA  208,2        : REM  BNE $02
170 DATA  230,252      : REM  INC $FC
180 ::                   REM OVER
190 DATA  177,251      : REM  LDA ($FB),Y
200 DATA  48,6         : REM  BMI $06
210 DATA  32,210,255   : REM  JSR $FFD2
220 DATA  76,6,192     : REM  JMP $C006
230 ::                   REM FINISH
240 DATA  108,251,0    : REM  JMP ($FB)
250 :
260 REM ** DEMO ROUTINE LOCATED AT $C200 **
270 DEMO=49664
```

```
280 FOR LOOP=0 TO 38
290  READ BYTE
300  POKE DEMO+LOOP,BYTE
310 NEXT LOOP
320 :
330 REM ** DEMO M/C DATA **
340 DATA  169,147       : REM  LDA #$93
350 DATA  32,210,255    : REM  JSR $FFD2
360 DATA  32,0,192      : REM  JSR $C000
370 REM ** NOW STORE ASCII CODES FOR PRINTING **
380 DATA  13            : REM CARRIAGE-RETURN
390 DATA  83,84,82,73,78,71,83,32
                        : REM  STRINGS<SPACE>
400 DATA  87,73,84,72,73,78,32
                        : REM  WITHIN<SPACE>
410 DATA  77,65,67,72,73,78,69,32
                        : REM  MACHINE<SPACE>
420 DATA  67,79,68,69,33
                        : REM  CODE!
430 DATA  234           : REM  NOP
440 DATA  96            : REM  RTS
450 :
460 SYS DEMO
```

The ASCII character string is placed in memory by leaving the machine code assembly (line 360) and POKEing the ASCII codes of the string directly into successive memory locations (lines 380 to 420).

For this routine to work, it is imperative that the first byte following the string is a negative byte—that is, one with bit 7 set. The opcode for NOP, $EA, is ideal for this purpose as it has its most significant bit set ($EA=11101010) and its only effect is to cause a very short delay.

The ASCII print routine is just 27 bytes in length and it should be called as a subroutine immediately before the string is encountered (line 360). On entry into the subroutine, the first four operations pull the return address from the stack and save it in a zero page vector at $FB and $FC. These bytes are then incremented by one to point at the byte following the subroutine call.

Because the string data follows on immediately after the ASCII print subroutine call, post-indexed indirect addressing can be used to load the first string character into the accumulator (line 190). The string terminating negative byte is tested for (line 200), and if not found the byte is printed with a CHROUT call. A JMP to

REPEAT is then performed and the loop reiterated. When the negative byte is encountered, and the branch of line 2ØØ succeeds, an indirect jump (line 24Ø) via the current vectored address is executed, returning control back to the calling machine code at the end of the ASCII string.

Line-by-line

A line-by-line description of Program 14 follows:

line 9Ø : set low byte RTS address
line 1ØØ : save in $FB
line 11Ø : get high byte RTS address
line 12Ø : save in $FC
line 13Ø : entry for REPEAT
line 14Ø : initialize index to zero
line 15Ø : increment low byte of vectored address
line 16Ø : branch to OVER if not zero
line 17Ø : else increment page value
line 18Ø : entry for OVER
line 19Ø : get byte from within program
line 2ØØ : if negative, branch to FINISH
line 21Ø : else print it
line 22Ø : jump to REPEAT
line 23Ø : entry for FINISH
line 24Ø : jump back into main program
line 34Ø : load accumulator with clear screen code
line 35Ø : and print it
line 36Ø : call string printing routine at $CØØØ
line 38Ø : ASCII code for RETURN
line 39Ø : ASCII string 'STRINGS '
line 4ØØ : ASCII string 'WITHIN '
line 41Ø : ASCII string 'MACHINE '
line 42Ø : ASCII string 'CODE!'
line 43Ø : negative byte
line 44Ø : back to BASIC

7 A Bubble of Sorts

Any program written to handle quantities of data will, at some time, require the data in a data table to be sorted into ascending or descending order. Several algorithims are available to facilitate this manipulation of data, of which the bubble sort is perhaps the simplest to implement in BASIC or machine code.

The technique involves moving through the data list and comparing pairs of bytes. If the first byte is smaller than the next byte in the list, the next pair of bytes is sought. If, on the other hand, the second byte is less than the first, the two bytes are swapped. This procedure is repeated until a pass is executed in which no elements are exchanged, so all are in ascending order. Program 15 is the BASIC version of such a bubble sort.

Program 15

```
 1Ø REM ** BASIC BUBBLE SORT **
 2Ø TABLE=828
 3Ø FOR LOOP=Ø TO 19
 4Ø  READ BYTE
 5Ø  POKE TABLE+LOOP,BYTE
 6Ø NEXT LOOP
 7Ø :
 8Ø REM ** BUBBLE-UP ROUTINE **
 9Ø FOR BUBBLE=Ø TO 19
1ØØ TEMP=BUBBLE
11Ø :
12Ø IF PEEK(TABLE+TEMP)>PEEK(TABLE+(TEMP-1))
    THEN GOTO 18Ø
13Ø HOLD=PEEK(TABLE+TEMP)
14Ø POKE TABLE+TEMP,PEEK(TABLE+(TEMP-1))
```

```
150 POKE TABLE+(TEMP-1),HOLD
160 TEMP=TEMP-1
170 IF TEMP<>0 THEN GOTO 120
180 NEXT
190 :
200 REM ** DATA FOR SORTING **
210 DATA 1,255,67,89,120
220 DATA 6,200,85,45,199
230 DATA 0,123,77,98,231
240 DATA 9,234,99,98,100
250 :
260 REM ** PRINT SORTED DATA **
270 FOR LOOP=0 TO 19
280 PRINT PEEK(TABLE+LOOP)
290 NEXT LOOP
```

The data bytes for sorting are held within the four data lines from 210 to 240 and these are read into a memory array called TABLE. The sorting procedure is performed through lines 90 to 180, line 120 checking to see if a swap is required. If a swap is unnecessary, GOTO 180 is executed and the swap routine bypassed. If it is required, however, the GOTO statement is not encounted, and the swap is performed in lines 130 to 160. The byte currently being pointed to is PEEKed into the variable HOLD (line 130) and the next byte is PEEKed and then POKEd into the location immediately before it (line 140). The swap is completed by POKEing the value of HOLD into the now 'vacant' location. The variable TEMP is used to keep track of the number of passes through the loop.

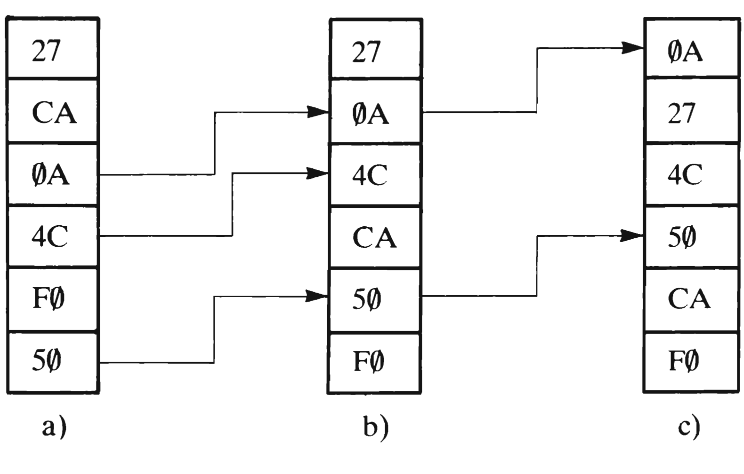


Figure 7.1 Numbers bubbling up

Figure 7.1 illustrates how small numbers bubble up through a data list using this sorting method. In this example, the data list consists of six numbers 27, CA, ØA, 4C, FØ and 5Ø (Figure 7.1a). After the first pass of the bubble sort three swaps have occurred (Figure 7.1b), thus:

1. 27 < CA therefore no change.
2. CA > ØA therefore swap items.
3. CA > 4C therefore swap items.
4. CA < FØ therefore no change.
5. FØ > 5Ø therefore swap items.

The next pass through the data list produces the ordered list of Figure 7.1c in which just two swaps occurred, as follows:

1. 27 > ØA therefore swap items.
2. 27 < 4C therefore no change.
3. 4C < 5Ø therefore no change.
4. CA > 5Ø therefore swap items.
5. CA < FØ therefore no change.

All the data elements are now in their final order, so the next pass through the list will have no effect. We can signal this by using an exchange flag to indicate whether the last pass produced any swaps, the sort routine exiting when the flag is cleared. This detail is included in the BASIC loader listed below as Program 16.

Program 16

```
 1Ø REM *** BUBBLE SORT ***
 2Ø CODE=49152
 3Ø TABLE=5Ø432
 4Ø FOR LOOP=Ø TO 44
 5Ø  READ BYTE
 6Ø  POKE CODE+LOOP,BYTE
 7Ø NEXT LOOP
 8Ø :
 9Ø REM ** M/C DATA **
1ØØ DATA 2Ø6,52,3      : REM  DEC $334
11Ø ::                   REM BUBBLE-LOOP
12Ø DATA 16Ø,Ø         : REM  LDY #$ØØ
13Ø DATA 14Ø,53,3      : REM  STY $335
14Ø DATA 174,52,3      : REM  LDX $334
15Ø ::                   REM LOOP
```

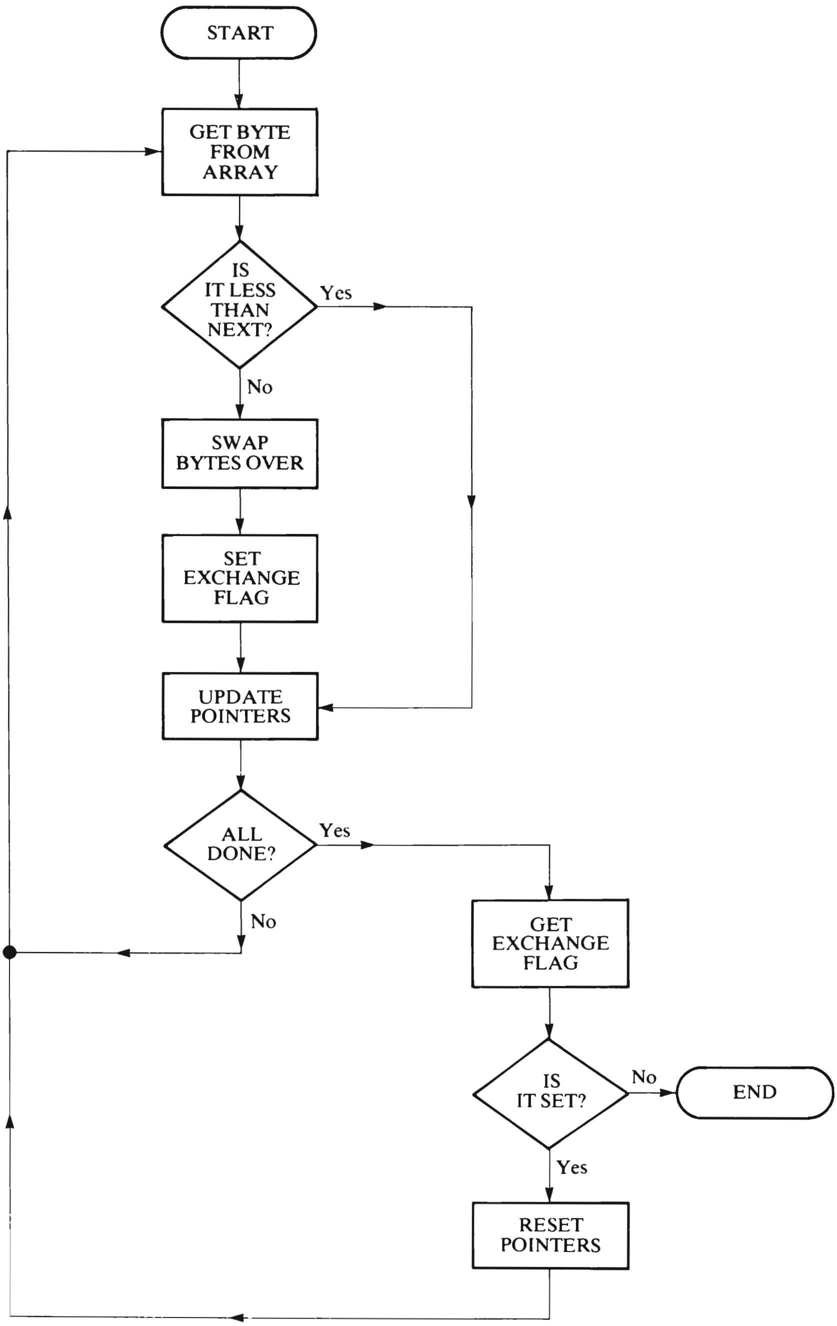


Figure 7.2   Bubble sort flowchart

