Pan

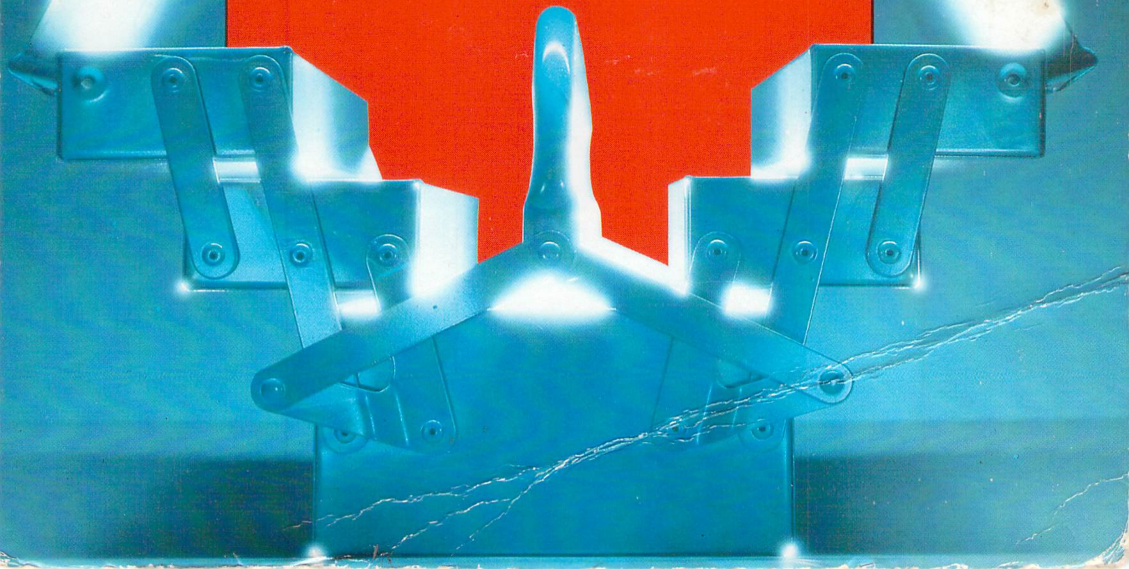
PERSONAL COMPUTER

COMPUTER NEWS LIBRARY

CLIVE EMBEREY & BOB TURNER

INVALUABLE UTILITIES for the COMMODORE 64

'The complete programmer's toolkit – essential programming aids for your micro'



Pan/Personal Computer News
Computer Library

Clive Emberey and Bob Turner

Invaluable Utilities for the Commodore 64

Pan Books London and Sydney

First published 1984 by Pan Books Ltd,
Cavaye Place, London SW10 9PG
in association with Personal Computer News
9 8 7 6 5 4 3 2 1

ISBN 0 330 28671 4
Photoset by Parker Typesetting Service, Leicester
Printed and bound in Great Britain by
Richard Clay (The Chaucer Press) Ltd, Bungay, Suffolk

We wish to thank the following people, to whom we dedicate this book:

Our nearest and dearest who remained (fairly) tolerant throughout.

Keith Bowden and members of the 64 IUC who continued to run the club in our leave of absence.

Dave Proctor for patiently testing much of the work.

And PAN/PCN for publishing the book.

Contents

Introduction

This book as the title suggests, is a book of utilities for the Commodore 64. It has been written not only to provide a set of useful routines, but also to help you to begin to understand some of the more detailed workings of your 64.

We have tried to cover a reasonable spectrum and hope that through our examples you will attack areas other than those covered here with increased confidence. Towards this end we have covered in depth the development, background and implementation of each utility.

We have made no attempt to cover programming, in either BASIC or machine code, in this book because many other texts cover this in detail. We have also assumed that most serious 64 users will be in possession of a copy of the *Programmer's Reference Guide* and have adopted its nomenclature throughout, particularly with reference to memory locations and KERNAL routines.

Wherever possible a utility has been implemented in both BASIC and machine code. We felt that the BASIC versions, though sometimes crude, are easier to experiment with and should also help those readers unfamiliar with machine code to appreciate the workings of the equivalent routines. They also go to prove that it is not what you know, but how you use it. In some cases it might prove beneficial to use the BASIC rather than the machine code versions. Typical circumstances might be where only one or two features are required, or when you need the full 38K of RAM available to BASIC, or if you wish to switch to bank 2 when using graphics.

To facilitate entering the machine code it has been given in two forms: as BASIC loaders and assembler listings. The assembler listings are suitable for use with an extended monitor. For anyone not owning a monitor program, we have included Jim Butterfield's Supermon and instructions for its use in Appendices D and E. (Supermon for the 64 was first published in the January 1983 issue of *Compute*). However, before attempting the considerable task of typing it in and then getting it to work, ask around your friends and user groups as they may have a copy. If you do find one, you will save yourself a lot of time, effort and frustration. Jim Butterfield has also published a very complete Memory Map for the 64 in the October 1982 issue of *Torpet* which has since appeared in many other journals. This complements the one in the *Programmer's Reference Guide* as it gives the nominal entry points to most ROM routines. A copy of this map could save you a great deal of time when disassembling ROM routines to find out how they work.

To assist in entering the BASIC code all listings have been provided in an annotated form. This, we hope, will avoid the all-too-common problems associated with deciphering the symbols for cursor keys, function keys, colours, and so on when in quotes mode. A detailed list of all mnemonics used is given in Appendix H, but you should find that most are self-explanatory. The program we wrote to generate them is included in this book in the UTILITY as the CODER command. The listings as given in the text always have a maximum line length of 40 characters in their annotated form. Where a line exceeds forty characters it is continued on the next and subsequent lines always commencing in column 1. When looking at the listings you may find it helpful to compare the rightmost characters of continued lines. When the code is typed in, replacing the mnemonics with the correct key(s), no line will exceed 80 characters on the display.

Any of the BASIC utilities intended for use from within another program have been numbered in the 60000's to allow you to merge them with your own programs (using the simple technique described in Chapter 5 or the MERGE command of the UTILITY itself).

We have chosen to put our code at 32768 ($8000), leaving the BASIC programmer with 30K free. There is no reason why the code could not be modified and relocated elsewhere in memory (the 4K block from $C000 is not a bad idea) and the initialization routine adjusted to take advantage of the increased memory available. In fact, nearly all the routines were developed and tested in isolation, being enabled by a simple SYS call. They were then incorporated into the UTILITY by simply relocating and including a keyword and token to activate them. To conserve memory, common subroutines have not been duplicated. Often a pick 'n' mix approach was found useful to check out a range of extensions which relied heavily on common routines.

We have used 'hidden' RAM beneath BASIC to store data to conserve valuable user RAM and implemented a simple switching routine to access this data when necessary. Applications like setting up the function keys require access to this RAM, as does CODER. We have made extensive use of the ROM routines and RAM vectors available, but on some occasions found it more economic and faster to write our own code. The UTILITY, in the form given, occupies the same area as cartridge ROM and cannot therefore co-exist with cartridges. It was not written to run in conjunction with them and is intended as a standalone, extendable facility. As the owners of a disk unit will have received DOS 5.1. on the demo disk, the UTILITY has been written to co-exist with DOS 5.1. Some of the isolated routines will temporarily disable DOS as they make use of the same operating routine – CHRGET – but more about that later. However, a simple SYS call will restore DOS 5.1 commands.

There are many commercial utilities and BASIC extensions. These may be purchased at reasonable prices and for many applications there will be no better solution. However, if you are interested in the Commodore 64 and wish to get the most from it, you may appreciate having a range of routines which you can modify, extend and, indeed, improve upon. After all, you can pay upwards of the cost of this book for a fairly simple renumber routine.

Before we finish, we would like to leave you with two suggestions and an option:

1) Always save a program before running it
2) Always make backup copies

This is good advice for BASIC and essential where machine code is concerned.

3) It is very easy to wire a reset switch to your 64 and the necessary reset line is available at both the Serial I/O and User I/O (see the *Programmer's Reference Guide*, Appendix I). This is almost essential if you use machine code, but *don't attempt this if you are not sure what you are doing.*

Have fun (if that's the right word)!

1 BASIC on the 64

Introducing BASIC

On powering up your 64 you will find it ready and waiting to go in BASIC, as part of the power reset sequence is to initialize BASIC and leave the user in direct mode.

The implementation of BASIC that Commodore has chosen to use for the 64 is identical to that on the VIC20 and PET microcomputers prior to the 4000 series. BASIC 2, as it is often called, differs from the later version only in its disk operating commands, the latter having a greatly improved and simplified instruction set for disk control. In producing BASIC 4 Commodore did maintain 99.99% downward compatibility and in doing so allowed users to run any program on a higher series PET. It was, therefore, a little surprising to find the BASIC on the 64 to be only V2. This may have been done to avoid a conflict of interests in so much as the new CBM micro, though in many respects far more powerful, was not quite the same.

BASIC, or to give its full title Beginners' All-purpose Symbolic Instruction Code, runs on the 64 as a high-level interpreted language. It is a subset of Microsoft's BASIC (who wrote the first implementation for the early PETs and now produce MBASIC and the MSDOS operating system for all major microcomputers). The history of BASIC is nothing to do with this book, but it is interesting to note that regardless of environment, or cost of system BASIC will usually be in there somewhere. It may only run in compiled form, or it may be syntactically different, but it is reassuring to know that a knowledge of 64 BASIC should allow you to grasp quickly other BASICS on other machines.

BASIC has its critics, particularly of CBM BASIC, who would advocate the use of Pascal, or Pilot, or Forth, or . . . Each of these languages is particularly suited to a range of tasks, but perhaps none lends itself as well as BASIC to the task of rapid development of 'untidy' and 'unstructured' programs which, most importantly, work. Arguments for and against will no doubt long continue, but as we are supplied with BASIC, let us make the most of it.

As its name imples, BASIC was developed to allow beginners to acquire programming skills rapidly. It adopted a system of naming its commands and functions to indicate the action produced. For example, if we wish to halt the execution of a program we issue the all-too-clear command : STOP. For non English-speaking countries even the use of English is no problem on the 64 as not only is it a simple matter to redefine the character set, but it is also easy to redefine the keyword table itself.

On the 64 we have 75 (76 if you include GO) BASIC commands, functions and operators as standard. For many applications this is perfectly adequate. Life would be simpler if more commands were available. Increasing the number of commands has the benefit of providing a more versatile programming language, but the disadvantage of slowing down the execution of the existing commands. This is true of any interpreted programming language. However, the way in which the interpreter has been implemented does allow you to add to the keywords to your heart's content, providing you understand how it works and are capable of writing the necessary machine code. For the moment the interpreter will be considered simply as a means of translating our 'meaningless entries' into something which is executable by the 6510 microprocessor at the machine code level. This it does by taking an instruction, finding the appropriate machine code routine, carrying it out and then returning to implement the next. The process is slow but very flexible and even allows us to interrupt the execution and take control should we wish to do so.

One of the best features of the 64 must be the screen editor. It allows changes to be made directly to anything appearing on the screen and, more importantly, allows you to implement these changes. The disadvantage is that the maximum length of program line or direct statement is limited to 80 characters (or two screen lines). Use of the standard abbreviations of first character and second (or third) character shifted, instead of typing the full keyword, does allow program lines, on listing, to exceed this limit. They cannot, however, be easily edited. On pressing RETURN to acknowledge the end of the edit only two screen lines are accepted. Anything beyond this point is not included in the revised line. Still, this limitation is far outweighed by the speed at which it allows existing code to be edited and repetitive code to be entered by simply using the cursor keys, altering the line number, modifying the necessary part of the line and pressing RETURN to enter the new line. It even allows us to write programs which can generate their own program lines, as we will see in Chapter 5.

BASIC may be used in two modes. These are direct (when a command is typed in without a line number and executed immediately) and program (a command preceded by a line number which is not executed until the program is RUN).

Storage of BASIC code

If we wish to examine a program, we may do so with the LIST command. What we see has undergone many processes from the form in which it was stored in RAM. To view the code *in situ* we first of all need to know where to look. In the default mode on powering-up a BASIC program will be sorted from memory location 2049 ($0801) upwards. We can examine a program by simply PEEKing out each of the locations used by

```
FOR I=START TO START+200:PRINT PEEK (I);:NEXT I
```

We would see a series of decimal numbers with only the fact in common that none was less than zero or greater than 255. We might also notice some sort of related pattern occurring, but not a great deal more. We could adopt another approach by moving an area of RAM used for program storage to the screen. This is easily accomplished, but in doing so we must also remember to set a colour at the screen location we are putting the data into for it to be visible. The resulting display is easier to decipher if the 64 is put in lower case mode by pressing the shift and logo keys together. The following line should be typed in direct mode:

```
S=0:FOR I=START TO START+800:POKE 1024+S,PEEK(I):POKE
55296+S,14:S=S+1:NEXT I
```

If you wish to start at the beginning of a program then 2049 must be used and it assumes, as does the first example, a program to be present which occupies memory at least to START+800 (or +200). This time we see a series of characters and where our program has text within quotes it appears almost unchanged as do variable names, punctuation and constants. If we combine the processes, and to produce a more consistent format express the numbers in hexadecimal format, we begin to see some sort of relationship. (You had better get used to using hexadecimal notation as we use it extensively, but to help you on your way there is a table of decimal to hex conversions in Appendix B). The following program does just this and may be used to examine itself. If you wish to experiment, simply enter new lines with numbers less than 60000. Those of you with extended monitors or who jumped straight in and typed in Supermon can use the 'memory display' option.

The program displays on each line the start address and the values held in this and the next seven locations. At the right of the line the characters with ASCII (CHR$()) codes corresponding to the byte values are printed. To avoid confusion, only those characters which are easily discernible are printed; all others are expressed by a ".". Appendix C of the PRG gives the full range of ASCII and CHR$() codes. If you want to display all the characters then some of the codes will have effects which will destroy the display, for example, cursor moves, clear screen, colours, and so on; so you will have to trap these. They do, however, occur in blocks and are therefore not too difficult to isolate.

As will be standard practice throughout this book, a description precedes most program listings.

LINE	ACTION
130	Examine selected range in groups of eight.
140	Convert current start address to low/high byte format, that is, units (0–255) and lots of 256s. Then convert to hex notation in two stages.
150 to 170	Get eight successive bytes from start and print two digit hex values each time.
190 to 220	Convert eight bytes to ASCII characters if printable. Else replace with a '.' and build eight character string.
230	Print string. Recycle if not end else start again.
1000	Convert start address to hex in two steps.
2000	Convert byte to two digit hex.

```
100 PRINT"MEMORY DISPLAY"
110 INPUT"DISPLAY FROM";F
120 INPUT"[10SPC]TO";T:IF T<F THEN PRINT
 "LESS THAN FROM":GOTO 120
130 FOR I=F TO T STEP 8
140 X=I:GOSUB 1000
150 FOR J=I TO I+7
160 X=PEEK(J):GOSUB 2000:PRINT X$;"  ";
170 NEXT J
180 PRINT "  ";
190 A$="":FOR J=I TO I+7
200 X=PEEK(J):IF X<32 OR X>95 THEN A$=A$
+".":GOTO 220
210 A$=A$+CHR$(X)
220 NEXT J
230 PRINT A$:NEXT I:GOTO 100
1000 MSB=INT(X/256):LSB=I-MSB*256
1010 X=MSB:GOSUB 2000:PRINT X$;
1020 X=LSB:GOSUB 2000:PRINT X$;"  ";
1030 RETURN
2000 X1=INT(X/16):X2=X-X1*16
2010 X$=CHR$(X1+48-7*(X1>9))+CHR$(X2+48-
7*(X2>9))
2020 RETURN
```

If the program is used to examine itself by entering a start of 2048 and an end of 2504 for the program as listed the following display is given:

```
0970 20 32 30 30 30 3A 99 20    2000:.
0978 58 24 3B 22 20 22 3B 00   X$;" ";.
0980 86 09 06 04 8E 00 A0 09   ........
0988 D0 07 58 31 B2 B5 28 58   ..X1..(X
0990 AD 31 36 29 3A 58 32 B2   .16):X2.
0998 58 AB 58 31 AC 31 36 00   X.X1.16.
09A0 CB 09 DA 07 58 24 B2 C7   ....X$..
09A8 28 58 31 AA 34 38 AB 37   (X1.48.7
09B0 AC 28 58 31 B1 39 29 29   .(X1.9))
09B8 AA C7 28 58 32 AA 34 38   ..(X2.48
09C0 AB 37 AC 28 58 32 B1 39   .7.(X2.9
09C8 29 29 00 D1 09 E4 07 8E   ))......
09D0 00 00 00                  ...
```

Its exact format will vary depending on how you typed the program in. A number of things are immediately apparent. All text inside quotes, all variable names, all destinations, all constants and punctuation appear unchanged. From just this information we can work out the general area of each line. Taking line 100 as an example, "MEMORY. . ." is clearly visible from \$0806 to \$0815. Immediately preceding it is the value \$99 which, not surprisingly, is the tokenized value for PRINT. The two bytes before this are \$64 and \$00. \$64 is the hex for 100 which is the line number. Line numbers may range from 0 up to 63999 and, like many values on the 64, are stored in low/high byte format. The actual line number is \$64+\$00*\$0100 (100+0*256). If we look along the hex values for line 100, we see that the byte immediately following the closing quote is a zero. This is how BASIC marks the end of a program line. The two bytes preceding the line number are \$17/\$08 which is the address (low/high) of the byte immediately following this end of line zero. These two bytes are known as the link address and point to the link address (and start) of the next line. If we follow the link address through the program, the sequence runs \$0817/082D/0864....09D1 and finally 0000. A link address of zero marks the end of the program, which in this case is \$09D1. A pointer to this address+2 is held in zero page (locations \$00=\$FF) at VARTAB (\$2D/2E) and marks the start of the BASIC variables. A second pointer on zero page, TXTTAB (\$2B/2C), points to the start of the program. This is the location of the first link address and in the default setting this will always be \$0801. Location \$0800 holds a zero; the byte before the start of a program must always be zero for RUN to work. This becomes of more significance when the start location of BASIC is changed.

A program can therefore be thought of as a 'linked list' of individual program lines. It is of the form:

START	LINK	LINE		END	....	END PROG
00	low high	low high	BASIC line	00	low high....	00 00 00

```
0800 00 17 08 64 00 99 22 4D  ......"M
0808 45 4D 4F 52 59 20 44 49  EMORY DI
0810 53 50 4C 41 59 22 00 2D  SPLAY".-
0818 08 6E 00 85 22 44 49 53  ...."DIS
0820 50 4C 41 59 20 46 52 4F  PLAY FRO
0828 4D 22 3B 46 00 64 08 78  M";F....
0830 00 85 22 20 20 20 20 20  .."
0838 20 20 20 20 20 54 4F 22       TO"
0840 3B 54 3A 8B 20 54 B3 46  ;T:. T.F
0848 20 A7 20 99 20 22 4C 45   . . "LE
0850 53 53 20 54 48 41 4E 20  SS THAN
0858 46 52 4F 4D 22 3A 89 20  FROM":.
0860 31 32 30 00 76 08 82 00  120.....
0868 81 20 49 B2 46 20 A4 20  . I.F .
0870 54 20 A9 20 38 00 85 08  T . 8...
0878 8C 00 58 B2 49 3A 8D 20  ..X.I:.
0880 31 30 30 30 00 95 08 96  1000....
0888 00 81 20 4A B2 49 20 A4  .. J.I .
0890 20 49 AA 37 00 B1 08 A0   I.7....
0898 00 58 B2 C2 28 4A 29 3A  .X..(J):
08A0 8D 20 32 30 30 30 3A 99  . 2000:.
08A8 20 58 24 3B 22 20 22 3B   X$;" ";
08B0 00 B9 08 AA 00 82 20 4A  ...... J
08B8 00 C4 08 B4 00 99 20 22  ...... "
08C0 20 22 3B 00 DA 08 BE 00   ";.....
08C8 41 24 B2 22 22 3A 81 20  A$."":.
08D0 4A B2 49 20 A4 20 49 AA  J.I . I.
08D8 37 00 05 09 C8 00 58 B2  7.....X.
08E0 C2 28 4A 29 3A 8B 20 58  .(J):. X
08E8 B3 33 32 20 B0 20 58 B1  .32 . X.
08F0 39 35 20 A7 20 41 24 B2  95 . A$.
08F8 41 24 AA 22 2E 22 3A 89  A$.".":.
0900 20 32 32 30 00 14 09 D2   220....
0908 00 41 24 B2 41 24 AA C7  .A$.A$..
0910 28 58 29 00 1C 09 DC 00  (X).....
0918 82 20 4A 00 2F 09 E6 00  . J./...
0920 99 20 41 24 3A 82 20 49  . A$:. I
0928 3A 89 20 31 30 30 00 4E  :. 100.N
0930 09 E8 03 4D 53 42 B2 B5  ...MSB..
0938 28 58 AD 32 35 36 29 3A  (X.256):
0940 4C 53 42 B2 49 AB 4D 53  LSB.I.MS
0948 42 AC 32 35 36 00 65 09  B.256...
0950 F2 03 58 B2 4D 53 42 3A  ..X.MSB:
0958 8D 20 32 30 30 30 3A 99  . 2000:.
0960 20 58 24 3B 00 80 09 FC   X$;....
0968 03 58 B2 4C 53 42 3A 8D  .X.LSB:.
```

The link addresses are not used when a program is run, but are important during listing and editing. We can alter their values without affecting the way a program runs, but on listing some strange effects are produced.

We can now look through the display and find the start and end of a line and the associated line number. Knowing these, we can start to deduce the tokenized values for the BASIC keywords used. By adding lines to the program we could find out all keyword values, but to save you the effort we have produced a complete list in Appendix A. This table has been extended to include the new token values used by the UTILITY. These values should be ignored for the moment. With a little practice, reading displays of this type becomes very easy.

From the table in Appendix A we see that all BASIC keywords have token values in excess of 127 ($7F). The highest token used for standard BASIC is 203 ($CB) for the GO command. (GO simply searches for a corresponding TO to ensure GOTO is equivalent to GO TO). When a line is entered from the keyboard it is transferred to the input buffer (BUF $0200–0258) on pressing RETURN. The line is then tokenized in accordance with this table with keywords being processed first. If no line number is present, the BASIC interpreter immediately executes the statement(s). If one is present then the line is put in its numerically correct position and the link addresses for the whole program are recalculated and VARTAB updated. A similar process is carried out when a line is deleted. These operations are discussed in greater detail in Chapter 3.

Variables

General

BASIC allows three types of variable. These are real, integer and string. String and integer are distinguished from real by trailing '$' and '%' characters respectively. The default is therefore to real. Variable names may be of any length, but only the first two characters and the last character are of significance. This means ABXXXX% will be considered equal to ABYYY% and to AB%, but different from ABXXXX or ABXXXX$. The last character is used to distinguish the variable's type, and if it is not one of the special characters above then the variable is treated as real. The only limitation on naming variables is that the first character must be alphabetic and the subsequent character, alphanumeric, providing that they do not form reserved keywords. For example, PEND would be treated as 'P' plus the keyword END and on running would produce a syntax error. Reserved words are any of those occurring in Appendix A with a token value exceeding 127 (bearing in mind that with the UTILITY in place the number of reserved words will be increased).

To allow each of the three types of variable (four, really, when we include function names) to be stored in two bytes, the high bit is set or unset on each of the name bytes to give the necessary four combinations. These are:

Type	Name: 1st char	Name: 2nd char
REAL	ASCII	ASCII
INTEGER	ASCII+128	ASCII+128
STRING	ASCII	ASCII+128
FUNCTION	ASCII+128	ASCII

Where the name is only a single character, the second byte is zero or 128 as appropriate.

Each of the three types of variable may be used in multi-dimensional arrays. These subscripted variables follow the same rules as for simple variables with the addition of a '(' following the name and type. This tells the interpreter it is dealing with an array and is handled accordingly.

Storage of variables

Any variable created in either direct or program mode is stored after the program currently in memory. Variables are stored in the order in which they are created. Strings are stored slightly differently from numeric values mainly due to their dynamic nature and are held in two parts. The first is a pointer to the string's location and the second is the length of the string itself. Strings are stored at the top of BASIC memory ($9FFF/40959) and grow downwards. The current lower limit of string storage is stored in FRETOP ($33–$34/51–52).

When searching for a variable the interpreter starts at the end of the program and searches upwards in memory for the named value according to the rules given above. If the variable does not exist, the next available space is allocated to it. Thus, if we define the more important values early on they can be accessed quicker and the time spent in 'garbage collection' reduced.

Variables are either simple or subscripted.

Simple variables

All non-subscripted variables use seven bytes of RAM. The first two hold the name in its adjusted form. For each real variable the remaining five bytes are used in the following way: one for its exponent and the remaining four for its sign and mantissa. Integers are stored in only two bytes with the remaining three unused. Strings use one byte to indicate the length and two bytes to point to the location of the characters, which is usually at the top of memory (though not always). A function also uses seven bytes, of which the third and fourth point to its definition (DEF FN), the next two point to the variable it uses and the last points to an initial value of the variable (zero).

The following table summarizes the storage of simple variables:

Byte

1 and 2: name	3	4	5	6	7
REAL	exp	sign+M1	M2	M3	M4
INTEGER	sign+high	low	unused	unused	unused
STRING	length	ptr low	ptr high	unused	unused
FUNCTION	pointer to DEF FN		pointer to variable		initial value
	low	high	low	high	

If we add the following lines of code to our memory display program, all variable types are generated (including arrays):

```
1 DIM A(5),B%(5),C$(5),D$(1,5)
2 FOR I=0 TO 5:A(I)=I:B(I)=I:C$(I)=CHR$(64+I):D$(2,I)=C$(CI):
NEXT
3 M$=M$+"STRING1":Z$=Z$+"STRING2"
```

We can now dump the memory associated with simple variables. The area to be displayed can be worked out from VARTAB, ARYTAB and FRETOP. To do this the program is RUN twice, the first time from 2641($0A61) to 2732($0AAC) to display the variables and the second time from 40900($9FC4) to 40960($A000 – the start of BASIC ROM) to display the strings *in situ*:

Simple variables and string pointers

```
0A51 49 00 8C 25 10 00 00 4D  I..%...M
0A59 80 07 ED 9F 00 00 5A 80  ......Z.
0A61 08 E5 9F 00 00 46 00 8C  .....F..
0A69 25 10 00 00 54 00 8C 2A  %...T..*
0A71 C0 00 00 58 00 00 00 00  ...X....
0A79 00 00 4D 53 84 20 00 00  ..MS. ..
0A81 00 4C 53 88 01 00 00 00  .LS.....
0A89 58 31 82 00 00 00 00 58  X1. ...X
0A91 32 82 00 00 00 00 58 80  2.....X.
0A99 02 19 9D 00 00 4A 00 8C  .....J..
0AA1 2A 20 00 00 41 80 08 D8  * ..A...
0AA9 9C 00 00 41 00 25 00 01  ...A.%..
```

Strings in situ

```
9FC4 30 30 30 30 30 31 30 30  00000100
9FCC 31 32 35 35 32 38 43 43  125528CC
9FD4 38 30 30 30 30 34 39 39  80000499
9FDC 34 35 31 31 35 30 41 41  451150AA
9FE4 30 53 54 52 49 4E 47 32  0STRING2
9FEC 29 53 54 52 49 4E 47 31  )STRING1
9FF4 45 45 44 44 43 43 42 42  EEDDCCBB
9FFC 41 41 40 40 94 E3 7B E3  AA....
```

Real variables

Looking through the display above, it is quite easy to spot the real variables as their names are stored in unmodified ASCII. At $0A51 we see 'I', the first non-subscripted variable to be used, with a zero second byte in its name. We can also spot F, T, X and all the others (noting that MSB and LSB are stored as MS and LS).

Real numbers are stored in binary floating point format, always to an accuracy of 31 bits. Due to the way in which they are stored in single precision form, rounding errors are introduced though these are usually not significant enough to affect the final results. Examples of this type of error are encountered all the time as in the 'X.000001'-type value. We can convert 'I' back to a decimal number quite easily.

The exponent is stored in byte 3 and is the power of two. A unit change in this doubles or halves the resulting value. Positive exponents are expressed as 129+EXP and negative, as 129−EXP. Therefore, the full range is from 2^(−129) to 2^(127) or in decimal, from about 10^(−38) to 10^37. The high bit of byte 4 indicates the sign and is set for negative numbers. To calculate the decimal value, we have to successively divide the mantissa starting at the right by 256, add the result to the next on the right and so on until we reach M1, when we only divide by 128 and finally add 1. The resulting number will lie between 1 and 1.999999. This must finally be adjusted for its exponent and sign. The values for 'I' used below are in decimal.

```
M4  0/256=0
M3        (0+0)/256=0
M2               (0+16)/256=.0625
M1                      (.0625+37)/128=0.28955
                                      +1.00000=1.28955
```

If this is then multiplied by the exponent of 2^(140−129)=2048, the value is 2048*1.28955=2640.999 (almost 2641). This is the upper limit for the first memory display. A general formula may be written to convert any real variable from its floating point to decimal form:

(−)^(M1 AND 128)*2^(EXP−129)*(1+((M1 AND 127)+ (M2+ (M3+ (M4/256)/256))/256). . ./128)

Integer variables
These are stored in a signed high/low byte format and can range from −32768 to 32767. The high bit of byte 3 is again used to indicate the sign. The value is easily determined from the following:

(BYTE3 AND 127)*256+BYTE4+(BYTE3>127)*32768

String variables
These are the easiest of all to pick out. At $0A59 in the display above is the variable M$(its second byte is not used so is set to $80). Byte 3 tells us it is seven characters long, and bytes 4 and 5 that it is located at $9FED. The seven bytes from $9FED are "STRING1" as would be expected. Strings therefore use seven plus the number of characters bytes of RAM. There is one important point to make before leaving strings. If line 3 had simply been M$="STRING1", its pointer would have pointed to the byte at which it occurred within the program itself, that is, the byte immediately following the quote. Only computed strings are stored at the top of memory which is why the line was written M$=M$+"STRING1". This economizes on memory usage by only storing the string once. It does have the drawback that if another program is loaded in program mode all non-computed strings are lost.

Subscripted variables
Arrays may be of any type, but unlike their 'simple' counterparts, only the required number of bytes are used to store the associated values. Real are stored in five bytes, integer in two and strings in three plus their length. In addition to the savings in storing the values, the array name is only stored once. Arrays are also created in the order in which they are encountered.

The area of memory used for arrays immediately follows that for simple variables. As for the latter, it, too, is recorded at two zero page locations. The start, ARYTAB, has already been mentioned when dealing with simple variables. The end, STREND, is held in $31–$32/49–50. For each new simple variable this whole block must be moved up seven bytes in memory. There will, of course, come a time when array storage builds up to meet that of the descending strings with the resulting 'OUT OF MEMORY' error.

Each array is preceded by a detailed header of the form shown below:

Byte							
1 and 2	3	4	5	6	7	N-1	N
NAME	OFFSET TO		NO.	LAST....FIRST			
	1ST VALUE		DIMS	DIM+1.....DIM+1			
Adj. form	low	high	<256	low	high..low		high

Bytes 1 and 2 hold the name in its adjusted form. Bytes 3 and 4 record the overall memory requirement for the array (this does not include string data at the top of memory) and is the offset from its start to the next array. Byte 5 records the level of dimensioning and may not exceed 255 (a little difficult to visualize at anything more than two or three). If an undimensioned array is used, this value will default to the number of subscripts at the first occurrence. Successive pairs of bytes then hold the number of elements in each dimension (plus one for the zero subscript) in the reverse order of dimensioning. If no dimensioning has been used, these each default to 11 (10+1). The following bytes are then used to store the data.

If the program is again run and memory between ARYTAB and STREND displayed, the following results:

```
0AAC 41 00 25 00 01 00 06 00  A.%.....
0AB4 00 00 00 00 81 00 00 00  ........
0ABC 00 82 00 00 00 00 82 40  .......
0AC4 00 00 00 83 00 00 00 00  ........
0ACC 83 20 00 00 00 C2 80 13  . ......
0AD4 00 01 00 06 00 00 00 01  ........
0ADC 00 02 00 03 00 04 00 05  ........
0AE4 43 80 19 00 01 00 06 01  C.......
0AEC FF 9F 01 FD 9F 01 FB 9F  ........
0AF4 01 F9 9F 01 F7 9F 01 F5  ........
0AFC 9F 44 80 3F 00 02 00 06  .D.?....
0B04 00 03 00 00 00 01 FE 9F  ........
0B0C 00 00 00 00 00 00 01 FC  ........
0B14 9F 00 00 00 00 00 00 01  ........
0B1C FA 9F 00 00 00 00 00 00  ........
0B24 01 F8 9F 00 00 00 00 00  ........
0B2C 00 01 F6 9F 00 00 00 00  ........
0B34 00 00 01 F4 9F 00 00 00  ........
0B3C 00 01 FF FF 00 01 FF FF  ........
```

The first array is 'A(' at $0AAC. It occupies 37 bytes ($0025), and has one dimension ($01) of five elements ($06–1). The six values are then held in 5 byte real format. The next array starts at $0AAC+$25=$0AD1. This is 'B%(' which occupies only 19 ($13) for its six values. The values are easily read out as 0, 1, 2, 3, 4 and 5. The next is 'C$(', and looking at the previous display we can read out its values as @, A, B, C, D and E. A little care has to be exercised here as in the loop which generated them 'D$(' was also defined each time. This last array is the most complex of all. In its dimension statement it was defined as D$(2,5). However, in its header these are reversed (last....first). The values set were assigned to D$(1,I) and from the display we can see these occur at the second and subsequently at every third byte. This shows us that multi-dimensional arrays are stored in the form X(0,0) X(1,0) ... X(N,0) X(0,1) X(1,1) ..... X(N,N).

This just about concludes our section on variables, except to say that the default values are zero for numeric and null for strings.

Link addresses and line numbers

General

Knowing where and in what form these are stored, there is no reason why we cannot modify them from BASIC itself. This we can do using simple POKES to produce some interesting results.

Links

If we modify a link address, the program will continue to run. It will, however, list in an unusual fashion and be difficult to edit. We can use this fact to make our programs difficult to read and modify. This we can do by hiding lines (the whole program if we wish). Hiding line 110 of the display program, as its listing was originally given, can be done by

```
POKE 2049,45
```

This simply skips the link at $0817. We could very easily write a short routine to eliminate whole blocks of line numbers. This we leave up to you.

Line numbers

We can change line numbers as we did link addresses. We could change the line number of 100 to any value we choose.

```
POKE 2051,110
```

will, on listing the program, give two line 110s. A little care should be taken here, because if the line number changed is the destination for a GOTO or a GOSUB, some confusion may result.

Saving modified code

BASIC's SAVE command transfers to tape or disk a copy of the RAM between TXTTAB and VARTAB. This means any modifications are also saved. The modified code returns on loading.

Modifying BASIC

Changing the load address

We can change the point at which BASIC programs load simply by setting the value in TXTTAB. Wherever the new start is to be, a zero must be set in the byte immediately before it. Once the new start has been set, a NEW will tidy up all other pointers.

Changing the start of BASIC is useful if using sprites or programmable characters within bank 0 (the default). The *Programmer's Reference Guide* recommends lowering the top of memory to make room for the necessary data. Instead, why not move up the start of BASIC and leave yourself with far more memory to use?

Chaining programs
Chaining in this context refers to loading one program from within another.

The question arises as to whether there is a bug in the chaining process. The answer is a qualified 'no' as there can be problems. The effect of a program LOAD is roughly equivalent to executing a GOTO the first statement of the chained program. Thus, the new program can use only real, integer and computed string variables from the first. The problem occurs when the incoming program is larger than the original. If this is the case, it will overwrite the start of the variables, causing utter confusion. Once this has happened you really need to issue a CLR at the start of the new program to tidy things up. You have apparently lost all the variables anyway so there is nothing to lose.

On some micros a CHAIN command exists in addition to the normal LOAD. The action of this command is to move all or only the specified variables out of the way during the loading process and then move them back and update the necessary pointers. On the 64 no such command is available. There are two solutions. The first is to ensure a larger program is never loaded from a smaller one. The second is to make the first program the largest. To do this we do not need to generate a 'large program'. All that is needed is a simple POKE and CLR sequence at the start of the first program to reserve the necessary memory. BASIC can be fooled by:

```
POKE 46, (SIZE OF BIGGEST PROG)/256+8+1:CLR
```

In this example we have not bothered to be exact and have simply reserved to the nearest page.

Speeding up program execution
There are many ways in which to increase the speed of a program. The speed of peripheral devices plays a major part when inputting or outputting, but the topics covered here are mainly concerned with the 64 itself.

There are commercial 'CRUNCH' or 'compactor' programs available. These traditionally remove all unnecessary spaces, REMs, combine lines not the destination of a GOTO or GOSUB and that is about all. Even these few changes can produce significant increases in execution speed. Some of this we can do from BASIC itself. A short routine at the start of a program can combine lines by eliminating link addresses and line numbers, and remove all REMs and all spaces not inside quotations. After each deletion, the remaining code is moved down in memory and VARTAB is updated. The end of line marker must be replaced by a ':' to separate the last statement from the leading statement on the crunched line. There are a number of problems here. If a line number is eliminated which is referenced by a THEN, GOTO or GOSUB, the run will fail. The second problem stems from all statements following an IF being ignored if the condition is false. The resulting compacted lines are so long that they cannot be edited. As such, a universal compactor program in BASIC is fraught with danger.

It is far more sensible to consider these points at the time of writing the program. A well-known technique is to ensure that all GOTOs have destinations as near the start of the program as possible as the interpreter starts its search there. If this cannot be done, then the destination should have its high byte greater than that of the GOTO line due to the search technique used which compares this byte first.

There are three other common methods used to optimize the code:

(i) the use of variables rather than constants;
(ii) the setting up of variables in the order of frequency of use;
(iii) and, specific to the 64, turning off the video display when not in use (see *Programmer's Reference Guide*, Appendix N, 'Screen Blanking').