```
16Ø DATA 177,253       : REM  LDA ($FD),Y
17Ø DATA 2Ø9,251       : REM  CMP ($FB),Y
18Ø DATA 176,13        : REM  BCS $ØD
19Ø DATA 72            : REM  PHA
2ØØ DATA 177,251       : REM  LDA ($FB),Y
21Ø DATA 145,253       : REM  STA ($FD),Y
22Ø DATA 1Ø4           : REM  PLA
23Ø DATA 145,251       : REM  STA ($FB),Y
24Ø DATA 169,1         : REM  LDA #$Ø1
25Ø DATA 141,53,3      : REM  STA $335
26Ø ::                   REM SECOND-FIRST
27Ø DATA 2ØØ           : REM  INY
28Ø DATA 2Ø2           : REM  DEX
29Ø DATA 2Ø8,233       : REM  BNE $E9
3ØØ DATA 173,53,3      : REM  LDA $335
31Ø DATA 24Ø,5         : REM  BEQ $Ø5
32Ø DATA 2Ø6,52,3      : REM  DEC $334
33Ø DATA 2Ø8,215       : REM  BNE $D7
335 ::                   REM FINISH
34Ø DATA 96            : REM  RTS
35Ø :
36Ø REM ** SET UP VECTORS **
37Ø REM $FB=$C5ØØ, $FD=$C5Ø1
38Ø POKE 251,Ø : POKE 252,197
39Ø POKE 253,1 : POKE 253,197
4ØØ :
41Ø REM ** SET UP SCREEN AND ARRAY **
42Ø PRINT CHR$(147)
43Ø PRINT "**** MACHINE CODE BUBBLE SORT ****"
44Ø PRINT:PRINT
45Ø INPUT"NUMBER OF ELEMENTS IN ARRAY ";N
46Ø POKE 82Ø,N         : REM LENGTH OF ARRAY
                         AT $334
47Ø FOR LOOP=Ø TO N-1
48Ø  PRINT"INPUT ELEMENT ";LOOP+1;
49Ø  INPUT A
5ØØ  POKE TABLE+LOOP,A
51Ø NEXT LOOP
52Ø :
53Ø REM ** CALL CODE THEN PRINT SORTED TABLE **
```

```
540 SYS CODE
550 PRINT"SORTED VALUES ARE AS FOLLOWS"
560 FOR LOOP=0 TO N-1
570  PRINT PEEK(TABLE+LOOP)
580 NEXT LOOP
```

After POKEing the machine code data into memory at $C000, two zero page vectors are created to hold the address of the TABLE and TABLE+1 (lines 370 to 390). The program then requests (in BASIC!) the number of elements in the array, which should be a series of integer values less than 256. These are then POKEd into memory (lines 450 to 510). The machine code begins by decrementing the length of array byte by one (line 100), because the last element in the array will have no element beyond it to swap with. The swap flag is then cleared (line 130) and the main loop entered using the X register to count the iterations.

The LOOP begins by loading the data byte into the accumulator (line 160) and comparing it with the one immediately preceding it. If the byte+1 is greater than the byte, the Carry flag will be set and no swap required, in which case the branch to SECOND-FIRST is executed (line 180).

If a swap is required, the second byte is saved, pushing it on to the hardware stack. The first byte is then transferred to the second byte's position (lines 200 and 210) and the accumulator is restored from the stack and transferred to the position of the first byte (lines 220 to 230). To denote that a swap has occured, the swap flag is set (lines 240 and 250). The index and counters are then adjusted (lines 270 and 280) and the loop continues until all the array elements have been compared. Upon completion of a full pass through the array, the swap flag is checked. If it is clear, no exchanges took place during the last pass, so the data list is now ordered and the sort finished (line 300 and 310). If the flag is set, the length of array byte is decremented and the procedure repeated once more (lines 320 and 330). On return from the SYS call, the now ordered list is printed out to the screen.

Line-by-line

A line-by-line description of Program 16 now follows:

line 100 : subtract one from the length of the array
line 110 : entry for BUBBLE-LOOP
line 120 : initialize indexing register
line 130 : clear the swap flag
line 140 : get the array size into the X register to act as a loop counter

line 15Ø : entry for LOOP
line 16Ø : get the byte at the byte+1 position
line 17Ø : compare it with the previous byte
line 18Ø : branch to SECOND-FIRST if the second byte (byte+1) is larger than the first (byte)
line 19Ø : save accumulator on hardware stack
line 2ØØ : get first byte at 'byte' position
line 21Ø : place in current location (byte+1)
line 22Ø : restore accumulator
line 23Ø : and complete swap of bytes
line 24Ø : load accumulator with 1
line 25Ø : and set the swap flag to denote that a swap has been performed
line 26Ø : entry for SECOND-FIRST
line 27Ø : move index on to next byte
line 28Ø : decrement loop counter
line 29Ø : branch to LOOP until done
line 3ØØ : get the swap flag into the accumulator
line 31Ø : if clear, branch to FINISH
line 32Ø : decrement outer counter
line 33Ø : branch to BUBBLE-LOOP until all done
line 335 : entry to FINISH
line 34Ø : back to calling routine

Projects

Rewrite the BASIC sections of the program to make it a complete machine code routine.

Adapt the sorting routine to handle 16-bit numbers.

8 Software Stack

One of the criticisms of the 6510 processor is that it has a very limited set of operation instructions—only 56, though addressing modes extend this to 152 functions. With some thought, however, it is possible to implement operations present on other processors, such as the Z80 or 6809, and build up a set of very useful subroutines which can ultimately be strung together to perform quite sophisticated operations, as well as making the conversion of programs written for other processors much easier.

The routine described below mimics an instruction in the 6809 instruction set which allows the contents of up to eight registers to be pushed on to a stack in memory. This stack is often known as the user stack. I said 'up to eight registers', because the ones to be pushed can be selected, this being determined by the bit pattern of the byte after the user stack subroutine call. But more of that in a moment. First, which registers are we going to push? Obviously all the processor registers: the Program Counter, Status register, accumulator, and Index registers. The three remaining ones, we will implement as three two-byte 'psuedo-registers' from the user area of zero page. These are:

PR1 : $8Ø and $81

PR2 : $82 and $83

PR3 : $84 and $84

This now enables us to save the contents of these locations when required.

As already stated, the byte after the user stack subroutine call determines by its bit pattern which registers are to be pushed, as follows:

bit Ø : pseudo-register 1

bit 1 : pseudo-register 2

bit 2 : pseudo-register 3

```
bit 3 : Y register
bit 4 : X register
bit 5 : accumulator
bit 6 : Status register
bit 7 : Program Counter
```

The rule here is that if the bit is set, the related register is pushed. Thus the instructions:

```
JSR USER-STACK
.BYTE $FF
```

would push all registers on to the user stack, the embedded byte being $FF or 11111111. Alternatively, the coding:

```
JSR USER-STACK
.BYTE $1E
```

where $1E = ØØØ1111Ø would push only the accumulator, Status and Index registers. Perhaps at this point a question is running through your mind: 'won't the embedded byte cause my program to crash?'. That's true on face value, but what we do is get the user stack coding to move the Program Counter on one byte, to pass over it, as Program 17 shows:

Program 17

```
 1Ø REM ** USER STACK **
 2Ø CODE=49152
 3Ø FOR LOOP=Ø TO 116
 4Ø  READ BYTE
 5Ø  POKE CODE+LOOP,BYTE
 6Ø NEXT LOOP
 7Ø :
 8Ø REM ** M/C DATA **
 9Ø DATA  8           : REM  PHP
1ØØ DATA  72          : REM  PHA
11Ø DATA  138,72      : REM  TXA : PHA
12Ø DATA  152,72      : REM  TYA : PHA
13Ø DATA  186         : REM  TSX
14Ø DATA  16Ø,6       : REM  LDY #$Ø6
15Ø ::                  REM PUSH-ZERO-PAGE
16Ø DATA  185,138,Ø   : REM  LDA $ØØ8A,Y
```

```
170 DATA  72          : REM  PHA
180 DATA  136         : REM  DEY
190 DATA  208,249     : REM  BNE $F9
200 DATA  254,5,1     : REM  INC $105,X
210 DATA  189,5,1     : REM  LDA $105,X
220 DATA  133,139     : REM  STA $8B
230 DATA  208,3       : REM  BNE $03
240 DATA  254,6,1     : REM  INC $106,X
250 ::                  REM PC-LOW
260 DATA  189,6,1     : REM  LDA $106,X
270 DATA  133,140     : REM  STA $8C
280 DATA  169,135     : REM  LDA #$87
290 DATA  133,141     : REM  STA $8D
300 DATA  177,139     : REM  LDA ($8B),Y
310 DATA  133,142     : REM  STA $8E
320 DATA  169,8       : REM  LDA #$08
330 DATA  133,143     : REM  STA $8F
340 DATA  136         : REM  DEY
350 DATA  198,252     : REM  DEC $FC
360 ::                  REM ROTATE-BYTE
370 DATA  38,142      : REM  ROL $8E
380 DATA  144,16      : REM  BCC $10
390 DATA  189,6,1     : REM  LDA $106,X
400 DATA  145,251     : REM  STA ($FB),Y
410 DATA  136         : REM  DEY
420 DATA  36,141      : REM  BIT $8D
430 DATA  16,6        : REM  BPL $06
440 DATA  189,5,1     : REM  LDA $105,X
450 DATA  145,251     : REM  STA ($FB),Y
460 DATA  136         : REM  DEY
470 ::                  REM BIT-CLEAR
480 DATA  202         : REM  DEX
490 DATA  38,141      : REM  ROL $8D
500 DATA  144,1       : REM  BCC $01
510 DATA  202         : REM  DEX
520 ::                  REM OVER
530 DATA  198,143     : REM  DEC $8F
540 DATA  208,226     : REM  BNE $E2
550 DATA  56          : REM  SEC
560 DATA  152         : REM  TYA
```

```
570 DATA  101,251     : REM  ADC $FB
580 DATA  133,251     : REM  STA $FB
590 DATA  144,2       : REM  BCC $02
600 DATA  230,252     : REM  INC $FC
610 ::                  REM CLEAR-STACK
620 DATA  162,0       : REM  LDX #0
630 ::                  REM REPEAT
640 DATA  104         : REM  PLA
650 DATA  149,139     : REM  STA $8B,X
660 DATA  232         : REM  INX
670 DATA  224,6       : REM  CPX #$06
680 DATA  208,248     : REM  BNE $F8
690 DATA  104,168     : REM  PLA : TAY
700 DATA  104,170     : REM  PLA : TAX
710 DATA  104         : REM  PLA
720 DATA  40          : REM  PLP
730 DATA  96          : REM  RTS
740 ::                  REM TEST-ROUTINE
750 DATA  169,240     : REM  LDA #$F0
760 DATA  162,15      : REM  LDX #$0F
770 DATA  160,255     : REM  LDY #$FF
780 DATA  32,0,192    : REM  JSR $C000
790 DATA  255         : REM EMBEDDED-BYTE
800 DATA  96          : REM  RTS
810 :
820 REM ** SET UP ZERO PAGE AND FREE RAM **
830 PRINT CHR$(147)
840 POKE 251,12   POKE 252,197
850 FOR N=139 TO 144      : POKE N,N : NEXT
860 FOR N=50432 TO 50440  : POKE N,0 : NEXT
870 :
880 SYS 49258         : REM SYS TEST-ROUTINE
890 :
900 REM ** READ RESULTS **
910 FOR LOOP=50432 TO 50443
920  READ NAME$
930  PRINT NAME$;
940  PRINT PEEK(LOOP)
950 NEXT LOOP
960 :
```

```
 970 DATA "ZERO PAGE  ","ZERO PAGE+1"
 980 DATA "ZERO PAGE+2","ZERO PAGE+3"
 990 DATA "ZERO PAGE+4","ZERO PAGE+5"
1000 DATA "Y REGISTER ","X REGISTER "
1010 DATA "ACCUMULATOR","STATUS     "
1020 DATA "PC LOW     ","PC HIGH    "
```

The problem to solve next is that of where to place the user stack. This will depend on your own requirements, so to make the whole thing flexible, a vectored address in the bytes at $FB and $FC contains the stack address. In the program listed above, this is $C512 (line 840). The vectored address is, in fact, the address + 12. This is because the stack is pushed in reverse (decreasing) order.

When executed, the coding first pushes all the processor registers on to the hardware stack and moves the stack pointer across into the X register (lines 90 to 140). Next, the six zero page pseudo-registers are pushed there (lines 150 to 190). The return address from the subroutine call is then incremented on the stack, using the contents of the X register (stack pointer) to access it (lines 200 to 240). The two bytes that form the RTS address are copied into pseudo-register 1 (now safely on the hardware stack) to form a vector though which the embedded data byte can be loaded into the accumulator and then saved for use in zero page (lines 250 to 310).

In line 280, a pre-defined byte was loaded into the accumulator and saved in zero page. This byte holds a bit code that will inform the program as to whether the register being pulled from the hardware stack for transfer to the software stack is one or two bytes long. The byte value, $87, is 10000111 in binary and the set bits correspond to the two-byte registers, the Program Counter and the three pseudo-registers. By rotating this byte left after each pull operation and using the BIT operation, the Negative flag can be tested to see if a further pull is needed. All this and the copy hardware stack/push software stack is handled by lines 320 to 550.

Finally, the registers and pseudo-registers are restored to their original values (lines 620 to 730). The test routine between lines 750 and 800 shows the way the program is used. When run, the test procedure produces the following output on the screen:

```
ZERO PAGE    139
ZERO PAGE+1  140
ZERO PAGE+2  141
ZERO PAGE+3  142
ZERO PAGE+4  143
ZERO PAGE+5  144
Y REGISTER   255
X REGISTER    15
```

```
ACCUMULATOR  24Ø
STATUS       176
PC LOW       115
PC HIGH      192
```

As can be seen, the zero page bytes contain the values POKEd into them by the FOR...NEXT loop of line 83Ø while the accumulator and Index registers display their seeded values (lines 75Ø to 77Ø). The Program Counter holds 192 * 256 + 115, or $CØ73, which was the point in the program where its contents where pushed at line 78Ø.

This program could be extended to provide a routine to perform a pull user stack, to copy the contents of a software stack into the processor and pseudo-registers.

Line-by-line

A line-by-line description of Program 17 follows:

line 9Ø	:	save all processor registers on hardware stack
line 14Ø	:	move stack pointer into X for index
line 15Ø	:	entry for PUSH-ZERO-PAGE
line 16Ø	:	get zero page byte
line 17Ø	:	push on to hardware stack
line 18Ø	:	decrement index
line 19Ø	:	branch to PUSH-ZERO-PAGE until done
line 2ØØ	:	increment low byte of RTS address
line 21Ø	:	get it from stack
line 22Ø	:	and save in zero page
line 23Ø	:	if not equal branch to PC-LOW
line 24Ø	:	else increment page byte of RTS address
line 25Ø	:	entry for PC-LOW
line 26Ø	:	get high byte of RTS address
line 27Ø	:	and save it to form vector
line 28Ø	:	get bit code to indicate register size
line 29Ø	:	and save it
line 3ØØ	:	get embedded code after subroutine call
line 31Ø	:	and save it
line 32Ø	:	eight bits in embedded byte to test
line 33Ø	:	save bit count
line 34Ø	:	decrement index to $FF
line 35Ø	:	decrement high byte of vectored address at $FB

line 36Ø : entry for ROTATE-BYTE
line 37Ø : move next coded bit into Carry flag
line 38Ø : if bit clear skip it, branch to BIT-CLEAR
line 39Ø : otherwise get byte from stack
line 4ØØ : save it on user stack
line 41Ø : decrement index
line 42Ø : is it a two byte register?
line 43Ø : no, so branch to BIT-CLEAR
line 44Ø : yes, so get the second byte from the stack
line 45Ø : and save it on the user stack
line 46Ø : decrement index
line 47Ø : entry for BIT-CLEAR
line 48Ø : decrement hardware stack index
line 49Ø : move bit of register code into Carry flag
line 5ØØ : if clear, branch to OVER
line 51Ø : else decrement hardware stack index
line 52Ø : entry for OVER
line 53Ø : decrement bit counter
line 54Ø : and repeat until all done
line 55Ø : set Carry flag
line 56Ø : move user stack pointer into accumulator
line 57Ø : add to low byte of address
line 58Ø : and save
line 59Ø : branch to CLEAR-STACK if carry is clear
line 6ØØ : else increment high byte of address
line 61Ø : entry for CLEAR-STACK
line 62Ø : initialize X register
line 63Ø : entry for REPEAT
line 64Ø : pull byte from stack
line 65Ø : and restore zero page
line 66Ø : increment index
line 67Ø : all bytes restored?
line 68Ø : no, branch to REPEAT
line 69Ø : yes, restore all registers
line 73Ø : back to calling routine
line 74Ø : entry for TEST-ROUTINE
line 75Ø : seed registers
line 78Ø : call user stack routine
line 79Ø : embedded byte
line 8ØØ : back to BASIC

BINARY INS AND OUTS

Sometimes when printing the values of registers, it is necessary to have their binary representation—for example, in the case of the Status register, because we are concerned with the state of the particular bits within it, rather than the overall value of the contents. Program 18 provides a short routine which produces such a binary output from a decimal input. This could easily be adapted for use within a program such as the software stack given above.

Program 18

```
 1Ø REM ** PRINT ACCUMULATOR AS A **
 2Ø REM ** BINARY NUMBER **
 3Ø CODE=49152
 4Ø FOR LOOP=Ø TO 17
 5Ø  READ BYTE
 6Ø  POKE CODE+LOOP,BYTE
 7Ø NEXT LOOP
 8Ø :
 9Ø REM ** M/C DATA **
1ØØ DATA  162,Ø        : REM  LDX #$Ø8
11Ø DATA  72           : REM  PHA
12Ø ::                   REM NEXT-BIT
13Ø DATA  1Ø4          : REM  PLA
14Ø DATA  1Ø           : REM  ASL A
15Ø DATA  72           : REM  PHA
16Ø DATA  169,48       : REM  LDA #$3Ø
17Ø DATA  1Ø5,Ø        : REM  ADC #$ØØ
18Ø DATA  32,21Ø,255   : REM  JSR $FFD2
19Ø DATA  2Ø2          : REM  DEX
2ØØ DATA  2Ø8,243      : REM  BNE $F3
21Ø DATA  1Ø4          : REM  PLA
22Ø DATA  96           : REM  RTS
23Ø :
24Ø REM ** SET UP DEMO RUN **
25Ø REM LDA $FB : JSR $CØØØ : RTS
26Ø POKE 82Ø,165 :POKE 821,251
27Ø POKE 822,32 : POKE 823,Ø
28Ø POKE 824,192 : POKE 825,96
29Ø PRINT CHR$(147)   PRINT
3ØØ INPUT "INPUT A NUMBER ";A$
```

```
31Ø A=VAL(A$)
32Ø POKE 251,A
33Ø PRINT"BINARY VALUE IS :";
34Ø SYS 82Ø
```

Line-by-line

The following line-by-line description should make the program's operation clear. It is simply moving each bit of the accumulator in turn into the Carry flag position, using the arithmetic shift left operation (see Figure 8.1) and adding its value to the ASCII code for Ø, i.e.

```
accumulator=48+carry
```

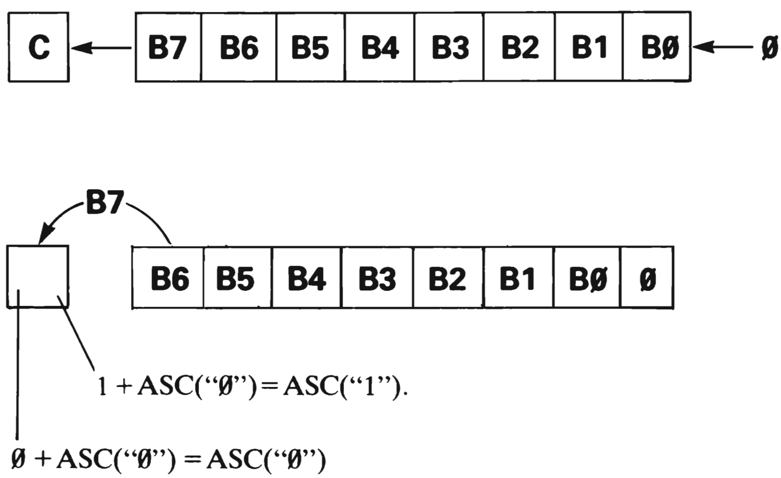


Figure 8.1 Arithmetic shift left

If the Carry flag is clear, the result will be 48+Ø=48, so the CHROUT routine will print a Ø. On the other hand, if the Carry flag is set, the result of the addition will be 48+1=49, so a 1 will be printed by CHROUT.

`line 1ØØ` : eight bits in a byte
`line 11Ø` : push accumulator on to stack
`line 12Ø` : entry for NEXT-BIT

line 13Ø : restore accumulator
line 14Ø : shift bit 7 into carry
line 15Ø : save shifted accumulator on stack
line 16Ø : get ASCII code for Ø
line 17Ø : add carry
line 18Ø : print either Ø or 1
line 19Ø : decrement bit counter
line 2ØØ : do NEXT-BIT until complete
line 21Ø : pull stack to balance push
line 22Ø : back to BASIC

COME IN

By reversing this process, it is possible to input a number directly into the accumulator in binary form as Program 19 shows. The program scans the keyboard for a pressed 1 or Ø key and the Carry flag is set or cleared respectively. A copy of the accumulator, initially cleared, is kept on the hardware stack and restored each time round to rotate the carry bit into it using the rotate left operation (see Figure 8.2). The loop is executed eight times, once for each bit, and on completion, the accumulator holds the hexadecimal value of the binary number.

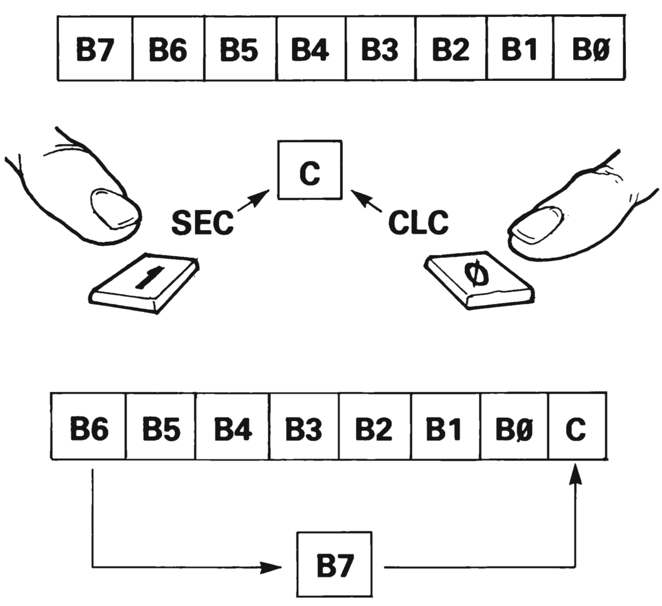


Figure 8.2 Input a number directly into the accumulator

Program 19

```
1Ø REM ** INPUT A HEX NUMBER IN BINARY FORM **
2Ø CODE=49152
3Ø FOR LOOP=Ø TO 41
4Ø READ BYTE
5Ø POKE CODE+LOOP,BYTE
6Ø NEXT LOOP
7Ø :
8Ø REM ** M/C DATA **
9Ø DATA  162,8         : REM  LDX #$Ø8
1ØØ DATA  169,Ø         : REM  LDA #$ØØ
11Ø DATA  72            : REM  PHA
12Ø DATA  24            : REM  CLC
13Ø ::                    REM MAINLOOP
14Ø DATA  134,243       : REM  STX $FD
15Ø ::                    REM LOOP
16Ø DATA  32,228,255    : REM  JSR $FFE4
17Ø DATA  24Ø,251       : REM  BEQ $FB
18Ø DATA  2Ø1,49        : REM  CMP #$31
19Ø DATA  24Ø,7         : REM  BEQ $Ø7
2ØØ DATA  2Ø1,48        : REM  CMP #$3Ø
21Ø DATA  2Ø8,243       : REM  BNE $F3
22Ø DATA  24            : REM  CLC
23Ø DATA  144,1         : REM  BCC $Ø1
24Ø ::                    REM SET
25Ø DATA  56            : REM  SEC
26Ø ::                    REM OVER
27Ø DATA  8             : REM  PHP
28Ø DATA  32,21Ø,255    : REM  JSR $FFD2
29Ø DATA  4Ø            : REM  PLP
3ØØ DATA  1Ø4           : REM  PLA
31Ø DATA  42            : REM  ROL A
32Ø DATA  72            : REM  PHA
33Ø DATA  166,253       : REM  LDX $FD
34Ø DATA  2Ø2           : REM  DEX
35Ø DATA  2Ø8,224       : REM  BNE $EØ
36Ø DATA  1Ø4           : REM  PLA
37Ø DATA  133,251       : REM  STA $FB
38Ø DATA  96            : REM  RTS
```