Using these techniques, the second program below runs almost 25% faster than the first:

```
10 PRINTTI
20 POKE 49152,0
30 FOR I=0 TO 5000
40 J=J+1
50 NEXT I
60 POKE 49152,16
70 PRINTTI
```

```
10 PRINTTI:POKE53265,0:P=1:FORI=0TO5000:
J=J+P:NEXT:POKE53265,16:PRINTTI
```

The second program does leave you in x and y scroll mode if you are wondering just what has happened.

Conclusion

This chapter should have given you one or two ideas to play around with. Before reading Chapter 5, you might like to think about how to write simple renumber, delete, dump, and recover NEWed program routines. You might also like to think about how to overcome the chaining problem by, as the last action on leaving a program, moving all variables as high as they can go in memory. The first action of the chained program should be to move them back down to the end of this program and reset the necessary zero page pointers.

2 Peripherals

Introduction

This chapter deals with some of the more common peripherals for the 64. Also included here are the keyboard and screen even though they are not quite peripherals in the same sense as a disk drive, cassette or printer.

It is not our intention to go into any of these in great detail as the subject could fill a book of its own. We have tried to look at features of more immediate use.

Keyboard

Use of the keyboard, its ROM drive routines and RAM vectors is covered in Chapter 4. Programming of the keyboard is used extensively in Chapter 5. The following are a few useful points not directly covered elsewhere.

Keyboard as a device
The keyboard is viewed as device 0 by the 64's operating system and is the default for input. As such it may be used like any other device and a file OPENed to it. This file may only be for input and any attempt to output to it will result in an error. Once opened for input the 'annoying' question mark prompt is removed from the INPUT command display. When information is being obtained from it using INPUT# all warning messages, such as the double question mark for insufficient data, also disappear. The open format is the same as for any other device:

```
OPEN 1,0 or OPEN 1,0,129,"QWERTY"
```

In the second example everything following the 0 will be ignored. Use of the keyboard in this way is highlighted in the DISK utility facility in this chapter (see below).

Auto-repeat
See Chapter 10.

Key detection
See Chapter 4 and Appendix I.

Keyboard Buffer – KEYD ($0277–$0280/631–640)
The 64 provides type-ahead of up to ten characters. The buffer operates on the principle of first in/first out. However, once full no new characters will be accepted until it has been partially emptied. Characters are taken singly by a GET, up to the first RETURN on an INPUT and the buffer is emptied on an END.

The length of the buffer is determined by XMAX ($0289/649) and as this is in RAM it may be changed. Theoretically, the buffer could be lengthened, but in practice this cannot be done as the RAM immediately following it is used for other purposes. The size, however, may be decreased and is perhaps most useful when the length is set to 1 where a program requires careful, restricted input or type-ahead is to be discouraged. Setting XMAX to 0 is quite a good way to prevent unwanted user input (the STOP key is still active as it is scanned by a different routine – see Chapter 10 to disable).

As the buffer is in RAM we can put data directly into it by simply POKEing the ASCII codes of the characters required. To complete the process NDX ($C6/198) must be set to tell the system how many characters are in the buffer. This type of approach is used extensively in the BASIC utilities in Chapter 5 and we refer you there for examples of using the keyboard in this way.

One final point before leaving the keyboard which many of you may have already discovered. Pressing the 'Control' key with any other simply sets bit 6 of the ASCII code of the character low. For example, CNRL/T is the same as the DEL key ($14/20).

Cassette

The 64 does not have to use a CBM Datassette. There are, to our knowledge, two manufacturers of interfaces which allow standard cassette recorders to be used. These interfaces duplicate the part of the interface normally resident within the Datassette. It is even possible to use a standard cassette through a suitable edge connector, but do not expect a high success rate in loading back saved programs or to get anything from recordings produced on a CBM recorder (see *Programmer's Reference Guide*, Appendix I for connection details).

Many consider the cost of the dedicated cassette high. However, it avoids the need to adjust the volume and tone controls to ensure an accurate save (a problem on many other micros where even saving twice is not guaranteed to work). It also seems slow, but perhaps is not as slow as it at first appears. Data is transferred between the 64 and the cassette at about 300 baud (some micros offer an optional fast 1200 baud rate). When a program is saved two copies are made. On reloading the first copy is put into memory and this is then compared with the second to check for and possibly recover load errors. In our experience it has proved worth the additional expense to buy the

Datassette for peace of mind and to avoid the loss of many hours of hard work.

The speed of operation of the cassette has been chosen for reliability, but like most things on the 64 we can even change that. For many years superfast, jet, turbo, fast or whatever you care to call them, operating systems have been available for the PET, more recently for the VIC20 (ARROW) and now for the 64. The machine code listing of the original PET version has even been published. Many games now come with a high-speed load (some without the option for a normal load which has proved annoying when your cassette cannot cope with fast loads). These fast operating systems can be made to run the cassette at a higher speed than the standard operating speed of the disk drive (even this can be increased). There is no secret as to how it is done, but as many software houses pay a royalty for its use, or even sell their own versions of a high-speed loader, we have decided not to include a version of our own.

LOAD and SAVE with cassette (see Programmer's Reference Guide, *Chapter 2)*

These are dealt with in detail in the *Programmer's Reference Guide*, so we will deal with them briefly here. The general syntax for SAVE is (where square brackets denote optional parameters):

```
SAVE["program" or string variable] [,device]
      [, secondary address]
```

If no parameters are specified, the BASIC program currently in memory will be copied to the default of cassette without a name.

The secondary address is the more interesting. A secondary address of 2 will write an end of tape marker and one of 3 appears at first sight to do exactly the same. Using either of these will prevent the tape being read beyond this point without being physically wound on. There is, however, a world of difference on loading (see below). With an address of 3 not only is the end of tape set, but an end of tape header is written which is a duplicate of the program header with a type of 5.

The area of RAM saved is that between the values held in TXTTAB and VARTAB. These pointers are automatically kept up to date by the operating system whilst a program is being edited. Should we wish to save an area of memory other than the BASIC program, we can set these up by POKEing in the appropriate values (remember low/high format). This allows us to save machine code from BASIC or even the screen itself. Data stored in memory is more economically saved this way as only single bytes are saved and not the ASCII characters which make up each number (saves at least two bytes per number between 0 and 255). The problem is that on returning from the save, the current BASIC program and variables are lost until these pointers are restored. If you are going to play with TXTTAB and VARTAB from BASIC, put the original values out of harm's way, say below $0800 or above $C000, to allow them to be recovered.

The syntax for LOAD is identical to that for SAVE:

LOAD ["program" or string variable] [,device]
        [,secondary address]

LOAD reads the next program from tape. If a program name is specified, then the named program will be searched for and if found loaded or if an end of tape marker is found first the cassette will stop. Again it is the secondary address which is of major importance. A 1 requests the operating system to put the program at the same location from which it was saved. If no secondary address is specified, then providing the program came from an address above the current start of BASIC it will return to its original location, but after the load TXTTAB will still hold the start of BASIC whereas VARTAB will hold the end address. The same is true when 1 is used, but in this case a load may be carried out below the start of BASIC. Typically, when loading machine code from BASIC an 'OUT OF MEMORY' error results if the code locates above $9FFF due to the setting of VARTAB. A save with a secondary address of 3 ensures the code is reloaded to its original address, regardless of the syntax of the LOAD command (extended monitors use 3).

Tape Buffer
The tape uses a 192 byte I/O buffer, TBUFFR, which in its default setting extends from $033C–03FB/828–1019. TBUFFR need not reside here and may be relocated, as a pointer to its start is held in RAM at SAL ($AC–AD/172–173). To move it, simply POKE in the new location in the usual low/high byte format (STOP/RESTORE will reset it). We have found this of use when storing sprite data blocks in bank 0 when memory is tight ($C000 is yet again a good place to put it). Usage of the buffer is very different between program and data files. Programs only use the buffer to store their header information (see below) and the transfer of memory is direct from the I/O port without passing through the buffer. Data files, on the other hand, use the buffer initially for the header then subsequently to hold 191 byte blocks (the first byte is used as a marker). This avoids continual starting and stopping of the tape motor and by using this block system the tape is more reliable as it is allowed to pick up speed between each read/write operation. Another zero page location, BUFPNT ($A6/166), holds the current position within the buffer.

Tape Headers
All files are stored on tape with an initial header which is the length of the buffer. The exact format depends on the syntax of the SAVE or OPEN command (secondary address of 2 on an OPEN also writes an end of tape

marker). Each is made up of an identifier, two addresses and a file name, the format of which is given below:

Program headers

ID	START	END	FILE NAME (spaces to pad)
1	1 8	25 16	65 66 67 32 32 32

Data headers

ID	START	END	FILE NAME
4	60 3	252 3	68 65 66 32 32

The ID identifies the file type and for a program may also take a value of 3. The two bytes immediately following it are the start load address in low/high format and the next two the end address. The file name is not limited to 16 characters and in fact can be up to 187 characters. This allows machine code to be embedded in a header to add additional security to a program. When the name is printed out by LOADING ...... only the first 16 characters are displayed. The header to a data file also contains the start/end bytes but these hold the start and end of TBUFFR itself.

The last operation on completion of a save or write is to store a duplicate header. If the command had a secondary address indicating an end of tape marker, then the ID would be changed to a 5 before writing. On loading or reading to the end of a file the last operation is to get back this trailing header (which remains until the next tape operation).

Tape directories

Tape directories as such do not exist unless you are using an improved cassette operating system such as ACOS+. There are times when it is necessary to catalogue a tape. The process is time-consuming as it is, not surprisingly, directly proportional to the length of the tape. The following program may be used to do the job. It is best left running whilst you go away to do something else.

Any header will be read with an OPEN statement. CLOSEing it immediately ceases tape operation and program execution continues. The parameters are then pulled from the buffer and stored for later use. The process is repeated for the next header. When the end of tape is reached or you stop the program, a simple GOTO 260 will display the file information. This is the file type, up to 16 characters of its name with non-alphanumeric characters replaced by a ".", and if a program its start and end addresses (in hex).

```
100 DIM F$(50),FT$(50),SA$(50),EA$(50):C
B=828
110 PRINT"[CLS]PRESS PLAY ON TAPE"
120 IF PEEK(1)<>7 GOTO 120
130 I=I+1:OPEN 1:CLOSE 1:PRINTF$(I-1)
140 FT$(I)=RIGHT$("[5SPC]"+STR$(PEEK(CB)
),4)
150 IF PEEK(CB)=4 THEN SA$(I)="[2SPC]***
*":EA$(I)="[2SPC]****":GOTO 200
160 X=PEEK(830):GOSUB 360:SA$(I)=X$
170 X=PEEK(829):GOSUB 360:SA$(I)=" $"+SA
$(I)+X$
180 X=PEEK(832):GOSUB 360:EA$(I)=X$
190 X=PEEK(831):GOSUB 360:EA$(I)=" $"+EA
$(I)+X$
200 A$="":FOR J=833 TO 848
210 X=PEEK(J):IF X<32 OR X>95 THEN A$=A$
+".":GOTO 230
220 A$=A$+CHR$(X)
230 NEXT J
240 F$(I)=LEFT$(" "+A$+"[18SPC]",17)
250 GOTO 130
260 H$="[CLS]TYPE FILENAME[9SPC]START[3S
PC]END":PRINTH$
270 FOR J=1 TO I:PRINTFT$(J);F$(J);SA$(J
);EA$(J)
280 IF INT(J/20)<>J/20 GOTO 320
290 PRINT"PRESS RETURN FOR NEXT PAGE"
300 GET A$:IF A$<>CHR$(13) GOTO 300
310 PRINTH$
320 NEXT J
330 INPUT "REVIEW AGAIN";Y$:IF Y$="Y" GO
TO 260
340 IF Y$<>"N" GOTO 330
350 CLOSE 1:END
360 X1=INT(X/16):X2=X-X1*16
370 X$=CHR$(X1+48-7*(X1>9))+CHR$(X2+48-7
*(X2>9))
380 RETURN
```

Unfortunately, during tape I/O the internal clock variable (TI$) is not updated as the interrupt is used exclusively for tape timing. Had this not been the case, a read of this variable could have been used to calculate the value of the tape counter. The best suggestion we can come up with is if the file is a program then the difference in its start and end addresses could be used to determine the loading time. For a data file bytes could be taken until the status is set to the end of file, the number of bytes read being an indication of the time. We might as well do this as the tape is running anyway. The time taken may be used to work out an approximate counter reading.

Auto-running
Generating programs which auto-run is also discussed in Chapter 10. There are many ways to accomplish this, most of which involve fairly detailed knowledge of the operating system. The following are suggestions only for you to pursue. All but one are suitable for disk or tape.

The stack
During LOAD the return address is placed on the stack. As this is an area of RAM, there is no reason why we cannot load through this area and put our own address on instead. This could then go to our own machine code routine. The file type should be 3 to ensure a load to its original position. The same would apply to disk or tape if loaded with a secondary address of 1.

BASIC warm start – $Ø3Ø2
After a load in direct mode BASIC is warm-started. Again as this vector is held in RAM we can load through it. The new value it then contained could jump to our machine code or straight to RUN (for BASIC programs).

IRQTMP – $Ø29F
This stores the current IRQ vector during tape I/O which is restored after the tape operation. Again we can do the same to this as in the above. On the first normal interrupt the action will be taken. This, of course, can only be used with tape.

CHAIN command of the UTILITY
See Chapter 8.

Screen

The utilities in this section are confined to the text screen.

The screen on the 64 is a 4Ø column by 25 line memory-mapped display. Chapter 3 and Appendices B to D of the *Programmer's Reference Guide* cover in great detail all aspects of the screen and it is to there that we refer you. All the following utilities assume that you are familiar with or know the following.

i) The screen may be moved from its default position.
ii) There are two character sets.
iii) The screen has an associated colour map at $D8ØØ on.
iv) The display codes differ from the ASCII codes.
v) Commodore 'ASCII' is not true ASCII which only ranges from Ø to 127. (Consult your printer manual.)

Printer dump

There are two routines, both of which output the current display in standard ASCII to a printer. One is a BASIC subroutine and the other is machine code. The second is noticeably faster than the first, as would be expected.

Both routines take account of whether the 64 is in upper or lower case mode as well as checking for the location of the screen.

64 owners with Commodore printers need not concern themselves with the conversions to standard ASCII.

BASIC printer dump subroutine

The version given here is for an RS232 printer running at 300 baud without auto-line feed. For this reason the output logical file is assigned at the start of the program. The out put file is designated 'P' to avoid specific reference to allow for easier change to other printers. The display is centred on an 80 column display by printing 20 spaces at the start of each line.

The program first examines the lower/upper case register at 53272 by calling the subroutine at 60090. If in lower case, LC is assigned a value of 32 (note lower case 'a' in character set 2 has a PEEK value of 1 which is standard ASCII is 97 – that is, bit 5 set). This adjustment will be applied to all letters between 'a' and 'z'. The whole dump is enclosed within two loops: I for the rows and J for the columns. All screen codes are ANDed with 127 to reduce them to values in the range 0 to 127 to eliminate reversed characters. If the screen code is <32, we have to add 64 and the LC adjustment. If it lies between 32 and 65, we can print it unchanged. Only if in lower case mode do we need to check for upper case letters. If we were in upper case, these would be non-printed graphic characters. If in LC then the ANDed code is already in standard ASCII. If all the tests have failed, we have a graphic character so we replace it by a space to maintain the layout of printable text. Once a screen line has been processed we print it preceded by 20 spaces and recycle for all remaining 24 lines.

```
10 OPEN 129,2,0,CHR$(6)
60000 GOSUB 60090
60010 FOR I=0 TO 24:A$="":FOR J=0 TO 39:
CH=PEEK(S+I*40+J)
60020 CH=CH AND 127
60030 IF CH<32 THEN CH=CH+64+LC:GOTO 600
70
60040 IF CH<65 GOTO 60070
60050 IF CH<91 AND LC GOTO 60070
60060 CH=G
60070 A$=A$+CHR$(CH):NEXT J
60080 PRINT#P,SPC(SP);A$:NEXT I:CLOSE P:
RETURN
```

```
60090 P=129:SP=20:G=32
60100 LC=0:IF PEEK(53272)=23 THEN LC=32
60110 S=PEEK(648)*256:RETURN
```

Whenever a dump is required, simply GOSUB 60000. This could be actioned by, say, a GET statement, but should not add to the display, or if it does then only 24 lines should be printed. To improve the presentation, blank lines or a form feed should be issued at the end of the dump.

Machine code printer dump

The logic of this routine is identical to that above and is therefore not described in detail. The differences are that it is much faster and it does not pad a line with 20 leading spaces.

The routine as written assumes logical file 2 is open to the printer at the time of calling. To change this, simply alter the byte at $C001 with a POKE. It works by changing the output device through the CHKOUT KERNAL call to that associated with file #2 (the equivalent of a CMD from BASIC). This then allows us to use the KERNAL routine CHROUT to output the data. There is a routine in ROM which could be used to do most of the conversion, but for this exercise the technique used here is adequate and easier to follow. The device need not be the printer and could be the disk or tape depending on the OPEN statement. We do not recommend you use this routine with anything other than a printer as far better screen saves follow. Once the dump is complete, the default device for output is restored to the screen before returning to BASIC.

The routine is used by at some point including an OPEN 2,4 or OPEN 2,2,CHR$() if using RS232. A simple SYS 49152 will perform the dump. If your printer requires a forced line feed, make the necessary adjustment to $C001 for a value greater than 127.