```
390 :
400 PRINT CHR$(147)
410 PRINT
420 PRINT"INPUT YOUR BINARY NUMBER :";
430 SYS CODE
440 PRINT PEEK(251)
```

Line-by-line

A line-by-line explanation of Program 19 now follows:

line 90 : eight bits to read
line 100 : clear accumulator—shift register
line 110 : push it on to stack
line 120 : clear the Carry flag
line 130 : entry for MAINLOOP
line 140 : save X register
line 150 : entry for LOOP
line 160 : jump to GETIN
line 170 : if null, branch to LOOP
line 180 : is it ASC"1"?
line 190 : yes, branch to SET
line 200 : is it ASC"0"?
line 210 : no, branch to LOOP
line 220 : yes, clear Carry flag
line 230 : and force branch to OVER
line 240 : entry for SET
line 250 : set Carry flag
line 260 : entry for OVER
line 270 : save Carry flag on stack
line 280 : print 0 or 1
line 290 : restore Carry flag
line 300 : restore accumulator
line 310 : move Carry flag into bit 0
line 320 : save accumulator
line 330 : restore bit count
line 340 : decrement it by one
line 350 : branch to MAINLOOP until all done
line 360 : restore accumulator
line 370 : save in zero page
line 380 : back to BASIC

Project

Convert the software stack program to print the binary values of each register upon completion.

Modify it further to allow register values to be seeded into the software stack test routine, using the binary input routine. Note that you should only attempt seeding the accumulator and Index registers. Why?

9 Move, Fill and Dump

MOVE IT!

The ability to move blocks of memory around within the bounds of the memory map is a necessity. When manipulating hi-resolution graphics, for example, large blocks of memory need to be moved around quickly and smoothly. The program could also be used to relocate sections of machine code rather than rewriting the assembler that created them—assuming, of course, that your code has been designed to make it portable.

At first sight, it may seem that the simplest method of moving a block of memory is to take the first byte to be moved and store it at the destination address, take the second byte and place it at the destination address + 1, and so forth. There would be no problem here if the destination address was outside the source address, but consider what would happen if the destination address was within the bounds to be searched by the source address—that is, the two regions overlapped. Figure 9.1 illustrates the problem using this straightforward method to move a block of five bytes forward by just a single byte, relocating the five bytes from $C500 to $C501.

Using the obvious method, the first character, 'S', is moved from $C500 to $C501 thereby overwriting the 'A'. The program then takes the next character at location START+1 ($C501), the 'S' that has just been written there, and places it at START+2 ($C502)

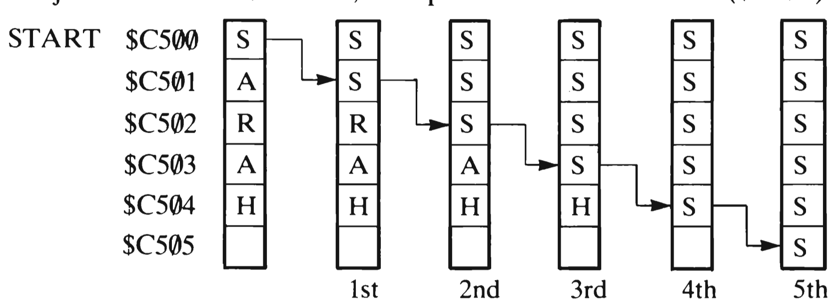


Figure 9.1 The overwriting move sequence

overwriting the 'R'. As you can see, the end result is SSSSS—the whole block is full of 'S's—not the required effect!

To avoid this problem, the MOVE routine acts 'intellegently' and if it calculates that an overwrite would occur, performs the movement of bytes in the reverse order, starting at the highest address and moving down the memory map as Figure 9.2 shows.

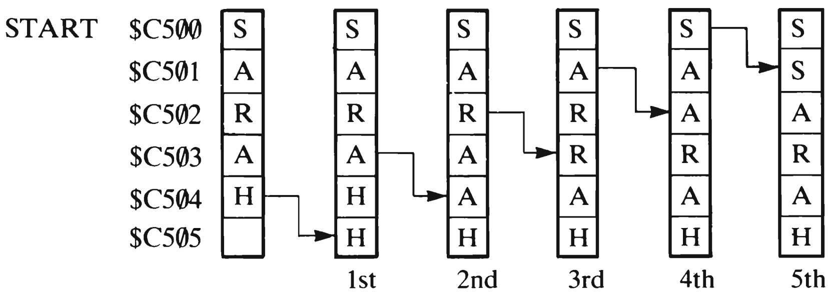


Figure 9.2 The correct move sequence

Program 20

```
10 REM ** MEMORY BLOCK MOVE RQUTINE **
20 REM ** 109 BYTES LONG WHEN ASSEMBLED **
30 REM ** PLUS 5 DATA BYTES IN ZERO PAGE **
40 CODE=49152
50 FOR LOOP=0 TO 108
60  READ BYTE
70  POKE CODE+LOOP,BYTE
80 NEXT LOOP
90 :
100 REM ** M/C DATA **
110 DATA 56          : REM  SEC
120 DATA 165,251     : REM  LDA $FB
130 DATA 229,253     : REM  SBC $FD
140 DATA 170         : REM  TAX
150 DATA 165,252     : REM  LDA $FC
160 DATA 229,254     : REM  SBC $FE
170 DATA 168         : REM  TAY
180 DATA 138         : REM  TXA
190 DATA 205,52,3    : REM  CMP $334
200 DATA 152         : REM  TYA
210 DATA 237,53,3    : REM  SBC $335
```

```
220 DATA 176,2        : REM  BCS $02
230 DATA 144,35       : REM  BCC $23
240 ::                  REM MOVE-LEFT
250 DATA 160,0        : REM  LDY #$00
260 DATA 174,53,3     : REM  LDX $335
270 DATA 240,14       : REM  BEQ $0E
280 ::                  REM LEFT-COMPLETE-PAGES
290 DATA 177,253      : REM  LDA ($FD),Y
300 DATA 145,251      : REM  STA ($FB),Y
310 DATA 200          : REM  INY
320 DATA 208,249      : REM  BNE $F9
330 DATA 230,254      : REM  INC $FE
340 DATA 230,252      : REM  INC $FC
350 DATA 202          : REM  DEX
360 DATA 208,242      : REM  BNE $F2
370 ::                  REM LEFT-PARTIAL-PAGE
380 DATA 174,52,3     : REM  LDX $334
390 DATA 240,8        : REM  BEQ $08
400 ::                  REM LAST-LEFT
410 DATA 177,253      : REM  LDA ($FD),Y
420 DATA 145,251      : REM  STA ($FB),Y
430 DATA 200          : REM  INY
440 DATA 202          : REM  DEX
450 DATA 208,248      : REM  BNE $F8
460 ::                  REM EXIT
470 DATA 96           : REM  RTS
480 :
490 ::                  REM MOVE-RIGHT
500 DATA 24           : REM  CLC
510 DATA 173,53,3     : REM  LDA $335
520 DATA 72           : REM  PHA
530 DATA 101,254      : REM  ADC $FE
540 DATA 133,254      : REM  STA $FE
550 DATA 24           : REM  CLC
560 DATA 104          : REM  PLA
570 DATA 101,252      : REM  ADC $FC
580 DATA 133,252      : REM  STA $FC
590 DATA 172,52,3     : REM  LDY $334
600 DATA 240,9        : REM  BEQ $09
```

```
610 ::                    REM TRANSFER
620 DATA 136            : REM  DEY
630 DATA 177,253        : REM  LDA ($FD),Y
640 DATA 145,251        : REM  STA ($FB),Y
650 DATA 192,0          : REM  CPY #$00
660 DATA 208,247        : REM  BNE $F7
670 ::                    REM RIGHT-COMPLETE-PAGES
680 DATA 174,53,3       : REM  LDX $335
690 DATA 240,221        : REM  BEQ $DD
700 ::                    REM UPDATE
710 DATA 198,254        : REM  DEC $FE
720 DATA 198,252        : REM  DEC $FC
730 ::                    REM PAGE
740 DATA 136            : REM  DEY
750 DATA 177,253        : REM  LDA ($FD),Y
760 DATA 145,251        : REM  STA ($FB),Y
770 DATA 192,0          : REM  CPY #$00
780 DATA 208,247        : REM  BNE $F7
790 DATA 202            : REM  DEX
800 DATA 208,240        : REM  BNE $F0
810 DATA 96             : REM  RTS
820 :
830 REM ** SET UP VARIABLES **
840 PRINT CHR$(147)
850 PRINT"    *** MEMORY MOVER  V1.1 ***"
860 INPUT"START ADDRESS    ";S
870 INPUT"DESTINATION      ";D
880 INPUT"LENGTH IN BYTES  ";L
890 :
900 S1=INT(S/256) : S2=S-(S1*256)
910 D1=INT(D/256) : D2=D-(D1*256)
920 L1=INT(L/256) : L2=L-(L1*256)
930 :
940 POKE 251,D2   : POKE 252,D1
950 POKE 253,S2   : POKE 254,S1
960 POKE 820,L2   : POKE 821,L1
970 :
980 REM ** SET UP DEMO **
990 FOR N=0 TO 15
1000 POKE 828+N,N
```

```
1010 POKE 900+N,0
1020 NEXT N
1030 :
1040 SYS CODE
1050 :
1060 REM ** PRINT THE RESULTS! **
1070 FOR N=0 TO 15
1080 PRINT PEEK(828+N);"      ";
1090 PRINT PEEK(900+N)
1100 NEXT N
```

Bytes reserved:

251–252	($FB–$FC)	:	Destination vector
253–254	($FD–$FE)	:	Source vector
820–821	($334–$335)	:	Length of block to be moved

When run, the BASIC test requests three inputs: the START address of the memory block to be moved, its DESTINATION address and its LENGTH in bytes. All values should be entered as decimal values. Thus, to move a 1K block of memory from 49152 to 56000, the values to input are:

```
START ADDRESS       : 49152
DESTINATION         : 56000
LENGTH IN BYTES       1024
```

For reasons already explained, the coding begins by ascertaining whether a left-move or a right-move operation is required. It calculates this (lines 110 to 210) by subtracting the source address from the destination address. If the result is less than the number of bytes to be moved, overwriting would occur using the MOVE-LEFT routine, so the MOVE-RIGHT coding is called (line 230). If the memory locations do not overlap, the quicker MOVE-LEFT routine is selected (line 220). For further description purposes we will examine the MOVE-LEFT routine (lines 240 to 470).

Memory movement is performed in two phases: complete memory pages are first relocated, and then any remaining bytes in the final partial page are moved. These details are held in the length of block bytes $334 and $335.

The routine begins by loading the number of pages to be moved into the X register (line 260), branching to LEFT-PARTIAL-PAGE if it is zero (line 280). Transfer of data bytes is completed using post-indexed indirect addressing through the zero page vectors. When all the whole pages have been transferred, any remaining bytes are transferred by the LEFT-PARTIAL-PAGE loop (lines 370 to 450).

The MOVE-RIGHT routine is similar in operation, except that it starts at the highest memory location referenced and moves down through memory, the highest address of the source and destination being calculated in lines 500 to 650.

Line-by-line

A line-by-line description of Program 20 now follows:

`line 110`	:	set Carry flag
`line 120`	:	get low byte destination address
`line 130`	:	subtract low byte source address
`line 140`	:	transfer result into X register
`line 150`	:	get high byte destination address
`line 160`	:	subtract high byte source address
`line 170`	:	save result in X register
`line 180`	:	restore result of low byte subtraction
`line 190`	:	compare it with low byte of length
`line 200`	:	restore result of high byte subtraction
`line 210`	:	subtract high byte of length from it
`line 220`	:	if Carry flag set, branch to MOVE-LEFT
`line 230`	:	else branch to MOVE-RIGHT
`line 240`	:	entry for MOVE-LEFT
`line 250`	:	initialize index
`line 260`	:	get number of pages to be moved
`line 270`	:	if zero, branch to LEFT-PARTIAL-PAGE
`line 280`	:	entry for LEFT-COMPLETE-PAGES
`line 290`	:	get source byte
`line 300`	:	store at destination
`line 310`	:	increment index
`line 320`	:	branch to LEFT-COMPLETE-PAGES until page done
`line 330`	:	increment source page
`line 340`	:	increment destination page
`line 350`	:	decrement page counter
`line 360`	:	branch to LEFT-COMPLETE-PAGES until all moved
`line 370`	:	entry for LEFT-PARTIAL-PAGE
`line 380`	:	get number of bytes on page to be moved
`line 390`	:	if zero, branch to EXIT

line 4ØØ : entry for LAST-LEFT
line 41Ø : get source byte
line 42Ø : store at destination
line 43Ø : increment index
line 44Ø : decrement byte count
line 45Ø : branch to LAST-LEFT until done
line 46Ø : entry for EXIT
line 47Ø : back to BASIC
line 49Ø : entry for MOVE-RIGHT
line 5ØØ : clear Carry flag
line 51Ø : get number of pages to be moved
line 52Ø : save on stack
line 53Ø : add it to source high byte
line 54Ø : and save result
line 55Ø : reclear Carry flag
line 56Ø : get length high byte off stack
line 57Ø : add it to destination high byte
line 58Ø : and save the result
line 59Ø : get low byte of length into Y register
line 6ØØ : branch to RIGHT-COMPLETE-PAGES if zero
line 61Ø : entry for TRANSFER
line 62Ø : decrement index
line 63Ø : get source byte
line 64Ø : and copy to destination
line 65Ø : is Y = Ø?
line 66Ø : no, branch to TRANSFER
line 67Ø : entry for RIGHT-COMPLETE-PAGES
line 68Ø : get number of pages to be moved
line 69Ø : if zero, branch to EXIT
line 7ØØ : entry for UPDATE
line 71Ø : decrement number of pages to do
line 72Ø : and also destination
line 73Ø : entry for PAGE
line 74Ø : decrement index
line 75Ø : get source byte
line 76Ø : copy to destination
line 77Ø : is Y = Ø?
line 78Ø : no, branch to PAGE

line 79∅ : decrement page counter
line 8∅∅ : if not zero, branch to UPDATE
line 81∅ : return to BASIC

FILL

Program 21 provides the BASIC loader listing to implement a memory FILL routine, which is particularly useful for clearing sections of RAM with a pre-determined value.

Program 21

```
 1∅ REM ** MEMORY FILL ROUTINE **
 2∅ REM ** 3∅ BYTES LONG WHEN ASSEMBLED **
 3∅ REM ** PLUS 5 DATA BYTES IN ZERO PAGE **
 4∅ CODE=49152
 5∅ FOR LOOP=∅ TO 3∅
 6∅  READ BYTE
 7∅  POKE CODE+LOOP,BYTE
 8∅ NEXT LOOP
 9∅ :
1∅∅ REM ** M/C DATA **
11∅ DATA  165,255  : REM  LDA $FF
12∅ DATA  166,252  : REM  LDX $FC
13∅ DATA  24∅,12   : REM  BEQ $∅C
14∅ DATA  16∅,∅    : REM  LDY #$∅∅
15∅ ::               REM COMPLETE-PAGE
16∅ DATA  145,253  : REM  STA ($FD),Y
17∅ DATA  2∅∅      : REM  INY
18∅ DATA  2∅8,251  : REM  BNE $FB
19∅ DATA  23∅,254  : REM  INC $FE
2∅∅ DATA  2∅2      : REM  DEX
21∅ DATA  2∅8,246  : REM  BNE $F6
22∅ ::               REM PARTIAL-PAGE
23∅ DATA  166,251  : REM  LDX $FB
24∅ DATA  24∅,8    : REM  BEQ $∅8
25∅ DATA  16∅,∅    : REM  LDY #$∅∅
26∅ ::               REM AGAIN
27∅ DATA  145,253  : REM  STA ($FD),Y
```

```
280 DATA  200        : REM  INY
290 DATA  202        : REM  DEX
300 DATA  208,250    : REM  BNE $FA
310 ::                 REM FINISH
320 DATA  96         : REM  RTS
330 :
340 REM ** GET DETAILS **
350 PRINT CHR$(147)
360 INPUT"FILL DATA          :";F
370 INPUT"START ADDRESS      :";S
380 INPUT"NUMBER OF BYTES :";L
390 :
400 S1=INT(S/256)    : S2=S-(S1*256)
410 L1=INT(L/256)    : L2=L-(L1*256)
420 :
430 POKE 251,L2     : POKE 252,L1
440 POKE 253,S2     : POKE 254,S1
450 POKE 255,F
460 :
470 SYS CODE
```

Bytes reserved:

251–252	($FB–$FC)	: number of bytes to be filled
253–254	($FD–$FE)	: start of address of bytes to be filled
255	($FF)	: value to fill with

When executed, the machine code expects to find the fill value, the start address and the amount of memory to be filled, in five zero page bytes of memory from $FB. Input of each of these is handled by a few lines of BASIC from line 360. To clear a 1K block of RAM from $C500 with zero, the following information should be entered in response to the 64's prompt:

```
FILL DATA          : 0
START ADDRESS      : 49152
NUMBER OF BYTES    : 1024
```

The FILL routine works in a similar manner to the MOVE routine described above, dealing with whole and partial pages separately. The main fill loop is embodied in lines 150 to 300.

Line-by-line

A line-by-line description of the program now follows:

line 11Ø	:	get data with which to fill
line 12Ø	:	get number of complete pages to be filled
line 13Ø	:	if zero, branch to PARTIAL-PAGE
line 14Ø	:	initialize index
line 15Ø	:	entry for COMPLETE-PAGE
line 16Ø	:	fill byte
line 17Ø	:	increment index
line 18Ø	:	branch to COMPLETE-PAGE until all of page is done
line 19Ø	:	increment page
line 2ØØ	:	decrement page counter
line 21Ø	:	branch to COMPLETE-PAGE until all pages are filled
line 22Ø	:	entry for PARTIAL-PAGE
line 23Ø	:	get number of bytes left to be filled
line 24Ø	:	if zero, branch to FINISH
line 25Ø	:	else clear index
line 26Ø	:	entry for AGAIN
line 27Ø	:	fill byte
line 28Ø	:	increment index
line 29Ø	:	decrement bytes left to do count
line 3ØØ	:	branch to AGAIN until all filled
line 31Ø	:	entry for FINISH
line 32Ø	:	back to BASIC

A MEMORY DUMP

A hex and ASCII dump of memory can be extremely useful, not only within machine code programs, but also when used from a BASIC program. Most often it provides information about the way a program is manipulating numeric and string variables and tables. Figure 9.3 shows the type of dump produced by the routine: twenty-four lines of eight bytes each. The example shows some text stored in memory. Each line starts with the current address, followed by the eight bytes stored in memory from that point. The far right of the listing provides the ASCII equivalents of each byte. Any non-ASCII character (that is, one greater than $7F) or control code (those less than $2Ø) is represented by a full stop.

```
C108 : 54 68 69 73 20 69 73 20 This is
C110 : 61 20 73 69 6D 70 6C 65 a simple
C118 : 20 65 78 61 6D 70 6C 65  example
C120 : 20 6F 66 20 68 6F 77 20  of how
C128 : 74 68 65 20 8D 64 75 6D the .dum
C130 : 70 20 72 6F 75 74 69 6E p routin
C138 : 65 20 66 6F 72 20 74 68 e for th
C140 : 65 20 43 6F 6D 6D 6F 64 e Commod
C148 : 6F 72 65 20 36 34 20 8D ore 64 .
C150 : 77 6F 72 6B 73 2E 0D 54 works..T
C158 : 68 65 20 64 75 6D 70 20 he dump
C160 : 63 61 6E 20 62 65 20 64 can be d
C168 : 69 76 69 64 65 64 20 69 ivided i
C170 : 6E 74 6F 20 74 68 72 65 nto thre
C178 : 65 20 8D 73 65 63 74 69 e .secti
C180 : 6F 6E 73 2E 20 54 68 65 ons. The
C188 : 20 66 69 72 73 74 20 63  first c
C190 : 6F 6C 75 6D 6E 20 6C 69 olumn li
C198 : 73 74 73 20 74 68 65 20 sts the
C1A0 : 8D 73 74 61 72 74 20 61 .start a
C1A8 : 64 64 72 65 73 73 20 6F ddress o
C1B0 : 66 20 74 68 65 20 62 6C f the bl
C1B8 : 6F 63 6B 2E 20 54 68 65 ock. The
C1C0 : 20 73 65 63 6F 6E 64 20  second
C1C8 : 8D 63 6F 6C 75 6D 6E 20 .column
C1D0 : 69 73 20 69 6E 20 66 61 is in fa
C1D8 : 63 74 20 74 68 65 20 68 ct the h
C1E0 : 65 78 61 64 65 63 69 6D exadecim
C1E8 : 61 6C 20 8D 76 61 6C 75 al .valu
C1F0 : 65 73 20 6F 66 20 65 69 es of ei
C1F8 : 67 68 74 20 62 79 74 65 ght byte
C200 : 73 20 66 72 6F 6D 20 74 s from t
C208 : 68 69 73 20 8D 61 64 64 his .add
C210 : 72 65 73 73 2E 20 46 69 ress. Fi
C218 : 6E 61 6C 6C 79 20 74 68 nally th
C220 : 65 20 6C 61 73 74 20 63 e last c
C228 : 6F 6C 75 6D 6E 20 8D 64 olumn .d
C230 : 65 70 69 63 74 73 20 74 epicts t
C238 : 68 65 20 41 53 43 49 49 he ASCII
C240 : 20 76 61 6C 75 65 73 20  values
C248 : 6F 66 20 74 68 65 73 65 of these
C250 : 20 8D 62 79 74 65 73 2E  .bytes.
C258 : 20 75 6E 6C 65 73 73 20  unless
C260 : 74 68 65 20 62 79 74 65 the byte
C268 : 20 69 73 20 6E 6F 6E 2D  is non-
C270 : 41 53 43 49 49 20 8D 77 ASCII .w
C278 : 68 69 63 68 20 69 73 20 hich is
C280 : 74 68 65 6E 20 64 69 73 then dis
C288 : 70 6C 61 79 65 64 20 61 played a
C290 : 73 20 61 2C 66 75 6C 6C s a full
C298 : 20 73 74 6F 70 21 0D 00  stop!..
C2A0 : 00 4C 00 C9 A9 FF 85 22 .L....."
C2A8 : 08 60 00 00 00 00 00 00 .`......
```

Figure 9.3 Memory dump

As it stands, the routine requires three zero page data bytes, two for the start address and one for the number of eight byte lines to be dumped. The routine also employs the ADDRESS-PRINT and HEXPRINT routines discussed earlier.

Program 22

```
10 REM ** DUMP LINES OF 8 BYTES OF **
20 REM ** MEMORY IN HEX AND ASCII  **
30 CODE=49152
40 FOR LOOP=0 TO 111
50  READ BYTE
60  POKE CODE+LOOP,BYTE
70 NEXT LOOP
80 :
90 REM ** M/C DATA **
100 DATA 32,71,192    : REM  JSR $C047
110 ::                  REM HEX-BYTES
120 DATA 162,7        : REM  LDX #$07
130 DATA 160,0        : REM  LDY #$00
140                     REM HEX-LOOP
150 DATA 177,251      : REM  LDA ($FB),Y
160 DATA 32,90,192    : REM  JSR $C05A
170 DATA 32,66,192    : REM  JSR $C042
180 DATA 200          : REM  INY
190 DATA 202          : REM  DEX
200 DATA 16,244       : REM  BPL $F4
210 DATA 32,66,192    : REM  JSR $C042
220 ::                  REM ASCII-BYTES
230 DATA 162,7          REM  LDX #$07
240 DATA 160,0        : REM  LDY #$00
250 ::                  REM ASCII-LOOP
260 DATA 177,251      : REM  LDA ($FB),Y
270 DATA 201,32       : REM  CMP #$20
280 DATA 48,4         : REM  BMI $04
290 DATA 201,128      : REM  CMP #$80
300 DATA 144,2        : REM  BCC $02
310 ::                  REM FULL-STOP
320 DATA 169,46         REM  LDA #$2E
330 ::                  REM LEAP-FROG
```

```
340 DATA 32,210,255   : REM  JSR $FFD2
350 DATA 200          : REM  INY
360 DATA 202          : REM  DEX
370 DATA 16,237       : REM  BPL $ED
380 DATA 169,13       : REM  LDA #$0D
390 DATA 32,210,255   : REM  JSR $FFD2
400 DATA 24           : REM  CLC
410 DATA 165,251      : REM  LDA $FB
420 DATA 105,8        : REM  ADC #$08
430 DATA 133,251      : REM  STA $FB
440 DATA 144,2        : REM  BCC $02
450 DATA 230,252      : REM  INC $FC
460 ::                  REM NO-CARRY
470 DATA 198,254      : REM  DEC $FE
480 DATA 208,191      : REM  BNE $BF
490 DATA 96           : REM  RTS
500 ::                  REM SPACE
510 DATA 169,32       : REM  LDA #$20
520 DATA 76,210,255   : REM  JMP $FFD2
530 ::                  REM ADDRESS-PRINT
540 DATA 162,251      : REM  LDX #$FB
550 DATA 181,1        : REM  LDA 1,X
560 DATA 32,90,192    : REM  JSR $C05A
570 DATA 181,0        : REM  LDA 0,X
580 DATA 32,90,192    : REM  JSR $C05A
590 DATA 32,66,192    : REM  JSR $C042
600 DATA 32,66,192    : REM  JSR $C042
610 DATA 96           : REM  RTS
620 ::                  REM HEXPRINT
630 DATA 72           : REM  PHA
640 DATA 74,74        : REM  LSR A : LSR A
650 DATA 74,74        : REM  LSR A : LSR A
660 DATA 32,99,192    : REM  JSR $C063
670 DATA 104          : REM  PLA
680 ::                  REM FIRST
690 DATA 41,15        : REM  AND #$0F
700 DATA 201,10       : REM  CMP #$0A
710 DATA 144,2        : REM  BCC $02
720 DATA 105,6        : REM  ADC #$06
730 ::                  REM OVER
```