BASIC loader for the machine code

The following must be loaded and run. Once this has been done the code remains present until overwritten by something else. Once run the machine code may be saved using an extended monitor for ease of loading later.

```
1 DATA 162, 2, 32, 201, 255, 173, 136, 2
, 133, 88, 169
2 DATA 0, 133, 87, 173, 24, 208, 201, 21
, 208, 6
3 DATA 169, 0, 133, 89, 240, 4, 169, 32,
 133, 89
4 DATA 169, 32, 133, 90, 24, 165, 88, 10
5, 3, 133
5 DATA 91, 162, 4, 160, 0, 177, 87, 41,
127, 24
```

```
6 DATA 201, 31, 176, 7, 24, 105, 64, 101
, 89, 144
7 DATA 24, 24, 201, 64, 176, 2, 144, 17,
 24, 165
8 DATA 89, 240, 10, 177, 87, 24, 201, 91
, 176, 3
9 DATA 24, 144, 2, 169, 32, 32, 210, 255
, 224, 1
10 DATA 208, 4, 192, 232, 240, 23, 230,
96, 201, 40
11 DATA 208, 9, 169, 13, 32, 210, 255, 1
69, 0, 133
12 DATA 96, 200, 208, 187, 230, 88, 202,
 208, 182, 169
13 DATA 13, 32, 210, 255, 162, 0, 32, 20
1, 255, 96
14 DATA 0, 255, 255, 0, 0, 255, 255, 0,
0, 255
15 FOR I=49152 TO 49292:READ A:POKE I,A:
NEXT I
```

Here is the assembly listing which is fully annotated to allow you to follow it:

```
C000  A202     LDX  #$02    log file to printer
C000  20C9FF   JSR  $FFC9   perform CMD2 via CHKOUT
C005  AD8802   LDA  $0288   screen start from HIBASE
C008  8558     STA  $58     set start registers
C00A  A900     LDA  #$00
C00C  8557     STA  $57
C00E  AD18D0   LDA  $D018   check upper/lower case
C011  C915     CMP  #$15    is it upper
C013  D006     BNE  $C01B   no
C015  A900     LDA  #$00    set adjustment value
C017  8559     STA  $59     for ASCII
C019  F004     BEQ  $C01F   skip lower case
C01B  A920     LDA  #$20    lower case set adj flag
C01D  8559     STA  $59     as ASCII a=97 etc.
C01F  A920     LDA  #$20    set non-printable flag
C021  855A     STA  $5A     to a space
C023  18       CLC
C024  A558     LDA  $58     set MSB end of screen
C026  6903     ADC  #$03
C028  855B     STA  $5B
C02A  A204     LDX  #$04    almost 4 pages/screen
C02C  A000     LDY  #$00    counter within page
```

```
C02E  B157    LDA  ($57),Y  get byte
C030  297F    AND  #$7F     eliminate high bit 7
Remember difference between screen and ASCII codes
C032  18      CLC           start checks
C033  C91F    CMP  #$1F     less than a space
C035  B007    BCS  $C03E    no go to next check
C037  18      CLC
C038  6940    ADC  #$40     make ASCII by adding 64
C03A  6559    ADC  $59      add lower case adj.
C03C  9018    BCC  $C056    always taken
C03E  18      CLC
C03F  C940    CMP  #$40     check for upper case in
C041  B002    BCS  $C045    l/c mode & branch > 65
C043  9011    BCC  $C056    !-? same in both sets
C045  18      CLC           check upper case
C046  A559    LDA  $59      if zero
C048  F00A    BEQ  $C054    branch to avoid graphic
C04A  B157    LDA  ($57),Y  get l/c byte again
C04C  18      CLC           check not gt Z
C04D  C95B    CMP  #$5B
C04F  B003    BCS  $C054    if so avoid graphic
C051  18      CLC
C052  9002    BCC  $C056    valid A-Z so skip space
C054  A920    LDA  #$20
C056  20D2FF  JSR  $FFD2    print char
C059  E001    CPX  #$01     on last page
C05B  D004    BNE  $C061    no - so branch
C05D  C0E8    CPY  #$E8     yes so check end $**E8
C05F  F017    BEQ  $C078    branch all done
C061  E660    INC  $60      end of screen line reg
C063  C928    CMP  #$28     is it 40 dec
C065  D009    BNE  $C070    no so skip next bit
C067  A90D    LDA  #$0D     output next bit
C069  20D2FF  JSR  $FFD2    print it
C06C  A900    LDA  #$00     rezero end of line reg
C06E  8560    STA  $60
C070  C8      INY           continue current page
C071  D0BB    BNE  $C02E    branch if not finished
C073  E658    INC  $58      inc next page register
C075  CA      DEX
C076  D0B6    BNE  $C02E    always taken
C078  A90D    LDA  #$0D     RETURN for last line
C07A  20D2FF  JSR  $FFD2
C07D  A200    LDX  #$00     restore screen output
C07F  20C9FF  JSR  $FFC9
C082  60      RTS
```

To improve this, why not patch into the interrupt routine to, for example, dump the screen whenever a designated key is pressed rather than using the SYS command? Chapter 4 explains the interrupt in detail and Chapter 10 gives an example of its use. If you decide to do this, remember to include a routine to disable the patch. The necessary enable and disable routines can be added at the end of the code as given. The logical file will still have to be OPENED unless the appropriate KERNAL routines are called.

Screen dumps

Three ways are given to save the screen and its associated colour map in this section. Two are in BASIC and the third is in machine code. Both BASIC programs use a sequential file to store the data, but differ in the length of file produced. The machine code saves the screen as a program file and is the most economical and by far the quickest.

A few points should be made before discussing the routines in detail. Any area of memory may be saved from BASIC by setting TXTTAB and VARTAB to its start and end addresses. The problem is that once we have changed these pointers we have temporarily lost our program. Another problem is that a LOAD will cause BASIC to warm-start, which is this case will be at the newly set TXTTAB address. The screen is an area of memory and may be loaded and saved in this way. Unfortunately, its default position is below the normal start of BASIC so a 'crash' or 'hang-up' is usually the result. Try it and see. So from a practical viewpoint we must resort to other means.

All the following routines check HIBASE for the current screen location. The resulting screens will always reload to the current screen position regardless of its location at the time of saving. The reloaded screen will be identical to that saved in both characters and colours.

Screen save using numbers
This routine firstly PEEKS out the border and background colours and writes them, as numbers, to a disk file (change the OPEN command for tape). It then proceeds, writing alternate screen and colour values until finished.

To save a screen: GOSUB 60000
To load a screen: GOSUB 60050

```
60000 OPEN 2,8,2,"@0:TEST,S,W"
60010 S=PEEK(648)*256:C=55296
60020 PRINT#2,PEEK(53280);",";PEEK(53281
);CHR$(13);
60030 FOR I=S TO S+999:PRINT#2,PEEK(I);"
,";PEEK(C+I-S);CHR$(13);:NEXT I:CLOSE 2
```

```
60040 RETURN
60050 OPEN 3,8,3,"TEST,S,R"
60060 INPUT#3,A,B:POKE 53280,A:POKE 5328
1,B
60070 S=PEEK(648)*256:C=55296
60080 FOR I=S TO S+999:INPUT#3,A,B:POKE
I,A:POKE C+I-S,B:NEXT I:CLOSE 3:RETURN
```

Because numbers are written as their ASCII codes three to five bytes are used for each value (spaceXXXreturn). Therefore, using this method we will generate a sequential file of between 6 and 10K, which seems rather excessive. The second method reduces the size of this file.

Screen save using characters

This time a single byte is used to store each value in the screen and colour maps. This is done by simply PEEKing the value and generating the corresponding CHR$() character with the ASC() function. Zero values must be trapped as ASC(0) will give a syntax error. The resulting file uses only one byte for most values and the file size is therefore about 2K. This is obviously far faster to generate and restore.

To save a screen: GOSUB 60000
To load a screen: GOSUB 60050

```
60000 OPEN 2,8,2,"@0:TEST,S,W"
60010 S=PEEK(648)*256:C=55296
60020 PRINT#2,CHR$(PEEK(53280));CHR$(PEE
K(53281));
60030 FOR I=S TO S+999:PRINT#2,CHR$(PEEK
(I));CHR$(PEEK(C+I-S));:NEXT I:CLOSE 2
60040 RETURN
60050 OPEN 3,8,3,"TEST,S,R"
60060 GET#3,A$:IF A$="" THEN A$=CHR$(0)
60070 POKE 53280,ASC(A$)
60080 GET#3,A$:IF A$="" THEN A$=CHR$(0)
60090 POKE 53281,ASC(A$)
60100 S=PEEK(648)*256:C=55296
60110 FOR I=S TO S+999:GET#3,A$:IF A$=""
 THEN A$=CHR$(0)
60120 POKE I,ASC(A$):GET#3,A$:IF A$="" T
HEN A$=CHR$(0)
60130 POKE C+I-S,ASC(A$):NEXT I:CLOSE 3:
RETURN
```

Machine code screen save
This is by far the best method. It is very simple to use the KERNAL LOAD and SAVE for both the screen and colour maps. Using these as they stand, two files would be generated – one for the colour map and one for the screen. This is no hardship, but a relocated load would be required if a screen is being restored to a different location from whence it came. This is not difficult, but perhaps is not the best way.

We have approached the problem slightly differently. Before performing the save, the screen and colour maps are combined into a 2K block at a convenient address. This has to be out of the way of BASIC to avoid corrupting program or data areas. This could be a reserved area at the end of BASIC or even under BASIC ROM if a switch like that used in the UTILITY is implemented to throw out and restore ROM. This is possible as no BASIC ROM calls are made. For this example we have chosen to move the screen from its current position to $C400 and the colour map to $C800. The routine also saves the sprite pointers and if you do not wish it to do so then you will have to modify the code to move 1000 rather than its current 1024 bytes from or to each area.

All the routine does to save is to move both screen and colour maps then use the KERNAL SAVE from $C400 to $CC00. To restore the screen it is reloaded to $C400 and moved back to the colour map and the current screen position.

BASIC loader for screen save
The following must be loaded and run before it can be called. Once run it may be saved using an extended monitor for ease of loading later.

```
1 DATA 32, 253, 174, 201, 76, 208, 6, 16
9, 0, 133, 87
2 DATA 240, 11, 201, 83, 240, 3, 32, 8,
175, 169
3 DATA 255, 133, 87, 32, 115, 0, 32, 253
, 174, 201
4 DATA 34, 240, 3, 32, 8, 175, 32, 115,
0, 165
5 DATA 122, 133, 187, 165, 123, 133, 188
, 160, 0, 177
6 DATA 122, 201, 34, 240, 8, 200, 192, 1
9, 208, 245
7 DATA 32, 8, 175, 132, 183, 152, 24, 10
1, 122, 133
8 DATA 122, 144, 2, 230, 123, 32, 115, 0
, 32, 253
9 DATA 174, 144, 3, 32, 8, 175, 56, 233,
 48, 133
```

```
10 DATA 186, 169, 1, 133, 184, 133, 185,
 169, 0, 133
11 DATA 88, 133, 90, 133, 92, 133, 94, 1
73, 136, 2
12 DATA 133, 89, 169, 196, 133, 91, 169,
 216, 133, 93
13 DATA 169, 200, 133, 95, 165, 87, 240,
 43, 160, 0
14 DATA 162, 4, 177, 88, 145, 90, 177, 9
2, 145, 94
15 DATA 200, 208, 245, 230, 89, 230, 91,
 230, 93, 230
16 DATA 95, 202, 208, 234, 166, 94, 164,
 95, 169, 196
17 DATA 133, 91, 169, 90, 32, 216, 255,
32, 115, 0
18 DATA 96, 165, 87, 32, 213, 255, 160,
0, 162, 4
19 DATA 177, 90, 145, 88, 177, 94, 145,
92, 200, 208
20 DATA 245, 230, 89, 230, 91, 230, 93,
230, 95, 202
21 DATA 208, 234, 32, 115, 0, 96, 0, 255
, 255, 0
22 FOR I=49152 TO 49362:READ A:POKE I,A:
NEXT I
nb"basic2"
```

To save a screen: SYS 49152,S, "filename",DEVICE
SYS 49152,S, "@0:filename",DEVICE to replace on disk
To load a screen: SYS 49152,L,"filename", DEVICE

All parameters are required and an illegal or missing parameter will produce a SYNTAX ERROR. A file name or a minimum of "" is required, even with cassette. Remember all bytes following a SYS command are ignored.

The following is the assembly listing, which should be self-explanatory. The first part of the routine is our own version of 'GET PARAMETERS' and is a useful technique when passing parameters to a routine enabled with a SYS. CHRGET (see Chapter 3) is used to gather the necessary bytes.

ASSEMBLY LISTING

```
Set up the parameters - common to both LOAD and SAVE
C000  20FDAE   JSR  $AEFD    check for comma
C003  C94C     CMP  #$4C     is next char L for LOAD
C005  D006     BNE  $C00D    no then test for SAVE
C007  A900     LDA  #$00     set flag at 57
C009  8557     STA  $57      to zero for later use
C00B  F00B     BEQ  $C018    always taken if LOAD
C00D  C953     CMP  #$53     is it S for SAVE ?
C00F  F003     BEQ  $C014    if so continue
C011  2008AF   JSR  $AF08    not L or S=SYNTAX ERROR
C014  A9FF     LDA  #$FF     set flag to FF for save
C016  8557     STA  $57
C018  207300   JSR  $0073    inc CHRGET (see ch.4)
C01B  20FDAE   JSR  $AEFD    next comma
C01E  C922     CMP  #$22     opening quote of name
C020  F003     BEQ  $C025    OK so continue
C022  2008AF   JSR  $AF08    no quote so SYNTAX ERROR
C025  207300   JSR  $0073    inc CHRGET to name
C028  A57A     LDA  $7A      set FNADR low byte
C02A  85BB     STA  $BB
C02C  A57B     LDA  $7B      do same for high
C02E  85BC     STA  $BC
C030  A000     LDY  #$00     find length of name
C032  B17A     LDA  ($7A),Y  by searching for closing
C034  C922     CMP  #$22     quote
C036  F008     BEQ  $C040    found it so exit
C038  C8       INY
C039  C013     CPY  #$13     limit of 16 chars +3 for
C03B  D0F5     BNE  $C032    "@0:" toreplace on disk
C03D  2008AF   JSR  $AF08    >limit so SYNTAX ERROR
C040  84B7     STY  $B7      store length in FNLEN
C042  98       TYA
C043  18       CLC
C044  657A     ADC  $7A      set CHRGET to end quote
C046  857A     STA  $7A      low byte
C048  9002     BCC  $C04C
C04A  E67B     INC  $7B      inc high if page crossed
C04C  207300   JSR  $0073    get next byte
C04F  20FDAE   JSR  $AEFD    comma ?
C052  9003     BCC  $C057    OK
C054  2008AF   JSR  $AF08    no then SYNTAX ERROR
C057  38       SEC
C058  E930     SBC  #$30     make byte a number 0-9
C05A  85BA     STA  $BA      store current device-FA
C05C  A901     LDA  #$01     set secondary add to 1
```

```
C05E  85B8     STA  $B8      and store at SA
C060  85B9     STA  $B9      same for logical file-LA
Set up pointers to be used in the move
C062  A900     LDA  #$00
C064  8558     STA  $58
C066  855A     STA  $5A
C068  855C     STA  $5C
C06A  855E     STA  $5E
C06C  AD8802   LDA  $0288    find the current screen
C06F  8559     STA  $59      start from HIBASE
C071  A9C4     LDA  #$C4
C073  855B     STA  $5B
C075  A9D8     LDA  #$D8
C077  855D     STA  $5D
C079  A9C8     LDA  #$C8
C07B  855F     STA  $5F
C07D  A557     LDA  $57      0 for LOAD
C07F  F02B     BEQ  $C0AC    do LOAD
Move screen and colour to one block and perform SAVE
C081  A000     LDY  #$00     SAVE use  Y within page
C083  A204     LDX  #$04     and X for page counter
C085  B158     LDA  ($58),Y  read byte from screen
C087  915A     STA  ($5A),Y  store at combined area
C089  B15C     LDA  ($5C),Y  get char colour
C08B  915E     STA  ($5E),Y  store in comb area+$0400
C08D  C8       INY
C08E  D0F5     BNE  $C085    cycle for one page
C090  E659     INC  $59      inc all high bytes
C092  E65B     INC  $5B
C094  E65D     INC  $5D
C096  E65F     INC  $5F
C098  CA       DEX           dec X and
C099  D0EA     BNE  $C085    repeat till 4 pages done
C09B  A65E     LDX  $5E      X holds low end of save
C09D  A45F     LDY  $5F      Y the high byte
C09F  A9C4     LDA  #$C4     use 5A/5B on zero page
C0A1  855B     STA  $5B      for start of SAVE
C0A3  A95A     LDA  #$5A     A must hold offset 5A
C0A5  20D8FF   JSR  $FFD8    do SAVE
C0A8  207300   JSR  $0073    must inc CHRGET before
C0AB  60       RTS           returning to BASIC
Perform LOAD and split block into char and colour maps
C0AC  A557     LDA  $57      read flag - A=0 for LOAD
C0AE  20D5FF   JSR  $FFD5    do LOAD
C0B1  A000     LDY  #$00
C0B3  A204     LDX  #$04
```

```
C0B5  B15A    LDA  ($5A),Y  reverse of SAVE
C0B7  9158    STA  ($58),Y  even if the screen
C0B9  B15E    LDA  ($5E),Y  was at a different
C0BB  915C    STA  ($5C),Y  location at the time of
C0BD  C8      INY           SAVE it will go to the
C0BE  D0F5    BNE  $C0B5    current position
C0C0  E659    INC  $59
C0C2  E65B    INC  $5B
C0C4  E65D    INC  $5D
C0C6  E65F    INC  $5F
C0C8  CA      DEX
C0C9  D0EA    BNE  $C0B5
C0CB  207300  JSR  $0073    must inc CHRGET before
C0CE  60      RTS           returning to BASIC
```

Disk

This section deals with the 1541 disk drive though much is directly applicable to the 3040 and 4040 units.

The manual supplied with the 1541 contains all the information that most users will require. Perhaps the most difficult to master are the direct access programming commands such as BLOCK-READ, and so on. There is only one way to become proficient in their use and that is to experiment. When experimenting we suggest you use a disk containing unwanted information as disasters can happen.

We supply only one utility in this section which we like to think of as an expandable disk utility. Once direct access programming is mastered, there are all sorts of fun things you can do. To use it to its best advantage you have to know something of how the disk operates and how information is stored. To this end we give below a very short introduction and would refer you to the 1541 manual itself.

Introduction

The 1541 is a self-contained, intelligent device. It has two processors and its disk operating system (DOS) in ROM along with an area of RAM used for input/output (buffering) operations. This differs from, say, the BBC Micro where the interface is within the micro itself and, depending on the type of interface, removes RAM from the user area. The disadvantage to this self-contained arrangement is that you cannot use non-Commodore units (there are one or two now available) as most disk manufacturers do not supply a suitable controller and DOS.