```
740 DATA 105,48        : REM  ADC #$30
750 DATA 76,210,255    : REM  JMP $FFD2
760 :
770 REM ** INPUT DETAILS FOR DUMP **
780 PRINT CHR$(147)
790 INPUT"DUMP START ADDRESS ";A
800 HIGH=INT(A/256)
810 LOW=A-(HIGH*256)
820 POKE 251,LOW : POKE 252,HIGH
830 INPUT"NUMBER OF LINES (20/SCREEN) ";B
840 POKE 254,B
850 SYS CODE
```

The program's operation is quite simple, using the X register to count the bytes as they are printed across the screen using HEXPRINT (lines 120 to 210). The second section of code (lines 220 to 370) is responsible for printing either the ASCII character contained in the byte, or a full stop if an unprintable character or a control code is encountered. The final section of code moves the cursor down one line and increments the address counter. The whole loop is repeated until the line count reaches zero.

Line-by-line

A line-by-line description of the Program 22 now follows:

line 100 : print start address of current line
line 110 : entry for HEX-BYTES
line 120 : eight bytes to do (0–7)
line 130 : clear index
line 140 : entry for HEX-LOOP
line 150 : get byte through vectored address
line 160 : print it as two hex digits
line 170 : print a space
line 180 : increment index
line 190 : decrement bit count
line 200 : branch to HEX-LOOP until all done
line 210 : print a space
line 220 : entry for ASCII-BYTES
line 230 : eight bytes to redo
line 240 : set index
line 250 : entry for ASCII-LOOP

line 26Ø : get byte through vectored address
line 27Ø : is it less than ASC" "?
line 28Ø : yes, branch to FULL-STOP
line 29Ø : is it greater than 128?
line 3ØØ : no, branch to LEAP-FROG
line 31Ø : entry for FULL-STOP
line 32Ø : get ASC"." into accumulator
line 33Ø : entry for LEAP-FROG
line 34Ø : print accumulator's contents
line 35Ø : increment index
line 36Ø : decrement bit count
line 37Ø : branch to ASCII-LOOP until all done
line 38Ø : get ASCII code for RETURN
line 39Ø : print new line
line 4ØØ : clear Carry flag
line 41Ø : get low byte of address
line 42Ø : add 8 to it
line 43Ø : save result
line 44Ø : if no carry, branch to NO-CARRY
line 45Ø : else increment high byte of address
line 46Ø : entry for NO-CARRY
line 47Ø : decrement line counter
line 48Ø : branch to start at $CØØØ until all lines done
line 49Ø : return to BASIC
line 5ØØ : entry to SPACE
line 51Ø : get ASCII code for space
line 52Ø : print it and return through jump
line 53Ø : entry to ADDRESS-PRINT
line 54Ø : load index into X register
line 55Ø : get high byte of address
line 56Ø : print it as two hex digits
line 57Ø : get low byte of address
line 58Ø : print it as two hex digits
line 59Ø : print a space
line 6ØØ : print a second space
line 61Ø : return to main program
line 62Ø : entry to HEXPRINT
line 63Ø : save accumulator on stack
line 64Ø : move high nibble into low nibble position
line 66Ø : call FIRST subroutine

line 67∅ : restore accumulator to do low byte
line 68∅ : entry for FIRST
line 69∅ : mask off high nibble
line 7∅∅ : is it less than 1∅?
line 71∅ : yes, so jump OVER
line 72∅ : add 7 to convert to A–F
line 73∅ : entry to OVER
line 74∅ : add 48 to convert to ASCII code
line 75∅ : print it and return

10 Hi-res Graphics

The Commodore 64 can support hi-resolution graphics. However, as you are no doubt aware, setting up the hi-res screen prior to using it can be a rather long-winded process, requiring several lines of BASIC text. In fact, four routines are normally required:

1. Move start of BASIC user area and set position for hi-res screen.
2. Clear screen memory.
3. Select screen colour and clear to that colour.
4. Reselect normal character mode.

All of these can be performed quite simply at machine level, and the routines for each follow. They can be compiled as DATA at the end of a graphics program, poked into memory at RUN time and executed via a SYS call. This does have one of the original disadvantages, in that a large chunk of program is required. However, the main advantage is speed, particularly in clearing the screen. Alternatively, any of these routines would make an admirable addition to the Wedge Operating System, allowing it to be called by name from within your programs. Suitable command names might be:

`@MOVEBAS`	: move BASIC program area to make room for hi-res screen
`@HIRES`	: select hi-res screen
`@CLEAR`	: clear hi-res screen
`@GCOL`	: clear to graphics colour specified in a dedicated byte
`@MODE`	: select normal character mode

Let us now examine each command in turn.

A BASIC MOVE

You may be wondering why we should bother to move the BASIC program area at all—why not just position the hi-res screen midway in memory? The reason for the careful positioning of the routine is as a matter of safety—placing the hi-res screen above the BASIC program area could lead to it being corrupted, especially if it is being used in conjunction with the program, because adding a line or two to the program could cause it to extend into the hi-res screen. Making sure the BASIC program fits in is no real safeguard either, as variables, strings and arrays all eat up memory at an incredible rate, and these could find their way into the screen memory. All these problems can be avoided by moving the start of BASIC up enough bytes to allow the hi-res screen to be tucked in underneath.

To do this requires a machine code program. The *Programmer's Reference Guide* lists five vectors associated with BASBAS (that's my mnemonic for BASIC's base!), as follows:

\$2B–\$2C	TXTTAB	:	start of BASIC text
\$2D–\$2E	VARTAB	:	start of BASIC variables
\$2F–\$3Ø	ARYTAB	:	start of BASIC arrays
\$31–\$32	STREND	:	end of BASIC arrays+1
\$281–\$282	MEMSTR	:	bottom of memory

To move BASIC, each of these vectors must be reset to point to the new start area and the first three bytes of the new start area must be cleared to keep the Kernal happy.

Program 23 performs each of these functions. The address of the new BASIC area is \$4ØØØ, which allows room for the hi-res screen plus 32 sprites.

Program 23

```
 1Ø REM ** MOVE BASIC PROGRAM AREA START **
 2Ø REM ** UP TO 16348 TO FREE HI-RES SCREEN **
 3Ø :
 4Ø CODE=49152
 5Ø FOR LOOP=Ø TO 39
 6Ø  READ BYTE
 7Ø  POKE CODE+LOOP,BYTE
 8Ø NEXT LOOP
 9Ø :
1ØØ REM ** M/C DATA **
11Ø DATA  169,Ø       : REM  LDA #$ØØ
```

```
12Ø DATA  141,2,64    : REM  STA $4ØØ2
13Ø DATA  141,1,64    : REM  STA $4ØØ1
14Ø DATA  141,Ø,64    : REM  STA $4ØØØ
15Ø DATA  141,129,2   : REM  STA $Ø281
16Ø DATA  169,64      : REM  LDA #$4Ø
17Ø DATA  133,44      : REM  STA $2C
18Ø DATA  133,46      : REM  STA $2E
19Ø DATA  133,48      : REM  STA $3Ø
2ØØ DATA  133,5Ø      : REM  STA $32
21Ø DATA  141,13Ø,2   : REM  STA $Ø282
22Ø DATA  169,1       : REM  LDA #$Ø1
23Ø DATA  133,43      : REM  STA $2B
24Ø DATA  169,3       : REM  LDA #$Ø3
25Ø DATA  133,45      : REM  STA $2D
26Ø DATA  133,47      : REM  STA $2F
27Ø DATA  133,49      : REM  STA $31
28Ø DATA  96          : REM  RTS
```

Line-by-line

A line-by-line description of Program 23 follows:

`line 11Ø` : initialize accumulator
`line 12Ø` : and clear first four bytes of new program area
`line 15Ø` : set low byte of MEMSTR (bottom of memory pointer)
`line 16Ø` : load high byte of new program area address into accumulator
`line 17Ø` : set high byte of TXTTAB
`line 18Ø` : set high byte of VARTAB
`line 19Ø` : set high byte of ARYTAB
`line 2ØØ` : set high byte of STREND
`line 21Ø` : set high byte of MEMSTR
`line 22Ø` : load accumulator with 1
`line 23Ø` : store in low byte of TXTTAB
`line 24Ø` : load accumulator with 3
`line 25Ø` : set low bytes of all vectored addresses

SELECTING HI-RES

Before selecting the hi-resolution screen mode, it is necessary to point the VIC chip to the start of screen memory. This is done by writing to the VIC Memory Control register located at $DØ18 (57272). The actual location is controlled by the condition of bits 3, 2 and 1. Table 10.1 details their settings for various addresses.

Table 10.1

Bit code	Value	Address selected	
xxxxØØØx	Ø	Ø–2Ø47	($ØØØO–$Ø7FF)
xxxxØØ1x	2	2Ø48–4Ø95	($Ø8ØØ–$ØFFF)
xxxxØ1Øx	4	4Ø96–6143	($1ØØØ–$17FF)
xxxxØ11x	6	6144–8191	($18ØØ–$1FFF)
xxxx1ØØx	8	8192–1Ø239	($2ØØØ–$27FF)
xxxx1Ø1x	1Ø	1Ø24Ø–12287	($28ØØ–$2FFF)
xxxx11Øx	12	12288–14335	($3ØØØ–$37FF)
xxxx111x	14	14336–16383	($38ØØ–$3FFF)

You can see from the table that the screen memory may be moved around in 2K block steps. An 'x' in each of the other bits denotes that these bits may be in either state. However, remember that these bits are controlling other aspects of the VIC's function, so that any reprogramming of bits 3, 2 and 1 must preserve the other bits. This is best done with the logical OR function. Looking at Table 10.1 we can see that bit 3 must be set to point the Memory Control register at location 8192. In BASIC this would simplify to:

```
1ØØ A=PEEK(53727)   : REM GET VALUE
11Ø A=A OR 8        : REM SET BIT 3
12Ø POKE 53727,A    : REM REPROGRAM
```

which translates to assembler as:

```
LDA #$Ø8
ORA $DØ18
STA $DØ18
```

Now that the hi-res screen has been defined, it can be switched in by setting bit 5 of the VIC Control register at $DØ11 (53265).

Again, the other bits in the register must be preserved, so the byte must be ORed with 32 (ØØ1ØØØØØ binary). In BASIC this is:

```
13Ø A=PEEK(53265)    : REM GET VALUE
14Ø A=A OR 32        : REM SET BIT 5
15Ø POKE 53265,A     : REM REPROGRAM
```

and in assembler:

```
LDA #$2Ø
ORA $DØ11
STA $DØ11
```

A CLEAR VIEW

Once hi-res mode has been selected, it will be filled with junk (often referred to as garbage). To clear this, each location must in turn be POKEd with zero. A BASIC program to do this would take the form:

```
2ØØ SB=8192
21Ø FOR L=SB TO SB+7999
22Ø  POKE L,Ø
23Ø NEXT L
```

Previously, in normal character mode, locations 1Ø24 to 2Ø23 were used to control which character was displayed—for example, POKEing a 1 into location 1Ø24 would make a letter A appear at the top left hand corner of the screen. When in hi-res mode, this area of memory is used to hold the colour information of that byte. Note that the colour information does not now come from the colour memory—colour details are taken directly from the hi-res screen itself. The high nibble of the byte (that is, bits 4 to 7) holds the colour code of any bit that is set in that 8 by 8 bit matrix, while the lower nibble (bits 3 to Ø) holds the colour of any bits that are clear in the same area.

To clear the hi-res screen to black ink on green paper in BASIC we would use:

```
24Ø FOR C=1Ø24 TO 2Ø23
25Ø  POKE C,13
26Ø NEXT C
```

If all the above BASIC program lines were to be combined and RUN, the resulting hi-res screen would take around 20 seconds to construct—a bit slow, you'll agree! Program 24 provides the machine code equivalent. Note that the value assigned to CODE is 49408 and NOT 49152 as we have been using previously. This is to allow the program to be used in conjunction with the MOVEBAS program described earlier. After you have entered and RUN MOVEBAS, try this one for an instant hi-res screen!

Program 24

```
 10 REM ** HI-RES GRAPHICS SCREEN SET AND
    CLEAR **
 20 CODE=49408
 30  FOR LOOP=0 TO 105
 40  READ BYTE
 50 POKE CODE+LOOP,BYTE
 60 NEXT LOOP
 70  :
 80 REM ** M/C DATA **
 85 ::                   REM SELECT-HI-RES
 90 DATA  169,8        : REM  LDA #$08
100 DATA  13,24,208    : REM  ORA $D018
110 DATA  141,24,208   : REM  STA $D018
120 DATA  169,32       : REM  LDA #$20
130 DATA  13,17,208    : REM  ORA $D011
140 DATA  141,17,208   : REM  STA $D011
150 ::                   REM CLEAR-SCREEN-MEMORY
160 DATA  169,0        : REM  LDA #$00
170 DATA  133,251      : REM  STA $FC
180 DATA  169,32       : REM  LDA #$20
190 DATA  133,252      : REM  STA $FC
200 DATA  169,64       : REM  LDA #$40
210 DATA  133,253      : REM  STA $FD
220 DATA  169,63       : REM  LDA #$3F
230 DATA  133,254      : REM  STA $FE
240 ::                   REM IN
250 DATA  165,252      : REM  LDA $FC
260 DATA  197,254      : REM  CMP $FE
270 DATA  208,9        : REM  BNE $09
280 DATA  165,251      : REM  LDA $FB
290 DATA  197,253      : REM  CMP $FD
300 DATA  208,3        : REM  BNE $03
```

```
310 DATA  76,62,192   : REM  JMP $C03E
320 ::                  REM CLEAR
330 DATA  160,0       : REM  LDY #$00
340 DATA  169,0       : REM  LDA #$00
350 DATA  145,251     : REM  STA ($FB),Y
360 DATA  230,251     : REM  INC $FB
370 DATA  208,231     : REM  BNE $E7
380 DATA  230,252     : REM  INC $FC
390 DATA  56          : REM  SEC
400 DATA  176,226     : REM  BCS $E2
410 :
420 ::                  REM COLOUR
430 DATA  169,0       : REM  LDA #$00
440 DATA  133,251     : REM  STA $FB
450 DATA  169,4       : REM  LDA #$04
460 DATA  133,252     : REM  STA $FC
470 DATA  169,231     : REM  LDA #$E7
480 DATA  133,253     : REM  STA $FD
490 DATA  169,7       : REM  LDA #$07
500 DATA  133,254     : REM  STA $FE
510 ::                  REM CIN
520 DATA  165,252     : REM  LDA $FC
530 DATA  197,254     : REM  CMP $FE
540 DATA  208,7       : REM  BNE $07
550 DATA  165,251     : REM  LDA $FB
560 DATA  197,253     : REM  CMP $FD
570 DATA  208,1       : REM  BNE $01
580 DATA  96          : REM  RTS
590 ::                  REM GREEN
600 DATA  160,0       : REM  LDY #$00
610 DATA  169,13      : REM  LDA #$0D
620 DATA  145,251     : REM  STA ($FB),Y
630 DATA  230,251     : REM  INC $FB
640 DATA  208,233     : REM  BNE $E9
650 DATA  230,252     : REM  INC $FC
660 DATA  56          : REM  SEC
670 DATA  176,228     : REM  BCS $E4
```

Line-by-line

A line-by-line description of Program 24 follows:

line 9∅	:	load accumulator with mask ∅∅∅∅1∅∅∅
line 1∅∅	:	force bit 3 to select 8196 as bit map start address
line 11∅	:	and program VIC Memory Control register
line 12∅	:	load accumulator with mask ∅∅1∅∅∅∅∅
line 13∅	:	force bit 5 to select bit map mode
line 14∅	:	and program CIC Control register
line 15∅	:	entry for bit map CLEAR-SCREEN-MEMORY routine
line 16∅	:	set up vector to point to screen start address $2∅∅∅
line 2∅∅	:	set up vector to point to screen end address $4∅3F
line 24∅	:	entry for IN
line 25∅	:	get high byte current address
line 26∅	:	is it same as high byte end address?
line 27∅	:	no, so branch to CLEAR
line 28∅	:	yes, get low byte current address
line 29∅	:	is it same as low byte end address
line 3∅∅	:	no, so branch to CLEAR
line 31∅	:	yes, all done jump to COLOUR
line 32∅	:	entry for CLEAR
line 33∅	:	initialize index
line 34∅	:	clear accumulator
line 35∅	:	clear byte of screen memory
line 36∅	:	increment low byte of current screen address
line 37∅	:	branch to IN if no carry over
line 38∅	:	increment high byte
line 39∅	:	set Carry flag
line 4∅∅	:	force branch to IN
line 42∅	:	entry for COLOUR
line 43∅	:	set up vector to point to start of colour memory
line 47∅	:	set up vector to point to end of colour memory
line 51∅	:	entry for CIN
line 52∅	:	get high byte of current address
line 53∅	:	is it the same as high byte end address?
line 54∅	:	no, branch to GREEN

line 55∅ : get low byte of current address
line 56∅ : is it the same as the low byte end address?
line 57∅ : no, branch to GREEN
line 58∅ : back to calling routine
line 59∅ : entry for GREEN
line 6∅∅ : clear indexing register
line 61∅ : get code for green into accumulator
line 62∅ : POKE it into colour memory
line 63∅ : increment low byte of current address
line 64∅ : branch to CIN if no carry over
line 65∅ : increment high byte
line 66∅ : set Carry flag
line 67∅ : and force branch to CIN

Appendix 1: 6510 Complete Instruction Set

ADC	Add with carry		NZCV
Address mode	*Op-code*	*Bytes*	*Cycles*
Immediate	$69	2	2
Zero page	$65	2	3
Zero page,X	$75	2	4
Absolute	$6D	3	4
Absolute,X	$7D	3	4 or 5
Absolute,Y	$79	3	4 or 5
(Indirect,X)	$61	2	6
(Indirect),Y	$71	2	5

AND	AND with accumulator		NZ
Address mode	*Op-code*	*Bytes*	*Cycles*
Immediate	$29	2	2
Zero page	$25	2	3
Zero page,X	$35	2	4
Absolute	$2D	3	4
Absolute,X	$3D	3	4 or 5
Absolute,Y	$39	3	4 or 5
(Indirect,X)	$21	2	6
(Indirect),Y	$31	2	5

ASL	Shift left		NZC
Address mode	*Op-code*	*Bytes*	*Cycles*
Accumulator	$ØA	1	2
Zero page	$Ø6	2	5
Zero page,X	$16	2	6
Absolute	$ØE	3	6
Absolute,X	$1E	3	7

BCC	Branch if C = Ø		Flags unaltered
Address mode	*Op-code*	*Bytes*	*Cycles*
Relative	$9Ø	2	3 or 2

BCS	Branch if C = 1		Flags unaltered
Address mode	*Op-code*	*Bytes*	*Cycles*
Relative	$BØ	2	3 or 2

BEQ	Branch if Z = 1		Flags unaltered
Address mode	*Op-code*	*Bytes*	*Cycles*
Relative	$FØ	2	3 or 2

BIT			Z,N,V
Address mode	*Op-code*	*Bytes*	*Cycles*
Zero page	$24	2	3
Absolute	$2C	3	4

BMI	Branch if N = 1		Flags unaltered
Address mode	*Op-code*	*Bytes*	*Cycles*
Relative	$3Ø	2	3 or 2

BNE	Branch if Z = Ø		Flags unaltered
Address mode	*Op-code*	*Bytes*	*Cycles*
Relative	$DØ	2	3 or 2

BPL	Branch if N = Ø		Flags unaltered
Address mode	*Op-code*	*Bytes*	*Cycles*
Relative	$1Ø	2	3 or 2

BRK	Break		B flag = 1
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	\$ØØ	1	7

BVC	Branch if V = Ø		Flags unaltered
Address mode	*Op-code*	*Bytes*	*Cycles*
Relative	\$5Ø	2	3 or 2

BVS	Branch if V = 1		Flags unaltered
Address mode	*Op-code*	*Bytes*	*Cycles*
Relative	\$7Ø	2	3 or 2

CLC	Clear Carry flag		C flag = Ø
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	\$18	1	2

CLD	Clear Decimal flag		D flag = ∅
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	$D8	1	2

CLI	Clear Interrupt flag		I flag = ∅
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	$58	1	2

CLV	Clear Overflow flag		V flag = ∅
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	$B8	1	2

CMP	Compare accumulator		NZC
Address mode	*Op-code*	*Bytes*	*Cycles*
Immediate	$C9	2	2
Zero page	$C5	2	3
Zero page,X	$D5	2	4
Absolute	$CD	3	4
Absolute,X	$DD	3	4 or 5
Absolute,Y	$D9	3	4 or 5
(Indirect,X)	$C1	2	6
(Indirect),Y	$D1	2	5 or 6

CPX	Compare X register		NZC
Address mode	*Op-code*	*Bytes*	*Cycles*
Immediate	$E0	2	2
Zero page	$E4	2	3
Absolute	$EC	3	4

CPY	Compare Y register		NZC
Address mode	*Op-code*	*Bytes*	*Cycles*
Immediate	$C0	2	2
Zero page	$C4	2	3
Absolute	$CC	3	4

DEC	Decrement memory		NZ
Address mode	*Op-code*	*Bytes*	*Cycles*
Zero page	$C6	2	5
Zero page,X	$D6	2	6
Absolute	$CE	3	6
Absolute,X	$DE	3	7

DEX	Decrement X register		NZ
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	$CA	1	2

DEY	Decrement Y register		NZ
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	$88	1	2

EOR	Exclusive-OR		NZ
Address mode	*Op-code*	*Bytes*	*Cycles*
Immediate	$ 49	2	2
Zero page	$ 45	2	3
Zero page,X	$ 55	2	4
Absolute	$ 4D	3	4
Absolute,X	$ 5D	3	4 or 5
Absolute,Y	$ 59	3	4 or 5
(Indirect,X)	$ 41	2	6
(Indirect),Y	$ 51	2	5

INC	Increment memory		NZ
Address mode	*Op-code*	*Bytes*	*Cycles*
Zero page	$E6	2	5
Zero page,X	$F6	2	6
Absolute	$EE	3	6
Absolute,X	$FE	3	7

INX	Increment X register		NZ
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	$E8	1	2

INY	Increment Y register		NZ
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	$C8	1	2

JMP	Jump		Flags unaltered
Address mode	*Op-code*	*Bytes*	*Cycles*
Absolute	$4C	3	3
Indirect	$6C	3	5

JSR	Jump to subroutine		Flags unaltered
Address mode	*Op-code*	*Bytes*	*Cycles*
Absolute	$2Ø	3	6

LDA	Load accumulator		NZ
Address mode	*Op-code*	*Bytes*	*Cycles*
Immediate	$A9	2	2
Zero page	$A5	2	3
Zero page,X	$B5	2	4
Absolute	$AD	3	4
Absolute,X	$BD	3	4 or 5
Absolute,Y	$B9	3	4 or 5
(Indirect,X)	$A1	2	6
(Indirect),Y	$B1	2	5 or 6

LDX	Load X register		NZ
Address mode	*Op-code*	*Bytes*	*Cycles*
Immediate	$A2	2	2
Zero page	$A6	2	3
Zero page,Y	$B6	2	4
Absolute	$AE	3	4
Absolute,Y	$BE	3	4 or 5

LDY	Load Y register		NZ
Address mode	*Op-code*	*Bytes*	*Cycles*
Immediate	$AØ	2	2
Zero page	$A4	2	3
Zero page,X	$B4	2	4
Absolute	$AC	3	4
Absolute,X	$BC	3	4 or 5

LSR	Logical shift right		N = ∅,ZC
Address mode	*Op-code*	*Bytes*	*Cycles*
Accumulator	$4A	1	2
Zero page	$46	2	5
Zero page,X	$56	2	6
Absolute	$4E	3	6
Absolute,X	$5E	3	7

NOP	No operation		Flags unaltered
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	$EA	1	2

ORA	Inclusive OR		NZ
Address mode	*Op-code*	*Bytes*	*Cycles*
Immediate	$∅9	2	2
Zero page	$∅5	2	3
Zero page,X	$15	2	4
Absolute	$∅D	3	4
Absolute,X	$1D	3	4 or 5
Absolute,Y	$19	3	4 or 5
(Indirect,X)	$∅1	2	6
(Indirect),Y	$11	2	5

PHA	Push accumulator		Flags unaltered
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	\$48	1	3

PHP	Push Status register		Flags unaltered
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	\$Ø8	1	3

PLA	Pull accumulator		NZ
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	\$68	1	4

PLP	Pull Status register		Flags as status
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	\$28	1	4

ROL	Rotate left		NZC
Address mode	*Op-code*	*Bytes*	*Cycles*
Accumulator	$2A	1	2
Zero page	$26	2	5
Zero page,X	$36	2	6
Absolute	$2E	3	6
Absolute,X	$3E	3	7

ROR	Rotate right		NZC
Address mode	*Op-code*	*Bytes*	*Cycles*
Accumulator	$6A	1	2
Zero page	$66	2	5
Zero page,X	$76	2	6
Absolute	$6E	3	6
Absolute,X	$7E	3	7

RTI	Return from interrupt		Flags as pulled
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	$4Ø	1	6

RTS	Return from subroutine		Flags unaltered
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	\$60	1	6

SBC	Subtract from accumulator		NZCV
Address mode	*Op-code*	*Bytes*	*Cycles*
Immediate	\$E9	2	2
Zero page	\$E5	2	3