Almost the whole of the disk's capacity can be used to store data except for one or two reserved areas. The disk is divided into tracks which are further subdivided into a varying number of sectors. Tracks are numbered from 1 through to 35 whereas sectors start at zero. The following is the arrangement for both the 1541 and 4040 units (see Table 6.1, 1541 manual):

TRACK	SECTOR	TRACK	SECTOR
1–17	0–20	25–30	0–17
18–24	0–18	31–35	0–16

In order to know where to find information, the disk uses an index or directory track. This is track 18 and it has two special areas. The first is track 18 sector 0 and is the Block Availability Map (BAM). This keeps a record of all sectors in use unless direct access programming operations have been used without a BLOCK-ALLOCATE command. If this is the case then the information is there, but can be overwritten as it is empty as far as DOS is concerned. The entries in BAM are made up as follows (see Table 5.1 of manual):

BYTES	CONTENTS
000–001	Pointer to start of directory 18/01
002	Holds an 'A' for 4040 format
004–143	Four bytes for each of the 35 tracks indicating whether in use 1=free 0=allocated
144–161	Disk name plus shifted spaces to make 16 in total
162–163	Disk ID
165–166	Disk version of 2A

Each track uses four bytes in BAM. The first stores the number of free sectors on a track and is used in computing BLOCKS FREE. The remaining three are used to indicate whether a particular sector is allocated (bit set low and one bit per sector). As the maximum number of sectors is 21, not all bits are used. The following is a dump of BAM from which you can pick out the information given above. All values are given in hex with the byte position within the sector given first followed by this byte's value and the next seven and at the end of the line the equivalent ASCII characters (if printable):

```
START  BYTES                             ASCII

00     12 01 41 00 15 ff ff 1f  ..a.....
08     15 ff ff 1f 15 ff ff 1f  ........
10     15 ff ff 1f 15 ff ff 1f  ........
18     15 ff ff 1f 15 ff ff 1f  ........
20     15 ff ff 1f 15 ff ff 1f
28     15 ff ff 1f 00 00 00 00  ........
30     00 00 00 00 00 00 00 00  ........

90     42 4f 4f 4b 20 50 52 4f  book pro
98     47 52 41 4d 53 a0 a0 a0  grams
a0     a0 a0 31 31 a0 32 41 a0  11  2a
```

The file information starts on track 18 sector 1 and can continue throughout the remainder of the track. Each file uses 32 bytes. Therefore, one sector can hold information for eight entries. With a possible 20 sectors available, information could be held for 160 files. This is unlikely to happen as each file would have to be less then 1K. The directory format is such that bytes 0–31 hold file 1, 32–63 hold file 2, and so on. Each entry is divided up as follows (see Table 5.3 of the 1541 manual):

BYTE	CONTENTS
000–001	Next directory track and sector. A track of 0 indicates last sector in use. These bytes only used for the first entry.
002	The type of file $00=scratched or not in use. $80=DELeted (scratched unclosed) $81=SEQuential $82=PROgraM $83=USeR $84=RELative $1–4=unclosed
003–004	Starting track and sector of file
005–020	Name padded with shifted spaces
023	Record size of relative file
028–029	New track and sector for disk ops with replacement – @
030–031	Number of blocks file uses in low/high byte format

Below is a typical dump of the first directory sector, track 18 sector 1, for two file entries:

START	CONTENTS								ASCII
00	12	04	82	11	00	44	55	4d	.....dum
08	50	2e	4d	49	4b	a0	a0	a0	p.mik
10	a0	a0	a0	a0	a0	00	00	00	...
18	00	00	00	00	00	00	0d	00	....
20	00	00	82	11	03	44	55	4d	.....dum
28	50	2e	4d	a0	a0	a0	a0	a0	p.m
30	a0	a0	a0	a0	a0	00	00	00	...
38	00	00	00	00	00	00	01	00	....

It is worth noting that directory sectors do not follow sequentially. The same is true for file storage, as can be seen when using the disk utility.

Just to round things off, here is a dump of a BASIC program which occupies less than one block. It is in fact the loader for the UTILITY at the end of Chapter 9.

START	CONTENTS								ASCII
00	00	4B	01	08	24	08	0A	00	.K..$...
08	41	B2	41	AA	31	3A	8B	41	A.A.1:.A
10	B2	31	A7	93	22	55	54	49	.1..''UT1
18	4C	49	54	59	20	44	41	54	LITY DAT
20	41	22	2C	38	2C	31	00	3C	A'',8,1.<
28	08	14	00	8B	41	B2	32	A7	....A.2.
30	93	22	55	54	49	4C	49	54	.''UTILIT
38	59	22	2C	38	2C	31	00	47	Y'',8,1.G
40	08	1E	00	9E	33	32	37	36	3276
48	38	00	00	00	A2	00	00	00	8.......

As it is less than one block, the linking track is zero, denoting the end. It is a straight copy of the RAM and like the memory dump in Chapter 1, we can pick out the link addresses, end of program, and so on.

Disk utility

This utility offers many of the housekeeping commands and provides a number of more interesting options. It is rather long as most of the subroutines are complete in themselves (to allow you to extract only those you want for your own programs). The listing has been left in lower case and when you are typing it in, it is easiest to put the 64 into that mode with the shift and logo keys. It makes extensive use of direct access programming so we suggest you use the information given above and the relevant sections of the 1541 manual to follow it. It has been run through the UTILITY's CODER command to produce the mnemonics. Most annotated characters are cursor moves, colours or simply capital letters.

The usual options of NEW, VALIDATE, SCRATCH, INITIALIZE, RENAME, COPY (within a drive) and READ DISK ERROR are all present. The directory option is unusual in that everything is input or displayed in hex notation. A much shorter way to get a directory is given in the BACKUP utility at the end of this chapter. The option also displays the first track and sector of a file, and if it is a program, also its load address. The listing is further split up into directory sectors and will display even SCRATCHed or DELETEd files if the disk has not been VALIDATEd. Two values are given for the BLOCKS FREE – the usual value exclusive of erased files and another inclusive of erased files. An erased file simply has its associated BAM set to 1 (not allocated).

The TRACE option follows a file through displaying its associated tracks and sectors. It will also check to see if the file it is following is scratched. If this is the case, it will ask whether you wish to recover it. If your answer is 'yes', then as it traces it will also allocate each block. Providing that all the blocks found were free it has recovered the file. If an allocated block is found then the original area of the file has been overwritten and recovery is not possible and you will be told. If the scratched file has been successfully traced, all that remains to be done is to use the MOD/DISP BLOCKS option to change its 'file type' byte (third byte in its entry) from 0 back to $81 or $82. The revised directory block must be rewritten to complete the process.

The MOD/DISP BLOCKS option is similar to the demonstration disk's program 'DISPLAY TRACK AND SECTOR'. The main difference is that it also allows the block to be modified and rewritten to disk. When the block is written it is also allocated. The usual options to review again and get the next track and sector are available. The subroutine called at 1680 is a little unusual and merits some explanation. Earlier in the chapter under the heading 'Keyboard' it was mentioned that a file could be opened to it. This eliminates the '?' prompt and also releases the cursor. The cursor may then be moved to the appropriate line, hex values changed and on pressing RETURN the revised values are processed and written to the disk buffer. The same 128 bytes are then redisplayed by reading from the buffer. At the end of a block the option to write the changes must be taken to change it on disk. It is also possible to recover files using this option by following a file through taking the next track and sector option (first two bytes) and always writing the changes. Unlike TRACE, it does not check to see if a file can be recovered. Files in which a READ ERROR occurs may also be reconstructed. This we discovered when the EASY SCRIPT appendices file of this book was corrupted. All we did was modify the next track and sector bytes of the preceding block to skip the corrupt block. The resulting file could be read with the loss of only 256 bytes (and was immediately saved on another disk).

The APPEND for program files is the same as that in Chapter 5 (where it is fully explained). The APPEND for sequential files (and SCRATCH) builds the command string (separated by commas) before actioning on RETURN with no input.

The final option is to modify the disk's header. This is done by simply reading BAM, moving the buffer pointer and writing in the new values. It is worth noting that whenever a byte is read, the buffer pointer is moved forward one position. So in order to write to the same position at which the read started, the pointer must be set using a 'B-P' command.

The utility is not foolproof, but with a little attention to detail, may be used to advantage. Our last comment before the listing is to point out that when you try to allocate an already allocated block error 65 NO BLOCK occurs. This must be checked for and trapped as in MOD/DISP BLOCKS. The locations of all the subroutines may be read from the IF statements in lines 210–340.

```
100 poke 53272,23:poke 53280,6:dim a$(10
0),t$(5)
110 for i=0 to 5:read t$(i):next i
120 data del,seq,prg,usr,rel,???
130 print"[cls][g>d][g>i][g>s][g>k] [g>u
][g>t][g>i][g>l][g>i][g>t][g>y]":print"[
cd][rev][g>n][off] new disk"tab(20);"[re
v][g>h][off] change header"
140 print"[cd][rev][g>v][off] validate d
isk";tab(20);"[rev][g>d][off] directory"
150 print "[cd][rev][g>t][off] trace fil
e";tab(20);"[rev][g>s][off] scratch file
(s)"
160 print"[cd][rev][g>r][off] rename fil
e";tab(20);"[rev][g>e][off] read disk er
ror"
170 print"[cd][rev][g>c][off] copy file"
;tab(20);"[rev][g>a][off] append files"
180 print"[cd][rev][g>b][off] backup fil
e";tab(20);"[rev][g>m][off] mod/disp blo
cks"
190 print"[cd][rev][g>i][off] initialize
 disk";tab(20);"[rev][g>x][off] exit"
200 gosub 2360
210 if y$="n" then gosub 530:goto 130
220 if y$="v" then gosub 580:goto 130
230 if y$="r" then gosub 620:goto 130
240 if y$="s" then gosub 670:goto 130
250 if y$="e" then gosub 740:goto 130
260 if y$="a" then gosub 360:goto 130
270 if y$="c" then gosub 840:goto 130
280 if y$="d" then gosub 890:goto 130
290 if y$="h" then gosub 1240:goto 130
300 if y$="x" then end
310 if y$="m" then gosub 1400:goto 130
320 if y$="i" then gosub 1740:goto 130
330 if y$="t" then gosub 1760:goto 130
340 if y$="b" then gosub 2130:goto 130
350 goto 130
360 rem append
370 print"[cls][g>a][2g>p][g>e][g>n][g>d
][cd]":print"[rev][g>p][off] prg files[c
d]":print"[cd][rev][g>s][off] seq files"
380 gosub 2360:if y$<>"p" and y$<>"s" go
to 520
390 if y$="s" then gosub 770:return
```

```
400 rem prg files
410 print"[cd]append prg files - sure":g
osub 2360:if y$<>"y" goto 520
420 input"[cd]combined prg";f$:input"[3s
pc]first prg";x1$:input"[2spc]second prg
";x2$
430 open 3,8,3,"0:"+f$+",p,w":open 2,8,2
,"0:"+x1$+",p,r"
440 get#2,y$
450 x$=y$:get#2,y$:if st<>0 goto 470
460 gosub 2330:print#3,x$;:goto 450
470 close 2:open 2,8,2,"0:"+x2$+",p,r"
480 get#2,y$:get#2,y$
490 get#2,x$:if st<>0 goto 510
500 gosub 2330:print#3,x$;:goto 490
510 close 2:print#3,chr$(0);:close 3
520 return
530 rem new disk
540 print"[cd]new disk - sure":gosub 236
0:if y$<>"y" goto 570
550 input"[cd]name";f$:input"i.d.";y$:f$
=left$(f$,16)+","+left$(y$,2)
560 open 15,8,15,"n0:"+f$:close 15
570 return
580 rem validate
590 print"[cd]validate - sure":gosub 236
0:if y$<>"y" goto 610
600 open 15,8,15,"v":close 15
610 return
620 rem rename
630 print"[cd]rename - sure":gosub 2360:
if y$<>"y" goto 660
640 input"[cd]old file";f$:input"new fil
e";y$:f$="r0:"+y$+"="+f$
650 open 15,8,15,f$:close 15
660 return
670 rem scratch file
680 print"[cd]scratch - sure":gosub 2360
:if y$<>"y" goto 730
690 f$="":print"[cd]use * or ? for patte
rn matching"
700 print"hit return to delete[cd]"
710 input "delete";y$:if y$<>"" then f$=
f$+","+y$:y$="":goto 710
720 f$="s0:"+mid$(f$,2):open 15,8,15,f$:
close 15
```

```
730 return
740 rem error
750 open 15,8,15:input#15,x$,f$,x1$,x2$:
close 15:print"[cd][g>e][2g>r][g>o][g>r]
:";x$,f$,x1$,x2$
760 gosub 2360:return
770 rem append seq files
780 print"[cd]append seq files - sure":g
osub 2360:if y$<>"y" goto 830
790 print"[cd]hit return to append[cd]"
800 f$="":input"[3spc]new";x$
810 input"append";y$:if y$<>"" then f$=f
$+","+"0:"+y$:y$="":goto 810
820 f$="c0:"+x$+"="+mid$(f$,2):open 15,8
,15:print#15,f$
830 close 15:return
840 rem copy same disk
850 print"[cd]copy - sure":gosub 2360:if
 y$<>"y" goto 880
860 input"[cd]new";f$:input"old";y$:f$="
c0:"+f$+"="+"0:"+y$
870 open 15,8,15,f$
880 close 15:return
890 rem directory
900 print"[cd]directory - sure":gosub 23
60:if y$<>"y" goto 1230
910 open 15,8,15:open 1,8,2,"#":t=18:s=0
:f$="":bf=0:bu=0
920 print#15,"u1";2;0;t;s;:print#15,"b-p
";2;144:print"[cls]";tab(10);"[rev]";
930 for i=1 to 16:get#1,x$:gosub 2290:pr
intx$;:next i:print",";
940 print#15,"b-p";2;162
950 for i=1 to 2:get#1,x$:gosub 2290:pri
ntx$;:next i:print"[off][blk]":t=18:s=1
960 print"$blk file[13spc]type $tk $st $
add[l blu]"
970 print#15,"u1";2;0;t;s:i=0:x=s:gosub
2200:print"[blk]trk 12";" sct ";x$;"[l b
lu]"
980 get#1,x$:gosub 2330:t=asc(x$):get#1,
x$:gosub 2330:s=asc(x$)
990 i=i+1:print#15,"b-p";2;(i-1)*32+2:ge
t#1,x$:gosub 2330:x=asc(x$):y$=""
1000 for j=0 to 5:if x=j then y$=t$(j)
1010 if j=0 then x=x-128
```

```
1020 next j:if y$="" then y$=t$(5)
1030 get#1,x$:gosub 2330:x=asc(x$):gosub
 2200:t$=x$
1040 get#1,x$:gosub 2330:x=asc(x$):gosub
 2200:s$=x$
1050 for j=1 to 16:get#1,x$:gosub 2290:f
$=f$+x$:next j
1060 if y$<>"prg" goto 1090
1070 open 3,8,3,f$+",s,p":get#3,x$:gosub
 2330:x=asc(x$):gosub 2200:l$=x$
1080 get#3,x$:gosub 2330:x=asc(x$):gosub
 2200:l$=x$+l$:close3
1090 print#15,"b-p";2;(i-1)*32+30
1100 get#1,x$:gosub 2330:j=asc(x$)
1110 get#1,x$:gosub 2330:k=asc(x$)
1120 x=k:gosub 2200:bf$=x$:x=j:gosub 220
0:bf$=bf$+x$
1130 bu=bu+256*k+j:if y$<>"del" then bf=
bf+256*k+j
1140 printbf$;" ";f$;"[2spc]";y$;"[2spc]
";t$;"[2spc]";s$;" ";l$:bf$="":f$="":l$=
""
1150 get y$:if y$<>"" then gosub 2360
1160 if i<8 goto 990
1170 if t<>0 goto 970
1180 bf=664-bf:x=bf/256:gosub 2200:y$=x$
:x=bf-256:gosub 2200:y$=y$+x$
1190 print"[yel]";y$;" blocks free[l blu
]"
1200 bu=664-bu:x=bu/256:gosub 2200:y$=x$
:x=bu-256:gosub 2200:y$=y$+x$
1210 print"[yel]";y$;" blocks free inc d
el files[l blu]"
1220 gosub 2360:close 15:close 1
1230 return
1240 rem change header
1250 print"[cd]change header - sure":gos
ub 2360:if y$<>"y" goto 1390
1260 open 15,8,15:open 1,8,2,"#":t=18:s=
0:bf=0:f$=""
1270 print#15,"u1";2;0;t;s;:print#15,"b-
p";2;144:print"[cd]current:[2spc]";
1280 for i=1 to 16:get#1,x$:gosub 2290:p
rintx$;:next i:print",";
1290 print#15,"b-p";2;162
```

```
1300 for i=1 to 2:get#1,x$:gosub 2290:pr
intx$;:next i:print
1310 input"[4spc]name";f$:f$=left$(f$+"[
16g>spc]",16)
1320 input"[4spc]i.d.";y$:y$=left$(y$+"x
x",2)
1330 print#15,"b-p";2;144
1340 for i=1 to 16:print#1,mid$(f$,i,1);
:next i
1350 print#15,"b-p";2;162
1360 print#15,"b-p";2;162
1370 for i=1 to 2:print#1,mid$(y$,i,1);:
next i
1380 print#15,"u2";2;0;t;s:close 15:clos
e 1
1390 return
1400 rem modify and display blocks
1410 print"[cd]modify and display blocks
 - sure":gosub 2360:if y$<>"y" goto 1670
1420 open 15,8,15:open 1,8,2,"#":f$=""
1430 input"[cd] track[cr]$[2cl]";t$:x$=t
$:gosub 2250:t=x:if x<0 or x>40 goto 143
0
1440 input"sector[cr]$[2cl]";s$:x$=s$:go
sub 2250:s=x:if x<0 or x>20 goto 1430
1450 print#15,"u1";2;0;t;s;:print#15,"b-
p";2;0
1460 get#1,x$:gosub 2330:x=asc(x$):tn=x:
gosub 2200:tn$=x$
1470 get#1,x$:gosub 2330:x=asc(x$):sn=x:
gosub 2200:sn$=x$
1480 nt$="[3spc][g>n][g>e][g>x][g>t]:[g>
t][g>r][g>a][g>c][g>k] "+tn$+" [g>s][g>e
][g>c][g>t][g>o][g>r] "+sn$
1490 ct$="[g>c][g>u][2g>r][g>e][g>n][g>t
]:[g>t][g>r][g>a][g>c][g>k] "+t$+" [g>s]
[g>e][g>c][g>t][g>o][g>r] "+s$
1500 print"[cls]";ct$:printnt$
1510 print#15,"b-p";2;0
1520 for i=0 to 15:f$="":x=i*8:gosub 220
0:printx$;" ";:for j=0 to 7
1530 get#1,x$:y$=x$:gosub 2330:x=asc(x$)
:gosub 2200:printx$;" ";:
1540 x$=y$:gosub 2290:f$=f$+x$:next j:pr
intf$:next i
1550 gosub 1680:if y$="y" goto 1500
```

```
1560 print"[cls]";ct$:printnt$
1570 print#15,"b-p";2;128
1580 for i=16 to 31:f$="":x=i*8:gosub 22
00:printx$;" ";:for j=0 to 7
1590 get#1,x$:y$=x$:gosub 2330:x=asc(x$)
:gosub 2200:printx$;" ";:
1600 x$=y$:gosub 2290:f$=f$+x$:next j:pr
intf$:next i
1610 gosub 1680:if y$="y" goto 1560
1620 print"review again":gosub 2360:if y
$="y" goto 1500
1630 print"write changes":gosub 2360:if
y$="y" then print#15,"u2";2;0;t,s
1640 if y$="y" then print#15,"b-a";0;t;s
:gosub 2390
1650 print"next t/s":gosub2360:if y$="y"
thent=tn:s=sn:t$=tn$:s$=sn$:goto 1450
1660 close 1:close 15
1670 return
1680 rem any changes
1690 print"any changes":gosub 2360:if y$
<>"y" goto 1730
1700 open 9,0:input#9,a$:close 9
1710 x$=left$(a$,2):gosub 2250:print#15,
"b-p";2,x
1720 for i=0 to 7:x$=mid$(a$,4+i*3,2):go
sub 2250:print#1,chr$(x);:next i
1730 return
1740 rem intialize disk
1750 open 15,8,15,"i0":close 15:return
1760 rem trace file
1770 print"[cd]trace file - sure":gosub
2360:if y$<>"y" goto 2120
1780 input"[cd]file";bf$
1790 open 15,8,15:open 1,8,2,"#":t=18:s=
1:f$=""
1800 print#15,"u1";2;0;t;s:i=0:f$=""
1810 get#1,x$:gosub 2330:t=asc(x$):get#1
,x$:gosub 2330:s=asc(x$)
1820 i=i+1:print#15,"b-p";2;(i-1)*32+2:g
et#1,x$:gosub 2330:x=asc(x$):y$=""
1830 for j=0 to 5:if x=j then y$=t$(j)
1840 if j=0 then x=x-128
1850 next j:if y$="" then y$=t$(5)
1860 get#1,x$:gosub 2330:x=asc(x$):gosub
 2200:t$=x$
```

```
1870 get#1,x$:gosub 2330:x=asc(x$):gosub
 2200:s$=x$
1880 f$="":for j=1 to 16:get#1,x$:gosub
2290:f$=f$+x$:next j
1890 if left$(f$,len(bf$))=bf$ goto 1930
1900 if i=8 and t>0 goto 1800
1910 goto 1820
1920 if t=0 goto 2120
1930 print"[cls][rev]trace of[off] ";bf$
;" file type ";y$:print:ft$=y$:no=0
1940 if ft$="del" then print"recover fil
e":gosub 2360
1950 print#15,"b-p";2;(i-1)*32+3
1960 get#1,x$:gosub 2330:t=asc(x$):get#1
,x$:gosub 2330:s=asc(x$)
1970 x=t:gosub 2200:t$=x$:x=s:gosub 2200
:s$=x$
1980 print"track ";t$;" sector ";s$
1990 if y$<>"y" goto 2030
2000 print#15,"b-a";0;t;s:input#15,e:if
e<>65 goto 2030
2010 print"recovering not on as a suppos
edly"
2020 print"free block is allocated.":no=
1
2030 print#15,"u1";2;0;t;s
2040 get#1,x$:gosub 2330:t=asc(x$):get#1
,x$:gosub 2330:s=asc(x$)
2050 if t=0 goto 2070
2060 goto 1970
2070 if ft$="del" and y$="y" and no=0 go
to 2090
2080 goto 2110
2090 print"recovery ok remember to chang
e"
2100 print"directory track and file type
"
2110 gosub 2360
2120 close 1:close 15:return
2130 rem backup
2140 print:print"backup file"
2150 print:print"[g>t]o allow larger fil
es to be backed up"
2160 print"on both disk and cassette a s
eparate"
```

```
2170 print"utility has been provided.[3s
pc][g>f]or "
2180 print"smaller files code could be i
ncluded here."
2190 gosub 2360:return
2200 rem dec-hex
2210 x=x and 255:x1=int(x/16):x2=x and 1
5
2220 x1$=chr$(48+x1):if x1>9 then x1$=ch
r$(55+x1)
2230 x2$=chr$(48+x2):if x2>9 then x2$=ch
r$(55+x2)
2240 x$=x1$+x2$:return
2250 rem hex-dec
2260 x$=right$("00"+x$,2)
2270 x1=asc(x$)-48:x2=asc(mid$(x$,2))-48
2280 x=16*(x1+7*(x1>9))+x2+7*(x2>9):retu
rn
2290 rem convert to ascii
2300 if x$=chr$(160) goto 2320
2310 if x$<" " or x$>"z" then x$="."
2320 return
2330 rem eliminate null string
2340 if x$="" then x$=chr$(0)
2350 return
2360 rem wait
2370 get y$:if y$="" goto 2370
2380 return
2390 rem error on b-a check
2400 input#15,en$,em$,et$,es$:if en$<"20
" or en$="65" then return
2410 close 15:print:printen$,em$,et$,es$
:gosub 2360:run
```

Many more options could be provided and some of the existing options could be made fully automatic. These are exercises for you to carry out.

BACKUP

We have produced this utility to allow selective backing-up of files between disk and tape, from disk to disk and from tape to tape. Commodore provide an excellent 1541 BACKUP program on the demonstration disk, but it only backs up whole disks. The following allows selective backing-up of single files, whether they be program or sequential. It could be modified to do more than one file when going between disk and tape, providing that the details of each file were known in advance. We wrote the program to avoid the need to produce a special program for sequential files and the use of an extended monitor to copy machine code. BASIC programs can, of course, be duplicated by a simple load and save sequence.

The program is in two parts: the machine code and the BASIC driver which uses the machine code for program files. The following describes the driver:

LINE	ACTION
100	Set top of memory to $1800/6144
110	Set source device and check whether valid
120	Do same for destination device
130	Contents of disk or tape
150	Set prog or seq file, if not known use 130
170–	Go to appropriate subroutine for source, destination and file type

Subroutines

250–	Seq file from disk to tape so read byte/write byte until status says end of file. Requires a file name.
290–	Seq tape to disk
290	Read header and display info with sub 700 and if non-ASCII in name then offer chance to change name.
310	Final check on name.
330–	Read and write bytes until status says end of file
370–	Seq tape to tape. Has a limited capacity.
370	Check for non-ASCII and option to change name.
410–	Read in bytes until end of file or until RAM filled eliminating ASC(0) on the way to avoid errors.
450	Warning – only part of file read.
480	Write to destination tape
490	Pause for destination tape
510–	Seq disk to disk. Same principle as for tape above
610	Pause for destination disk
630	Simple wait for any key
650	Print TAPE or DISK in set up
690	Contents of next file on tape and prompt to rewind
700	Display tape buffer in full – highlighting any non-ASCII
750	Get file name

760 Eliminate trailing spaces in file name

800 Display directory of disk

940– Prog file disk to disk.
940 Get file name and set it up in cassette buffer
950 Fill up rest of buffer with spaces
960 Set length of name register – FNLEN and enter m/c to do a relocated load
980 Delete file from destination disk
990 Do relocated save

1010– Prog disk to tape. Initial part as for disk to disk
1040 Write header created
1045 Write RAM

1060 Prog tape to tape

1100 Prog tape to disk checking as before

BASIC PROGRAM

```
100 POKE 52,24:POKE 56,24:CLR
110 PRINT"[CLS]BACKUP UTILITY":PRINT"[CD
]FROM T/D";:GOSUB 650:F$=Y$
120 PRINT"[CD][2SPC]TO T/D";:GOSUB 650:T
$=Y$
130 PRINT"[CD]CONTENTS T/D":GOSUB 630:IF
 Y$="D" THEN GOSUB 800:GOTO 130
140 IF Y$="T" THEN GOSUB 690:GOTO 130
150 PRINT"[CD]TYPE P/S";:GOSUB 630:IF Y$
<>"P" AND Y$<>"S" GOTO 150
160 FT$=Y$:PRINT " ";FT$
170 IF F$="D" AND T$="T" AND FT$="S" THE
N GOSUB 250:RUN
180 IF F$="T" AND T$="D" AND FT$="S" THE
N GOSUB 290:RUN
190 IF F$="D" AND T$="D" AND FT$="S" THE
N GOSUB 510:RUN
195 IF F$="T" AND T$="T" AND FT$="S" THE
N GOSUB 370:RUN
200 IF F$="D" AND T$="D" AND FT$="P" THE
N GOSUB 940:RUN
210 IF F$="D" AND T$="T" AND FT$="P" THE
N GOSUB 1010:RUN
220 IF F$="T" AND T$="T" AND FT$="P" THE
N GOSUB 1060:RUN
230 IF F$="T" AND T$="D" AND FT$="P" THE
N GOSUB 1100:RUN
```

```
240 RUN
250 GOSUB 750:OPEN 2,8,2,N$+",S,R":OPEN
1,1,1,N$
260 GET#2,Y$:IF ST GOTO 280
270 PRINT#1,Y$;:GOTO 260
280 CLOSE 2:CLOSE 1:RETURN
290 OPEN 1,1,0:GOSUB 700:IF E=1 THEN GOS
UB 750:GOTO 310
300 GOSUB 760
310 PRINT"[CD]FILE NAME ";CHR$(34);N$;CH
R$(34);" OK Y/N":GOSUB 630
320 IF Y$="N" THEN GOSUB 750
330 OPEN 3,8,3,"@0:"+N$+",S,W"
340 GET#1,Y$:IF ST GOTO 360
350 PRINT#3,Y$;:GOTO 340
360 CLOSE 1:CLOSE 3:RETURN
370 Y=6144:OPEN 1,1,0:GOSUB 700:IF E=1 T
HEN GOSUB 750::GOTO 390
380 GOSUB 760
390 PRINT"[CD]FILE NAME ";CHR$(34);N$;CH
R$(34);" OK Y/N":GOSUB 630
400 IF Y$="N" THEN GOSUB 750
410 GET#1,Y$:IF ST GOTO 460
420 IF Y$="" THEN Y$=CHR$(0)
430 POKE Y,ASC(Y$)
440 IF Y<40959 THEN Y=Y+1:GOTO 410
450 PRINT"[CD]FILE TOO BIG ONLY 34K COPI
ED"
460 CLOSE1
470 GOSUB 490
480 OPEN 1,1,1,N$:FOR I=6144 TO Y:PRINT#
1,CHR$(PEEK(I));:NEXT I:CLOSE 1:RETURN
490 PRINT"[CD]DESTINATION TAPE Y":GOSUB
630:IF Y$<>"Y" GOTO 490
500 RETURN
510 GOSUB 750:OPEN 2,8,2,N$+",S,R":Y=614
4
520 GET#2,Y$:IF ST GOTO 570
530 IF Y$="" THEN Y$=CHR$(0)
540 POKE Y,ASC(Y$)
550 IF Y<40959 THEN Y=Y+1:GOTO 520
560 PRINT"[CD]FILE TOO BIG ONLY 34K COPI
ED"
570 CLOSE2
580 GOSUB 610
```

```
590 OPEN 3,8,3,"@0:"+N$+",S,W":FOR I=614
4 TO Y:PRINT#3,CHR$(PEEK(I));:NEXT I
600 CLOSE 3:RETURN
610 PRINT"[CD]DESTINATION DISK Y":GOSUB
630:IF Y$<>"Y" GOTO 610
620 RETURN
630 GET Y$:IF Y$="" GOTO 630
640 RETURN
650 GOSUB 630:IF Y$="T" THEN PRINT" TAPE
":GOTO 680
660 IF Y$="D" THEN PRINT" DISK":GOTO 680
670 GOTO 650
680 RETURN
690 PRINT"[CD]":OPEN 1:CLOSE 1:GOSUB 700
:PRINT"[CD][REV]REWIND TAPE[OFF]":RETURN
700 PRINT"TYPE FILENAME";SPC(10);"BUFFER
 START":I=PEEK(828):E=0
710 Y$=" PRG ":IF Y=4 THEN Y$=" SEQ "
720 PRINTY$;"[REV]";:FOR I=833 TO 1019:X
=PEEK(I):IF X<32 OR X>95 THEN X=63:E=1
730 PRINT CHR$(X);:IF I=849 THEN PRINT"[
OFF]<";
740 NEXT I:PRINT">[REV]END[OFF]":RETURN
750 INPUT "[CD]FILENAME";N$:N$=LEFT$(N$,
16):RETURN
760 N$="":FOR I=848 TO 833 STEP -1:X=PEE
K(I)
770 IF X=32 AND N$="" GOTO 790
780 N$=CHR$(X)+N$
790 NEXT I:RETURN
800 PRINT"[CLS]";:OPEN 1,8,0,"$0":GET#1,
Y$,Y$
810 I=0:GET#1,Y$,Y$,Y$,X$:IF Y$<>"" THEN
 I=ASC(Y$)
820 IF X$<>"" THEN I=I+ASC(X$)*256
830 PRINTRIGHT$("[2SPC]"+STR$(I),3);" ";
:I=0
840 GET#1,Y$:IF ST GOTO 930
850 IF Y$=CHR$(34) THEN I=I+1:PRINT CHR$
(34);:GOTO 840
860 IF I=0 GOTO 840
870 IF I=1 THEN PRINT Y$;:GOTO 840
880 IF I=2 THEN PRINT TAB(22);:I=I+1
890 IF I=3 AND Y$=" " GOTO 840
900 IF Y$<>"" THEN PRINT Y$;:GOTO 840
```

```
910 PRINT:GET Y$:IF Y$<>"" THEN GOSUB 63
0
920 IF ST=0 GOTO 810
930 PRINT "BLOCKS FREE":CLOSE 1:GOSUB 63
0:RETURN
940 GOSUB 750:FOR I=1 TO LEN(N$):POKE 83
2+I,ASC(MID$(N$,I,1)):NEXT I
950 FOR I=833+LEN(N$) TO 1019:POKE I,32:
NEXT I
960 POKE 183,LEN(N$):SYS 49244
970 PRINT"[CD][REV]FILE WILL BE DELETED[
OFF]":GOSUB 610
980 OPEN 15,8,15,"S0:"+N$:CLOSE 15
990 POKE 183,LEN(N$):SYS 49343
1000 RETURN
1010 GOSUB 750:FOR I=1 TO LEN(N$):POKE 8
32+I,ASC(MID$(N$,I,1)):NEXT I
1020 FOR I=833+LEN(N$) TO 1019:POKE I,32
:NEXT I
1030 POKE 183,LEN(N$):SYS 49244
1040 SYS 49203
1045 SYS 49206
1050 RETURN
1060 SYS 49152
1070 GOSUB 490
1080 SYS 49203
1085 SYS 49206
1090 RETURN
1100 SYS 49152
1110 GOSUB 700:IF E=1 THEN GOSUB 750:GOT
O 1130
1120 GOSUB 760
1130 PRINT"[CD]FILE NAME ";CHR$(34);N$;C
HR$(34);" OK Y/N":GOSUB 630
1140 IF Y$="N" THEN GOSUB 750
1150 PRINT"[CD][REV]FILE WILL BE DELETED
[OFF]":GOSUB 610
1160 OPEN 15,8,15,"S0:"+N$
1170 POKE 183,LEN(N$):SYS 49343
1180 RETURN
```

The following is the BASIC loader for the machine code and must be loaded and run before using the above program.

```
1 DATA 32, 44, 247, 173, 60, 3, 133, 255
, 169, 0, 133
2 DATA 193, 169, 24, 133, 194, 56, 173,
63, 3, 237
3 DATA 61, 3, 170, 173, 64, 3, 237, 62,
3, 168
4 DATA 24, 138, 101, 193, 133, 174, 152,
 101, 194, 133
5 DATA 175, 32, 162, 245, 165, 255, 141,
 60, 3, 96
6 DATA 32, 183, 247, 169, 0, 133, 193, 1
69, 24, 133
7 DATA 194, 56, 173, 63, 3, 237, 61, 3,
170, 173
8 DATA 64, 3, 237, 62, 3, 168, 24, 138,
101, 193
9 DATA 133, 174, 152, 101, 194, 133, 175
, 32, 124, 246
10 DATA 96, 169, 96, 133, 185, 169, 1, 1
41, 60, 3
11 DATA 133, 184, 169, 8, 133, 186, 169,
 0, 133, 195
12 DATA 133, 147, 169, 65, 133, 187, 169
, 3, 133, 188
13 DATA 169, 24, 133, 196, 164, 183, 32,
 175, 245, 32
14 DATA 213, 243, 165, 186, 32, 9, 237,
165, 185, 32
15 DATA 199, 237, 32, 19, 238, 141, 61,
3, 32, 19
16 DATA 238, 141, 62, 3, 32, 232, 244, 1
65, 174, 141
17 DATA 63, 3, 56, 165, 175, 233, 24, 14
1, 64, 3
18 DATA 24, 173, 61, 3, 109, 63, 3, 141,
 63, 3
19 DATA 173, 62, 3, 109, 64, 3, 141, 64,
 3, 96
20 DATA 169, 97, 133, 185, 169, 1, 133,
184, 169, 8
21 DATA 133, 186, 169, 65, 133, 187, 169
, 3, 133, 188
22 DATA 165, 185, 164, 183, 32, 213, 243
, 32, 143, 246
23 DATA 165, 186, 32, 12, 237, 165, 185,
 32, 185, 237
```

```
24 DATA 169, 0, 133, 172, 169, 24, 133,
173, 56, 173
25 DATA 63, 3, 237, 61, 3, 133, 174, 173
, 64, 3
26 DATA 237, 62, 3, 133, 175, 24, 169, 2
4, 101, 175
27 DATA 133, 175, 173, 61, 3, 32, 221, 2
37, 173, 62
28 DATA 3, 160, 0, 32, 33, 246, 96, 255,
 255, 0
29 FOR I=49152 TO 49432:READ A:POKE I,A:
NEXT I
```

Once this has been run it could be saved as its machine code for later ease of loading. A detailed description of the machine code follows.

MACHINE CODE

The machine code is called by the driver as required. It consists of four parts:

i) Read any header and do relocated load
ii) Write to tape current header and write relocated code
iii) Load from disk retaining original details but relocate
iv) Save to disk relocated code with original details

Chapter 5 of the *Programmer's Reference Guide*, 'The KERNAL', discusses the use of LOAD and SAVE in detail. The entry points given are for complete loads and saves (it is possible to do a relocated load, but not a relocated save using these). Unfortunately, as we are using an all-purpose BASIC driver, these entry points may overwrite it. To overcome this problem, every operation is carried out in two stages. The first is to read or write the file's details which are always stored in or taken from the cassette buffer. This avoids having to do too much moving of information. The second is to perform a relocated load or save with the correct amount of code going to or being taken from $1800 on.

To do this we must enter the load and save routines at much later points with the parameters already set. It would consume too much space to describe these in detail, so we leave it up to you to follow them through. The only tricks are to prevent a forced load to its original address with a tape marker of 3 and to prevent a header being written with a marker of 5 (when an end of tape has been written – see Tape Headers).