Zero page,X	\$F5	2	4
Absolute	\$ED	3	4
Absolute,X	\$FD	3	4 or 5
Absolute,Y	\$F9	3	4 or 5
(Indirect,X)	\$E1	2	6
(Indirect),Y	\$F1	2	5 or 6

SEC	Set Carry flag		C = 1
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	\$38	1	2

SED	Set Decimal flag		D = 1
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	\$F8	1	2

SEI	Set Interrupt flag		I = 1
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	$78	1	2

STA	Store accumulator		Flags unaltered
Address mode	*Op-code*	*Bytes*	*Cycles*
Zero page	$85	2	3
Zero page,X	$95	2	4
Absolute	$8D	3	4
Absolute,X	$9D	3	5
Absolute,Y	$99	3	5
(Indirect,X)	$81	2	6
(Indirect),Y	$91	2	6

STX	Store X register		Flags unaltered
Address mode	*Op-code*	*Bytes*	*Cycles*
Zero page	$86	2	3
Zero page,Y	$96	2	4
Absolute	$8E	3	4

STY	Store Y register		Flags unaltered
Address mode	*Op-code*	*Bytes*	*Cycles*
Zero page	$84	2	3
Zero page,X	$94	2	4
Absolute	$8C	3	4

TAX	Transfer accumulator to X		NZ
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	$AA	1	2

TAY	Transfer accumulator to Y		NZ
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	$A8	1	2

TSX	Transfer Stack Pointer to X		NZ
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	$BA	1	2

TXA	Transfer X to accumulator		NZ
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	$8A	1	2

TXS	Transfer X to Stack Pointer		Flags unaltered
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	$9A	1	2

TYA	Transfer Y to accumulator		NZ
Address mode	*Op-code*	*Bytes*	*Cycles*
Implied	$98	1	2

Appendix 2: 6510 Opcodes

All numbers are hexadecimal.

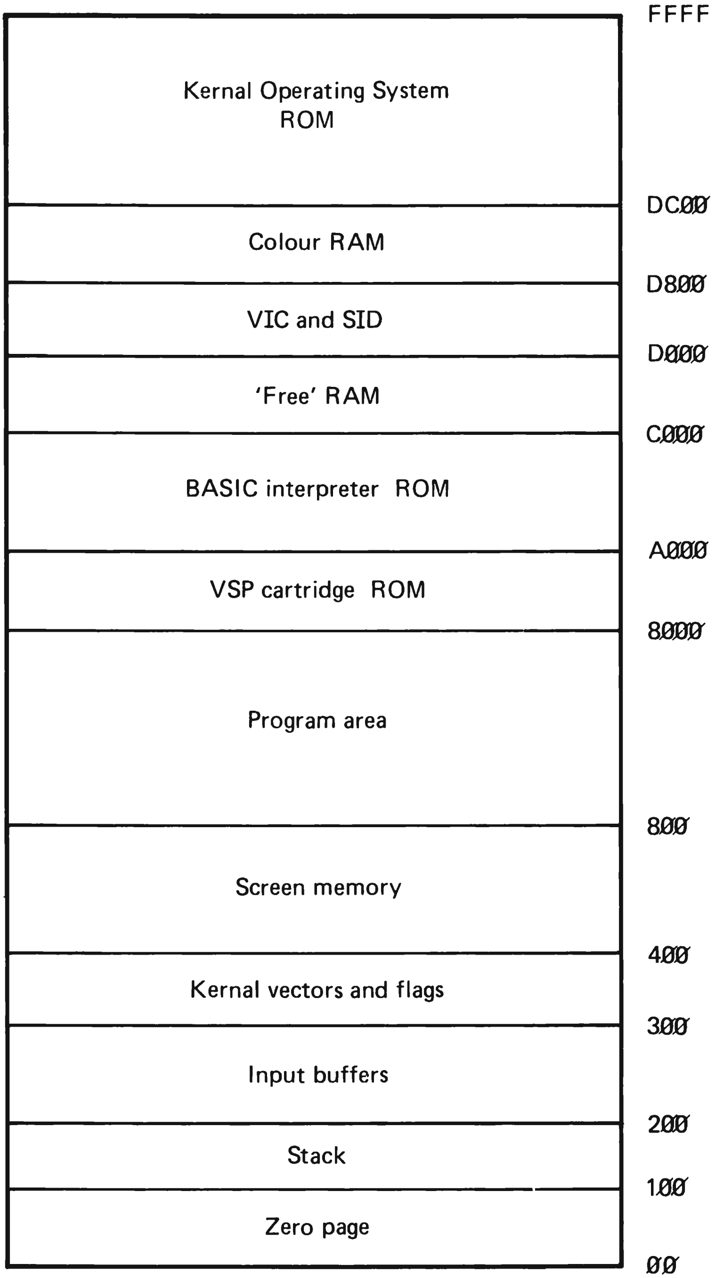
00 BRK implied
01 ORA (zero page, X)
02 Future expansion
03 Future expansion
04 Future expansion
05 ORA zero page
06 ASL zero page
07 Future expansion
08 PHP implied
09 ORA #immediate
0A ASL accumulator
0B Future expansion
0C Future expansion
0D ORA absolute
0E ASL absolute
0F Future expansion
10 BPL relative
11 ORA (zero page), Y
12 Future expansion
13 Future expansion
14 Future expansion
15 ORA zero page, X
16 ASL zero page, X
17 Future expansion
18 CLC implied
19 ORA absolute, Y
1A Future expansion
1B Future expansion
1C Future expansion
1D ORA absolute, X
1E ASL absolute, X
1F Future expansion
20 JSR absolute
21 AND (zero page, X)
22 Future expansion
23 Future expansion
24 BIT zero page
25 AND zero page
26 ROL zero page
27 Future expansion
28 PLP implied
29 AND #immediate
2A ROL accumulator
2B Future expansion
2C BIT absolute
2D AND absolute
2E ROL absolute
2F Future expansion
30 BMI relative
31 AND (zero page), Y
32 Future expansion
33 Future expansion
34 Future expansion
35 AND zero page, X
36 ROL zero page, X
37 Future expansion

38	SEC implied
39	AND absolute, Y
3A	Future expansion
3B	Future expansion
3C	Future expansion
3D	AND absolute, X
3E	ROL absolute, X
3F	Future expansion
40	RTI implied
41	EOR (zero page, X)
42	Future expansion
43	Future expansion
44	Future expansion
45	EOR zero page
46	LSR zero page
47	Future expansion
48	PHA implied
49	EOR #immediate
4A	LSR accumulator
4B	Future expansion
4C	JMP absolute
4D	EOR absolute
4E	LSR absolute
4F	Future expansion
50	BVC relative
51	EOR (zero page), Y
52	Future expansion
53	Future expansion
54	Future expansion
55	EOR zero page, X
56	LSR zero page, X
57	Future expansion
58	CLI implied
59	EOR absolute, Y
5A	Future expansion
5B	Future expansion
5C	Future expansion
5D	EOR absolute, X
5E	LSR absolute, X
5F	Future expansion
60	RTS implied
61	ADC (zero page, X)
62	Future expansion
63	Future expansion
64	Future expansion
65	ADC zero page
66	ROR zero page
67	Future expansion
68	PLA implied
69	ADC #immediate
6A	ROR accumulator
6B	Future expansion
6C	JMP (indirect)
6D	ADC absolute
6E	ROR absolute
6F	Future expansion
70	BVS relative
71	ADC (zero page), Y
72	Future expansion
73	Future expansion
74	Future expansion
75	ADC zero page, X
76	ROR zero page, X
77	Future expansion
78	SEI implied
79	ADC absolute, Y
7A	Future expansion
7B	Future expansion
7C	Future expansion
7D	ADC absolute, X
7E	ROR absolute, X
7F	Future expansion
80	Future expansion
81	STA (zero page, X)

82 Future expansion
83 Future expansion
84 STY zero page
85 STA zero page
86 STX zero page
87 Future expansion
88 DEY implied
89 Future expansion
8A TXA implied
8B Future expansion
8C STY absolute
8D STA absolute
8E STX absolute
8F Future expansion
90 BCC relative
91 STA (zero page), Y
92 Future expansion
93 Future expansion
94 STY zero page, X
95 STA zero page, X
96 STX zero page, Y
97 Future expansion
98 TYA implied
99 STA absolute, Y
9A TXS implied
9B Future expansion
9C Future expansion
9D STA absolute, X
9E Future expansion
9F Future expansion
A0 LDY #immediate
A1 LDA (zero page, X)
A2 LDX #immediate
A3 Future expansion
A4 LDY zero page
A5 LDA zero page
A6 LDX zero page
A7 Future expansion
A8 TAY implied
A9 LDA #immediate
AA TAX implied
AB Future expansion
AC LDY absolute
AD LDA absolute
AE LDX absolute
AF Future expansion
B0 BCS relative
B1 LDA (zero page), Y
B2 Future expansion
B3 Future expansion
B4 LDY zero page, X
B5 LDA zero page, X
B6 LDX zero page, Y
B7 Future expansion
B8 CLV implied
B9 LDA absolute, Y
BA TSX implied
BB Future expansion
BC LDY absolute, X
BD LDA absolute, X
BE LDX absolute, Y
BF Future expansion
C0 CPY #immediate
C1 CMP (zero page, X)
C2 Future expansion
C3 Future expansion
C4 CPY zero page
C5 CMP zero page
C6 DEC zero page
C7 Future expansion
C8 INY implied
C9 CMP #immediate
CA DEX implied
CB Future expansion

CC CPY absolute
CD CMP absolute
CE DEC absolute
CF Future expansion
D0 BNE relative
D1 CMP (zero page), Y
D2 Future expansion
D3 Future expansion
D4 Future expansion
D5 CMP zero page, X
D6 DEC zero page, X
D7 Future expansion
D8 CLD implied
D9 CMP absolute, Y
DA Future expansion
DB Future expansion
DC Future expansion
DD CMP absolute, X
DE DEC absolute, X
DF Future expansion
E0 CPX #immediate
E1 SBC (zero page, X)
E2 Future expansion
E3 Future expansion
E4 CPX zero page
E5 SBC zero page
E6 INC zero page
E7 Future expansion
E8 INX implied
E9 SBC #immediate
EA NOP implied
EB Future expansion
EC CPX absolute
ED SBC absolute
EE INC absolute
EF Future expansion
F0 BEQ relative
F1 SBC (zero page), Y
F2 Future expansion
F3 Future expansion
F4 Future expansion
F5 SBC zero page, X
F6 INC zero page, X
F7 Future expansion
F8 SED implied
F9 SBC absolute, Y
FA Future expansion
FB Future expansion
FC Future expansion
FD SBC absolute, X
FE INC absolute, X
FF Future expansion

Appendix 3: Commodore 64 Memory Map



Appendix 4: Branch Calculators

The branch calculators are used to give branch values in hex. First, count the number of bytes you need to branch. Then locate this number in the centre of the appropriate table, and finally, read off the high and low hex nibbles from the side column and top row respectively.

Example For a backward branch of 16 bytes:

Locate 16 in the centre of Table A4.1 (bottom row), then read off high nibble (#F) and low nibble (#0) to give displacement value (#F0).

Table A4.1 Backward branch calculator

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Table A4.2 Forward branch calculator

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

Index

Other titles of interest

Easy Programming for the Commodore 64 **£6.95**
Ian Stewart & Robin Jones
An intoductory guide to BASIC programming.

Commodore 64 Assembly Language **£7.95**
Bruce Smith

The Commodore 64 Music Book **£5.95**
James Vogel & Nevin B. Scrimshaw

Commodore 64 Machine Codew **£6.95**
Ian Stewart & Robin Jones
'An excellent introduction to the subject'—*Popular Computing Weekly*

Gateway to Computing with the Commodore 64
Ian Stewart
'Recommended'—*Popular Computing Weekly*

Book One	**£4.95(p)**	**£6.95(h)**
Book Two	**£4.95(p)**	**£6.95(h)**

Computers in a Nutshell **£4.95**
Ian Stewart

Microchip Mathematics: Number Theory for Computor Users **£12.95**
Keith Devlin
A fascinating book about the interaction of mathematics and computing.

Brainteasers for BASIC Computers **£4.95**
Gorden Lee
'A book I would warmly recommend'—*Computer & Video Games*

Commodore 64 Assembler Workshop provides the experienced machine code programmer with a bench full of programming tools and techniques.

The programs include utilities for:

- comparing, copying, deleting, inserting and storing strings
- implementing a software stack
- utilizing hi-res graphics procedures

Add your own commands to the Commodore 64 vocabulary with the Wedge Interpreter. Find out how simple base conversion can be, using routines that convert binary to decimal, and vice versa.

As each program is presented in the form of a BASIC loader, it is not necessary to own an expensive assembler to benefit from using the book. Full mnemonic descriptions are included for those who have assemblers, and detailed descriptions of each program ensure that its operation is easy to follow and to understand.

Sharpen up your Commodore 64 with the tools in the **Assembler Workshop!**

UK pri... et

GB £ NET
ISBN 1-85014-004-9
00695
9 781850 140047

SHIVA
Shiva Publishing Limited