TAPE

Read any tape header without loading

```
C000  202CF7   JSR  $F72C  read any header
C003  AD3C03   LDA  $033C  get sec add
C006  85FF     STA  $FF    and store for later
C008  A900     LDA  #$00   set start of load
C00A  85C1     STA  $C1    to $1800 by setting
C00C  A918     LDA  #$18   STAL
C00E  85C2     STA  $C2
C010  38       SEC         subtract MSBs
C011  AD3F03   LDA  $033F  of start and end
C014  ED3D03   SBC  $033D  of original load
C017  AA       TAX         put result in X
C018  AD4003   LDA  $0340  do same for LSBs
C01B  ED3E03   SBC  $033E  putting answer in Y

C01E  A8       TAY
C01F  18       CLC         find overall length
C020  8A       TXA
C021  65C1     ADC  $C1    and add to STAL
C023  85AE     STA  $AE    to give the new
C025  98       TYA         end i.e. $1800
C026  65C2     ADC  $C2    plus result
C028  85AF     STA  $AF
Load from $1800 on
C02A  20A2F5   JSR  $F5A2  do the relocated load
C02D  A5FF     LDA  $FF    restore the sec add
C02F  8D3C03   STA  $033C  in case end of tape 5
C032  60       RTS         load complete
Write to tape in two parts the correct header and then
the code from $1800 on
C033  20B7F7   JSR  $F7B7  write header in orig form
C036  A900     LDA  #$00   reset STAL as it
C038  85C1     STA  $C1    has been changed by
C03A  A918     LDA  #$18   writing the header
C03C  85C2     STA  $C2
C03E  38       SEC         recalc the relocated end
C03F  AD3F03   LDA  $033F  MSB
C042  ED3D03   SBC  $033D
C045  AA       TAX
C046  AD4003   LDA  $0340  LSB
C049  ED3E03   SBC  $033E
C04C  A8       TAY
C04D  18       CLC
C04E  8A       TXA
C04F  65C1     ADC  $C1    set up EAL
```

```
C051  85AE      STA  $AE
C053  98        TYA
C054  65C2      ADC  $C2
C056  85AF      STA  $AF
Save RAM for reloading
C058  207CF6    JSR  $F67C  save RAM
C05B  60        RTS         complete
```

DISK

```
Load from disk relocated to $1800
C05C  A960      LDA  #$60
C05E  85B9      STA  $B9    set sec add
C060  A901      LDA  #$01   put type 1 in tape buffer
C062  8D3C03    STA  $033C
C065  85B8      STA  $B8    make log file 1
C067  A908      LDA  #$08   make device 8
C069  85BA      STA  $BA    and put in FA
C06B  A900      LDA  #$00
C06D  85C3      STA  $C3
C06F  8593      STA  $93    A=0 for load
C071  A941      LDA  #$41   set pointer to file name
to file name
C073  85BB      STA  $BB    in FNADR to TBUFFR + 5
C075  A903      LDA  #$03
C077  85BC      STA  $BC
C079  A918      LDA  #$18   MEMUSS set to $1800
C07B  85C4      STA  $C4    for relocated load
C07D  A4B7      LDY  $B7    read len name set in BASIC
C07F  20AFF5    JSR  $F5AF  print SEARCHING
C082  20D5F3    JSR  $F3D5  print LOADING ....
C085  A5BA      LDA  $BA    get current device
C087  2009ED    JSR  $ED09  send talk
C08A  A5B9      LDA  $B9    get sec add
C08C  20C7ED    JSR  $EDC7  send talk sec add
C08F  2013EE    JSR  $EE13  receive from serial
C092  8D3D03    STA  $033D  and store LSB in TBUFFR
C095  2013EE    JSR  $EE13  do same for MSB of start
C098  8D3E03    STA  $033E
C09B  20E8F4    JSR  $F4E8  do relocated load
C09E  A5AE      LDA  $AE    get end LSB
C0A0  8D3F03    STA  $033F  put in TBUFFR
C0A3  38        SEC         subtract relocated start
C0A4  A5AF      LDA  $AF    and end
C0A6  E918      SBC  #$18   and put in appropriate
C0A8  8D4003    STA  $0340  locations of TBUFFR
```

```
C0AB  18       CLC
C0AC  AD3D03   LDA  $033D  leaving a header
C0AF  6D3F03   ADC  $033F  suitable for
C0B2  8D3F03   STA  $033F  a tape write
C0B5  AD3E03   LDA  $033E
C0B8  6D4003   ADC  $0340
C0BB  8D4003   STA  $0340
C0BE  60       RTS         back to BASIC
Save relocated code to reload at correct address
C0BF  A961     LDA  #$61   set parameters
C0C1  85B9     STA  $B9
C0C3  A901     LDA  #$01
C0C5  85B8     STA  $B8
C0C7  A908     LDA  #$08
C0C9  85BA     STA  $BA
C0CB  A941     LDA  #$41
C0CD  85BB     STA  $BB
C0CF  A903     LDA  #$03
C0D1  85BC     STA  $BC
C0D3  A5B9     LDA  $B9
C0D5  A4B7     LDY  $B7
C0D7  20D5F3   JSR  $F3D5  send sec add
C0DA  208FF6   JSR  $F68F  print SAVING
C0DD  A5BA     LDA  $BA    send listen
C0DF  200CED   JSR  $ED0C  device 8
C0E2  A5B9     LDA  $B9    send listen
C0E4  20B9ED   JSR  $EDB9  sec add
C0E7  A900     LDA  #$00   set up SAL
C0E9  85AC     STA  $AC    with $1800
C0EB  A918     LDA  #$18
C0ED  85AD     STA  $AD
C0EF  38       SEC
C0F0  AD3F03   LDA  $033F  calculate prog length
C0F3  ED3D03   SBC  $033D  and put
C0F6  85AE     STA  $AE    in EAL
C0F8  AD4003   LDA  $0340
C0FB  ED3E03   SBC  $033E
C0FE  85AF     STA  $AF
C100  18       CLC
C101  A918     LDA  #$18   add $1800 to
C103  65AF     ADC  $AF    give relocated
C105  85AF     STA  $AF    end
C107  AD3D03   LDA  $033D
C10A  20DDED   JSR  $EDDD  send serial deferred
C10D  AD3E03   LDA  $033E  send actual start in A and Y
```

```
C110  A000      LDY  #$00
C112  2021F6    JSR  $F621   save RAM to reload
C115  60        RTS          back to BASIC
```

The utility is only intended for your own files. It will not as it stands backup relative files.

As a point of interest, Supersoft's ZOOM monitor offers not only the option to perform relocated loads and saves, but to save in a form suitable for reloading on a PET, which eliminates an ID of 3 not used on the PET.

3 A token approach to BASIC

Introduction

In this chapter we deal with the five main routines BASIC uses in interpreting your programs or commands. One of these, CHRGET, picks up single bytes from the program and is called by the majority of routines in your 64. The other four routines covered are concerned with keywords – converting from ASCII to tokens, the reverse process (LISTing), and directing them to their respective routines.

Other than using SYS commands a knowledge of these routines is essential if you wish to extend existing commands or add further ones. Those of you owning a disk drive will be familiar with the program DOS 5.1 and may know that this modifies CHRGET to trigger its commands.

CHARGET

BASIC gets its information from the input or program lines through a routine called CHRGET (CHaracterR GET). A copy of this routine is held in the KERNAL operating system and is copied into zero page on power-up. Each time BASIC wants a character it calls this routine.

The routine is held at locations \$0073–\$008A (KERNAL is \$E3A2–\$E3B9) and is as follows:

```
$0073  E6 7A     INC $7A
$0075  D0 02     BNE $0079
$0077  E6 7B     INC $7B
$0079  AD 00 02  LDA $2000
$007C  C9 3A     CMP #$3A
$007E  B0 0A     BCS $008A
$0080  C9 20     CMP #$20
$0082  F0 EF     BEQ $0073
$0084  38        SEC
$0085  E9 30     SBC #$30
$0087  38        SEC
$0088  E9 D0     SBC #$D0
$008A  60        RTS
```

Bytes 0073–0077

Every time the CHRGET routine is called it increases, by one, the location from where it gets its information. After it increments the LSB, in location 007A, it checks whether the page has been crossed, that is, from $FF to $00. Only if it has will the MSB be increased.

Bytes 0079–007B
Here it takes the information from store and puts it into the accumulator. The store location is present before the initial entry to the routine. It is always set one byte less due to initial increment. If you were going to use the routine yourself, it would be these bytes you would change, as we shall see later.

Bytes 007C–007F
Here it checks to see if the character is a numeral. It is testing to see if it is greater than ASCII numeral 9 ($39). If so then the routine is left via $008A with the carry set.

Bytes 0080–0083
Here is a straightforward test to see if a space was picked up; if so the routine is carried out again. CHRGET cannot be left on encountering a space.

Bytes 0084–0089
These successively subtract two numbers from the original byte and end up with the same number. You may say that is senseless but it will set two flags in the status register that help us later. These are the carry flag and the zero flag.

The carry flag. If this is clear on exit the byte will be a numeral in ASCII form. If it is set we have something else.

When subtracting two numbers in machine code the carry flag must always be set first. If the number we are subtracting from is the larger then the carry will remain set. On the other hand, if the number we are subtracting is the larger the carry will clear.

In this case we have already eliminated any byte that has a higher ASCII value than the numeral 9, in bytes 007E and 007F. It now subtracts $30 (ASCII for digit zero) from the accumulator in preparation for setting the final flags which can be used for testing for numbers. The carry flag at this point does not matter as it is set again anyway. Bytes with ASCII values lower than numerals now range between $D0 and $FF.

With the next subtraction the original value is restored. The carry flag will now be set or unset as BASIC requires. As numerals are the only ones less than $D0 (the last figure subtracted) they will be the only ones to clear the carry flag.

The zero flag. This could be set in two instances. First, in bytes 007E and 007F where we tested our byte against $3A, the ASCII value for a colon. If it was a colon then the zero flag would have been set as it was equal (the carry would also have been set). Secondly, the flag would be set after the second subtraction if, and only if, the original byte was zero (not ASCII digit zero). In that instance after the first subtraction the accumulator would hold $D0 and subtracting the same value would set our zero flag.

A colon in a BASIC line signifies an end of statement and a zero, the end of a line, and hence an end of statement. Therefore, by testing the zero flag, we can quickly tell if we have reached the end of that particular instruction.

CHRGOT

Keyword routines are entered immediately after a call to CHRGET. Before using that byte it may require the accumulator for something else. To recover the CHRGET byte we can use the CHRGOT routine. This is a shortened version of CHRGET. If, instead of entering the routine at $0073, we enter at $0079, we miss out the instructions which update the pointer and get the original byte again.

Wedges

If we want to patch in our own machine code routines to work alongside BASIC, one way to do this is to insert a wedge. Simply, this is a routine which diverts CHRGET to check whether it is one of our additions. From this a decision can be made whether to revert to the normal CHRGET flow or to a routine of our own.

Let us say that we put in some routines all to be actioned on the character '@'. For instance, we could have a renumber routine and a delete routine. The command for renumber might be '@R' and delete, '@D' and could place a wedge at $C000.

The first thing we have to do is alter the CHRGET routine. We want to change it after it has collected a byte but before it starts checking and manipulating it. The alteration would therefore be at $007C with a JMP to our coding. The routine starts with six bytes of data with our changes and the original bytes. (The latter is for restoring CHRGET if you require so to do later). We can load them by using the load with the X register. Our first instructions will look like this:

```
C000  4C 11 C0     JMP $C011
C003  C9 3A        CMP #$3A
C005  B0           BCS
C006  A2 02        LDX #$02
C008  BD 00 C0     LDA $C000,X
C00B  95 7C        STA $7C,X
C00D  CA           DEX
C00E  10 F8        BPL $C008
C010  60           RTS
C011 Our coding will start here
```

This is the routine to initialize our wedge and is called immediately after loading the program by using SYS 49158 ($C006). This loads a byte X places from $C000 and stores it X places from $7C. We decrease the counter X and the branch to collect the next byte will work until we decrement it below zero, that is, no longer a positive number. If the branch fails, we go back to BASIC through the RTS.

The beginning of the CHRGET now looks like this:

```
0073  INC $7A
0075  BNE $0079
0077  INC $7B
0079  LDA $0200  (this number varies)
007C  JMP $C011
```

Now each time CHRGET is used it will go to our routine. As all our commands would be triggered with '@', the first thing we would do is to check to see if it is present:

```
C011   C9 40        CMP   #$40
C013   F0 03        BEQ   $C018
```

If it is, we can branch to do further checks, and if not, we continue with the next code. Here we will have to revert to the normal course of events. We have two options which we can take. First, we include the bytes of CHRGET we changed into our program – we do not want to change CHRGET itself as we want it to use again – and jump back to CHRGET at $0082. Secondly, we can use the CHRGET routine in the KERNAL ROM, jumping in at $E3AB, and BASIC will continue as if nothing has happened.

The first method would be like this:

```
C015   4C AB E3     JMP   $E3AB
```

And the second like this (of course, the routine address will change from $C018 to $C01D):

```
C013  F0 07        BEQ $C01D
C015  C9 3A        CMP #$3A
C017  B0 03        BCS $C01C
C019  4C 80 00     JMP $0080
C01C  60           RTS
```

We now want to find out if it is one of our routines. This we do by checking the next character without updating the CHRGET pointer (in case it isn't). The code would look like this:

```
C018  08           PHP
C019  48           PHA
C01A  98           TYA
C01B  48           PHA
```

```
C01C  8A          TXA
C01D  48          PHA
C01E  A6 7A       LDX $7A
C020  E8          INX
C021  BD 00 02    LDA $0200,X
C024  C9 52       CMP #$52
C026  F0 ??       BEQ - TO RENUMBER ROUTINE
C028  C9 44       CMP #$44
C02A  F0 ??       BEQ - TO DELETE ROUTINE
C02C  68          PLA
C02D  AA          TAX
C02E  68          PLA
C02F  A8          TAY
C030  68          PLA
C031  28          PLP
C032  4C AB E3    JMP $E3AB
```

Bytes C018–C01D
Here we are preserving our registers, including the status register, on the stack in case it is not destined for our own routines.

Bytes C01E–C023
Location $7A has the LSB of the pointer used by CHRGET and this is one less than the next character we want. So if we load it into the X register and increase it by one, we will have the position of the next byte. We can now load the byte using X as a pointer.

Bytes $C024–$C2B
A check is made to see if it is the letter R, signifying the renumber routine. If not we then check for the letter D and if so go to delete.

Bytes $C2C–$C034
It not, we restore our registers from the stack (in the reverse order we put them on). Then we continue the KERNAL routine as before.

One last point is that in the routines, such as renumber or whatever, it would be advisable to call a subroutine to remove the bytes from the stack (placed there in $C018 to $C01D). We do not require them, but as your routines are called the stack will become fuller and fuller, resulting in an 'OUT OF MEMORY' error.

Keywords

A more professional approach to adding routines than altering CHRGET is the use of *Keywords*. This approach holds with the idea behind the BASIC language that actions can be performed by using words which are indicative of the desired action.

Keywords can be divided into two types: commands and functions.

Functions get information, for example, PEEK returns the contents of a location, and will always supplement a command keyword. For example, we use PRINT PEEK(xx) but never PEEK(xx)PRINT.

On the 64 it is possible to incorporate new routines actioned by keywords as they go to the relative routines through vectors held in RAM. As the vectors are in RAM we can change them to go to routines of our own.

The other items we will have to add are three tables of data. One will have the new keywords in ASCII code. This will be used when LISTing and tokenizing. The end of a word is indicated by adding $80 (128) to its last letter. To signify the end of the table, a zero is used. If we had a table of two keywords, say END and NOT, it would look like this:

45 4E C4 4E 4F D4 00

The other two tables will have the addresses of our routines, one for command keywords and one for functions. These will hold the address of each routine less 1. The reason for this is that we will put them on the stack for an RTS instruction. The program counter will add 1 when it takes them from the stack, thus getting the correct address for the routine. The table will have two bytes for each routine, the LSB and then the MSB. For example, if we had a routine at $C4DF, on the table it will look like this: DE C4.

There are four vectors we will have to change (three if not adding functions).

ADD OF VECTOR	ADD OF ROM	DESCRIPTION
$0304/5	$A57C	Tokenize BASIC
$0306/7	$A71A	Print Tokens(LIST)
$0308/9	$A7E4	Token Dispatch – Command words
$030A/B	$AE86	Token Dispatch – Function words

Tokenize BASIC Text

The object of this subroutine is to take an input line, check it for keywords, tokenize them and condense the line. It does this by taking every byte from the input buffer, not using CHRGET, and then checking through the keyword table for a match. If the letters do not make a keyword, it stores them as variables, meaning that variables cannot have keywords in them.

There are two ways we could approach the problem of incorporating our own keywords and tokens. First, we could copy the BASIC from ROM into RAM, and alter the tokenize routine within BASIC so that if it cannot find a match it jumps to a routine to check through our table of keywords. This would mean we would not have to change the vector but would lose the RAM area under the BASIC ROM which is useful for storing hires screens, data tables, and so on. The second way would be to change the vector to a routine of our own. Here we have a copy of the ROM routine altered slightly to be able to search our table as well. We would only use the ROM routine when we finished tokenizing a whole line.

We shall describe the second method, which we think is the better in the long run. A description follows the code.

```
                LDX $7A
                LDY #$04
                STY $0F
ANOTHER         LDA $0200,X
                BPL SPACE
                CMP #$FF
                BEQ STORE
                INX
                BNE ANOTHER
SPACE           CMP #$20
                BNE STORE
                STA $08
                CMP #$22
                BEQ QUOTE
                BIT $0F
                BVS STORE
                CMP #$3F
                BNE NUMBER
                LDA #$99
                BNE STORE
NUMBER          CMP #$30
                BCC CONT
                CMP #$3C
                BCC STORE
CONT            STY $71
                LDY #$00
                STY $0B
                DEY
                STX $7A
                DEX
NEXT LETTER     INY
                INX
CONT 1          LDA $0200,X
                SEC
                SBC $A09E,Y
                BEQ NEXT LETTER
                CMP #$80
                BNE NEXT WORD
STORE A         ORA $0B
FOUND           LDY $71
```

```
420 STORE        INX
430              INY
440              STA $01FB,Y
450              LDA $01FB,Y
460              BEQ END
470              SEC
480              SBC #$3A
490              BEQ COLON
500              CMP #$49
510              BNE DATA
520 COLON        STA $0F
530 DATA         SEC
540              SBC #$55
550              BNE ANOTHER
560              STA $08
570 LINE         LDA $0200,X
580              BEQ STORE
590              CMP $08
600              BEQ STORE
610 QUOTE        INY
620              STA $01FB,Y
630              INX
640              BNE LINE
650 NEXT WORD    LDX $7A
660              INC $0B
670 FIND         INY
680              LDA $A09D,Y
690              BPL FIND
700              LDA $A09E,Y
710              BNE CONT 1
720              LDY #$FF
730              DEX
740 NEXT         INY
750              INX
760 NEXT B       LDA $0200,X
770              SEC
780              SBC $START OF OUR WORD TABLE,X
790              BEQ NEXT
800              CMP #$80
810              BNE NEXT NEW
820              BEQ STORE A
830 NEXT NEW     LDX $7A
840              INC $0B
850 NEXT A       INY
860              LDA $START OF OUR WORD TABLE -1,X
870              BPL NEXT A
```

```
880                LDA $START OF OUR WORD TABLE,X
890                BNE NEXT B
900                LDA $0200,X
910                BPL FOUND
920 END            JMP $A609
```

LINES 10–30: Initialization. Location $7A will have a value the same as the position within the input buffer. In immediate mode this will be 0, the start of the buffer. If inputting a program line, it would be after the line number, which has already been taken care of. Now X will be our pointer to the original contents of the buffer and Y will be the pointer to our new buffer set up. Y is stored in $0F just to initialize that location. The value does not matter as long as it was not over $3F, as we shall see.

LINES 40–50: We load in a byte and check to see if it is under $80 (128). If it is, we branch off to line 100.

LINES 60–90: Values of $80 and over arrive here and we check to see if it is 'PI' ($FF). If it is, we branch further into the routine to store it. If not, we branch back to get another byte. This means we cannot use the first character of a keyword as a shifted letter because it would be greater than $80.

LINES 100–140: If a space is found we branch off to store it, otherwise we store the accumulator in a register, in case the following check succeeds, for comparison later. Now we check to see if the byte is a quote. Items between quotes do not require tokenizing and therefore we branch and continually store them until we come across another quote or the end of the line.

LINES 150–160: Here we are checking to see if location $0F has bit 6 set or not. Amongst other things, the BIT instruction takes bit 6 of $0F and places it in bit 6 of the status register, the overflow register. This bit in $0F will only be set in this routine if we tokenize DATA later in the routine. It means that after DATA all characters will be stored and do not go through the keyword table. Bit 6 can be unset by a colon if outside quotes and for this reason colons have to be in quotes within DATA statements. A colon outside quotes will mean that information after is tokenized. BASIC instructions can be placed at the end of a data line and will be actioned as the line is encountered. BASIC differentiates between DATA and REM. On REM it will go to the next line whereas with data it will search through it for another BASIC instruction.

LINES 170–200: The '?' is the shortened version of the keyword PRINT. It is the only keyword which can be shortened to a single character. These lines check for the question mark and if found place the token for PRINT, $99, in the accumulator and go to store it.

LINES 210–240: Here we find out if the byte is a numeral, colon or semicolon. If it is off we go to store it, if not then we continue the routine.

LINES 250–300: We now set up for the search through the table of keywords. We store our Y register, which, if you remember, is the pointer to the 'new look' buffer. Location $0B will be our counter to the number of keywords we encounter; it will not hold the token value but helps determine it. We store X in $7A. This is part of CHRGET, but we are only using it as a store. We decrease both X and Y as the first part of the next section will increment them.

LINES 310–390: This section of the routine explains why on the Commodore 64 we can use shortened keywords using shifted letters. There are two things to remember here. First, the last letter of the keyword in the table is the value of the letter plus $80 (128) which when loaded will set the negative flag. Secondly, the value of a shifted letter (not logo shift) is also the value of the letter plus $80.

Back in the routine we increase the registers and load in our byte again; it will later load the next byte. We set the carry for subtraction and subtract the value of a letter, from the keyword table, from our byte. If we are left with zero then we have a match and we go back to get the next letter from the buffer. If it fails, we check to see if we have the value $80 left. This would indicate we have a match by either the input letter being the shifted letter or we have reached the end of the keyword in the table and have also matched. Failing this second check, it branches off to find the end of that word so we can check the next for a match.

As the second letter can be shifted to give our match this explains not only shortened keywords, but also why some require at least three. As an example, take the keywords CLR and CLOSE. CLR comes before CLOSE in the table so that C and shift L will match with the former before it gets to CLOSE, which will therefore need two standard letters before a shifted letter. We can also explain why there is no shortened version for INPUT. INPUT# comes before INPUT and so any shortened version will always match with INPUT#.

LINES 400–460: Back to the routine now. We have in the accumulator the value $80 and have found a match in the table. Here we perform the logical OR of the accumulator and location $0B. Later on we will find out that every time we pass through a keyword that does not match, we increase the value of $0B. As we started at 0, $0B will have the number of the match word, the first word in the table being 0. The instruction OR forces bits into the accumulator if they are not set. In this case it has the same effect as adding but saves bytes doing it this way. The accumulator always has $80 and if you OR it with a value of one you get $81. This is how we arrive at the token value. The keyword GO is the last word in the table and that is the 76th, giving a value in $0B of $4B (75), and ORing it with $80 gives a value of $CB which is the token value of GO.

Having got our token value, we load back into the Y register from $71 – the pointer to the new buffer layout. We have now reached the point where we store a lot of the characters into the new buffer layout. This is the point where many of the earlier branches arrive. First, we increase both buffer pointers. The base location to store is $01FB indexed with Y. As we started with Y equal to $04 and increase it immediately, our first location will be $01FB + 5, giving $0200. We will not overwrite anything in the original line we have not checked for two reasons. First, if there was a line number at the beginning it has been dealt with and is no longer needed. Secondly, the routine only shortens and never lengthens.

Once the byte has been stored, it is loaded back in and checked for zero which signifies the end of the line.

LINES 470–550: This section is going to test if the byte we have stored is either a colon or one of the keywords DATA or REM. If it is a colon, it unsets bit 6 of location $0F which we discussed earlier, then goes to get more bytes. DATA will set the 6th bit of that location before getting more bytes.

REM is slightly different in that nothing after it requires tokenizing. We set the location of $0B to zero, as that is the result of the subtractions, which we will not actually need for checking but which stops us from branching out of the next section for any reason other than the end of line.

LINES 560–640: These lines are used only in two instances: on encountering a quote or encountering REM. All this does is to move bytes from their original to their new position in the input buffer, until we reach the end of the line or, in the case of a quote, we find a closing quote. Finding either of these, we branch back to the normal store lines of 420–460. The branch in line 640 is enforced as X can never be zero as before we get here we have looked at a minimum of two bytes so the least X can be is 1 and the maximum $58 (the maximum input line length).

LINES 650–710: This is the section we come to if we did not find a keyword match. All this does is to search for the next character in the table that has a value greater than $80, and return with its position. The pointer will be increased before we start another match. It also checks for a 0, signifying the end of the table.

So far the routine is the same as in ROM but now we change the course of events. In the ROM routine, when it finds the end of the table it assumes the characters are variables and stores them as such. We, on the other hand, want to see if it is one of our keywords, so on getting zero we have to search our table. The Y register is loaded at the beginning of the section because either way we want to get back the first character of this particular check from the original input buffer line up.

LINES 720–910: This is a repeat of the checking of the standard keywords except it will have the address of our keyword table. We will now be checking for our keywords.

LINE 920: When we have found and stored the end of line zero, we get here. The routine now jumps off to end or to the original ROM routine. There it will reset the CHRGET pointers to their initial setting of $01FF and continue the normal flow of BASIC to either store the line or carry out its instructions if in direct mode.

Print tokens

This routine is part of the LIST routine in BASIC. It takes the token value, finds the keyword and prints it to the screen, or other device. The ROM print token routine is not a subroutine by itself but an integral part of LIST, but thankfully it is vector-started. The vector points to the next instruction in the ROM routine. What we would have to do is to change the vector to a routine of our own, PRINT keywords of either the standard ones or our own, and then jump back to the LIST routine at an appropriate point. The coding for such a routine is as follows:

```
                BPL ROM 1
                CMP #$FF
                BEQ ROM 1
                BIT $0F
                BMI ROM 1
                CMP #$CC
                BCC CBM TOKEN
                SEC
                SBC $CB
                TAX
                LDA # LSB START OF OUR KEYWORD TABLE
                STA $22
                LDA #$MSB START OF OUR KEYWORD TABLE
                STA $23
                BNE START
CBM TOKENS      SEC
                SBC #$7F
                TAX
                LDA $9E
                STA $22
                LDA #$A0
                STA $23
START           STY $49
                LDY #$FF
NEXT WORD       DEX
```

```
260               BEQ WORD FOUND
270 NEXT CHAR     INY
280               LDA ($22),Y
290               BPL NEXT CHAR
300               BMI NEXT WORD
310 WORD FOUND    INY
320               LDA ($22),Y
330               BMI ROM 2
340               JSR $AB47
350               BNE WORD FOUND
360 ROM 1         JSR $A6F3
370 ROM 2         JSR $A6EF
```

LINE 10: This tests the negative flag. A value of $80 (128) or over is signalled as negative. As all tokens are $80 or over, this branch will succeed; values under $80 go back to LIST unchanged.

LINES 20: $FF is the value of 'PI'. If it is that value we again return to LIST.

LINES 40–50: What we are doing here is putting bit 7 of location $0F into the negative flag, although it does other things which are of no concern to us. Location $0F is the flag used by the LIST routine to signal if it is listing in quotes or not. If bit 7 of $0F is 1, then it is listing in quotes and we do not want to print tokens but the ASCII of the bytes. Therefore, if the negative flag is set, we branch to go back to ROM.

LINES 60–70: Here we find out whether it is one of the standard tokens or one of ours. It will branch off if it is standard.

LINES 80–150 OUR TOKENS

80–100: Here we subtract a number that is one less than our first token value. The result is then transferred to the X register to act as a counter. The value of X is one greater than the position in the table (starting at 0) but X will be decreased before we start the search.

110–140: We store the start address of our table in what will be our search registers.

150: This is enforced as the last figure in the accumulator, the MSB of our table, will not be zero. We are hardly likely to have a keyword table in the zero page which has many important BASIC locations.

LINES 160–220 CBM TOKENS: This is a duplicate of lines 80–140 except it is for the standard tokens and keyword table.

LINES 230–240: So far we have not used or altered the Y register but we store it here in location $49 for the LIST routine as that is where it will expect to find it later. We initialise Y with $FF but will increase it before our search so it will start a zero.

It may be worth a note here that we will not alter the values of the search registers, $22 and $23, as the 64's keyword table is not longer than

256 bytes and it is unlikely that ours would be. Therefore, incrementing Y through its 256 range (0–FF) will serve our purpose. It also saves bytes and time.

LINES 250–260: Every time we read a word from the table we will come here and decrease the X register. If X is zero, then we have found the position one byte before the keyword we want. In that case we branch off to PRINT the keyword.

LINES 270–300: Here we increase our table pointer, the Y register, and then load in the next character from the table. Remember that the last character of a keyword is its ASCII form plus $80 (128) and this is what we look for. This will set the negative flag in the status register.

The first check is to see if the negative flag is unset signifying a branch back to get the next character. If the negative is set, then the end of the word is found and we branch back to test X to see if we have come far enough. One of these two branches must work as a byte is determined as either negative or positive.

LINES 310–350: We have found our word and now have to print it out. First we increase our pointer to pick up the first character. We load it and test to see if it is the last character. If it is we go to the BASIC ROM to have it printed through the LIST routine. Failing this, we go to another ROM routine to have the character printed. We will return with the same character in the accumulator. As a keyword will not have a byte of zero value, the branch in line 350 is enforced, to get another character to print.

LINE 360–ROM 1: The character was not a token at the beginning so we go back to the LIST routine to have it printed and continue with the listing.

LINE 370–ROM 2: We have here the last character of the keyword in the accumulator. Now we go back to the LIST routine where it will be turned into the proper ASCII value, printed and the listing continued.

BASIC token dispatch

This is the routine that BASIC uses on finding a token to get the address for the routine. It deals only with command keywords, such as PRINT. It is a subroutine in itself. What we need to do is put in a routine that it goes to first, through the vector.

```
10          JSR $0073
20          CMP #$CC
30          BCC ROM
40          CMP #HIGHEST COMMAND TOKEN VALUE
50          BCS ROM
60          JSR DISPATCH
70          JMP $A7EA
```

```
80  DISPATCH    SEC
90              SBC #$CC
100             ASL
110             TAY
120             LDA START OF OUR VECTOR TABLE+1,Y
130             PHA
140             LDA START OF OUR VECTOR TABLE,Y
150             PHA
160             JMP $0073
170 ROM         JSR $0079
180             JMP $A7E7
```

LINE 10: Get the token from the input buffer or program line through the CHRGET routine.

LINES 20–30: We check to see if it is one of our tokens. If it is not, we branch off to the normal routine in ROM.

LINES 40–50: Now we find out if it is a command or a function token of ours. If it is a function vector, then it is a 'SYNTAX ERROR' so we branch to ROM to print it.

LINES 60–70: Here we go to our subroutine for dispatch. When the keyword routine has been completed, the program flow will come back here where we shall jump back to BASIC for continuation.

LINES 80–150: We subtract our lowest token value from the value we have. This will give us values of 0 upwards. Now as each routine has a two byte address, we must double our 'new' token value to get its proper place in the vector table. The instruction ASL does just this by shifting all bits one place left and putting a zero in bit 0. This new value is transferred to the Y register as a pointer in the table. What we are going to do is to put a new return address on top of the stack (the program counter expects the LSB on top with the MSB underneath). Therefore, we take the second byte of the table first, put it on the stack, and then the first. Remember our vector table is made up of LSB then the MSB.

We now JMP to the CHRGET routine to pick up the next byte. The RTS at the end of CHRGET will now be to our keyword routine as we have just put its address on the stack. We came to these lines (80–150) by a JSR command so its return address was originally on the top of the stack. We then put another address on top of that which was pulled off in CHRGET leaving our original return address once more at the top. At the end of the keyword routine this address will be pulled off and we will return to line 70 of this routine.

LINES 170–180: Here we go to the normal dispatch routine. This is not the address normally found in the Token Dispatch Vector because we will miss out the first instruction which is to get the next byte. We go to CHRGOT first not to get the byte we have already got but to set the flags that the ROM routine wants to test.

BASIC function dispatch

This is the routine that will find the routine addresses of function keywords.

```
10                LDA #00
20                STA $0D
30                JSR $0073
40                CMP #$LOWEST FUNCTION TOKEN VALUE
50                BCC ROM
60                CMP #$HIGHEST TOKEN VALUE
70                BCS ROM
80                JSR DISPATCH
90                RTS
100 DISPATCH      SEC
110               SBC #$LOWEST FUNCTION TOKEN VALUE
120               ASL
130               TAY
140               LDA #$START OUR FUNCT VECT TABLE+1,Y
150               PHA
160               LDA #$START OUR FUNCT VECT TABLE,Y
170               PHA
180               JMP $0073
190 ROM           JSR $0079
200               JMP $AE8D
```

This is basically the same as the previous routine. The return addresses to ROM are different, as will be the table address. The first two lines load a location which BASIC uses to decide whether to accept numeric or string data, the latter value would be $80.

The other difference is that on return from the function routine we will arrive back at line 90. The previous routine went back to BASIC for another command, but here we RTS as functions will be performed as part of a command routine and therefore we go back to it.

4 Keyboard revisited – making use of the wasted keys

On the far right of your keyboard there are four keys that do not really do much, at least at the moment. They are, of course, the function keys. In this chapter we are going to show you how to make use of them. First we thought it a good idea to describe the ROM routine in the 64 which services the keyboard. In doing so we will also come across the locations that appertain to the keys.

The hardware interrupt vector

Every 1/60th second the computer hands control to an interrupt system. When the microprocessor receives an interrupt signal it will not do anything until the present instruction has been completed. The processor will then save the program counter and the status register. The program counter is then loaded with the contents of locations $FFFE and $FFFF. This will start a routine at $FF48 which saves the register contents on the stack before doing an indirect jump to the vector at $0314 and $0315.

The interrupt routine found at this vector points to address $EA31. This KERNAL routine performs several housekeeping operations such as the update of the system clock, but it also scans the keyboard. The key that you press is picked up by the Complex Interface Adapter #1, and in particular the Data Port B within that chip. From this the value of the key pressed, and shift keys if used, is calculated and stored.

There seems to be some doubt from what we have read about which location the current key value is stored in. The current value is stored in $CB (203) and the last in $C5 (197). This to the BASIC programmer does not make a lot of difference unless the key buffer is full when the key value is not logged except in $CB. The shift, logo and CTRL keys have the same system, with the current location being $028D (653) and the last press in $028E (654).

Having stored your current input it will check to see if it is the same as the last key press. Its next action will depend on whether it is the same, or not. If the same, it will see if it is a repeat function such as the cursor keys or if location $02BA (650) has been set for all keys to repeat. Failing this, the value will not be placed into the keyboard buffer. Where the key values are processed it does so by looking up a table to obtain the ASCII code for your key press. This value is placed in the keyboard buffer and its counter updated.

The keyboard buffer is situated at $0277–$0280, a size of ten characters. It operates on the system that the first character in will also be the first out. The pointer for the number of characters in the buffer at a particular time is $C6 (198). The size of the buffer can be reduced from its initial value of ten by setting register $0289 (649).

Earlier we said that every key has a value. These are from 0 to 64, the latter being no key press. A table of these, and the shift key values are given in the appendices.

Here is a summary of keyboard locations:

$CB	203	Current key press.
$C5	197	Last key pressed.
$028D	653	Current shift etc.
$028E	654	Last shift etc.
$028A	650	Repeat flag: $80 all, $00 normal.
$0277–$0280	631–640	Keyboard buffer.
$0289	649	Size of keyboard buffer.
$C6	198	No of chars in buffer.

The Function Keys

These keys have values and ASCII codes like any other key. They are:

Function key	Value ($CB and $C5)	ASCII
F1	4	133
F2	4	137
F3	5	134
F4	5	138
F5	6	135
F6	6	139
F7	3	136
F8	3	140

Knowing these values and the locations mentioned earlier, we can make use of the function keys.

Function keys within a BASIC program

One of the most used BASIC statements for evaluating a key press is the GET function. This function returns the ASCII code for the first key in the keyboard buffer, or the latest key if the buffer is empty. It will not wait for a key press. A BASIC routine could look like this:

```
100 GET A$
110 IF A$="[F1]" THEN 1000:REM ACTION ON F1 PRESS
120 REM ACTION IF ANY OTHER KEY PRESSED
```

This routine will not stop and wait for a key press. It will only branch off to line 1000 if key F1 is pressed at the same time as the GET statement is actioned or the next character in the keyboard buffer is the ASCII for F1.

We could adapt this so that it will wait until a key is pressed – any key.

```
100 GET A$ :IF A$=""THEN 100
```

Here line 100 will be repeated until one key is pressed or there is a value in the key buffer that has not been read.

The next thing we could add is a line to clear the input buffer before we GET a character. The easiest way is to set the register for the number of buffer characters to zero.

```
90 POKE 198,0
```

At the moment the routine actions on any key. If we wanted it to action on only two keys, say F1 and F7, we would have to alter line 120 to:

```
120 IF A$ <> "[F7]" THEN 90
130 REM ACTION ON F7 PRESSED
```

Now the routine will wait until a key is pressed. Once a key is pressed it goes to 110 to see if it was F1 and branches if so. Failing that it goes to line 120 where we look to see if it was not F7. On F7 the program will continue its flow. Now lines 90–120 will keep repeating until either F1 or F7 is pressed.

The only other alteration we could do is to rid ourselves of the graphic characters in the quotes that represent the function keys. This would make it easier for someone else to read and on a non-Commodore printer the graphic character would not print. This we can do by using the CHR$(function when checking A$. Line 110 would now look like:

```
110 IF A$ = CHR$(133) THEN 1000: REM
    ACTION ON F1 PRESSED
```

In the GET statement all eight function keys can be tested in the same way, either by changing the character in the quotes or changing the CHR$ value.

Another way of testing for the keys is by examining the key press registers set in the interrupt routines. From a BASIC programmer's viewpoint it does not really matter whether you test the current or the last key register. The snag with this method is that without checking the shift register only four of the function keys can be detected. On the other hand, by checking the shift register with all its combinations you can have up to 32 function key combinations. Here is a routine that tests for function key F1:

```
 90 POKE 198,0:REM CLEAR KEY BUFFER
100 IF PEEK(203)=4 THEN 1000 : REM F1 VALUE.
110 REM PROG CONTINUES IF NOT F1.
```

This will not wait for F1. To wait, line 110 will have to be changed to:

```
110 GOTO 90
```

We have now set up a loop and the only exit is F1 being pressed. Now if we wanted to test for F2, the shift flag would have to be introduced. Line 100 could look like this:

```
100 IF PEEK(203)=4 AND PEEK(648)=1 THEN
     1000 : REM ACTION ON F2 PRESSED
```

If you wanted to go to line 1000 on any key, or no key, apart from F2, then the equals sign should be replaced by greater than and less than signs.

Programming the keys in immediate mode

Our interrupt routine

The routines that follow will allow you to program the function keys with commands or phrases to be actioned as if you typed them in full, but using only one keystroke.

Most of the routines we have seen to do this operation change the vector address of the Hardware Interrupt Routine in $0314 and $0315. They alter it to point to their routine, which when finished will return direct to the normal interrupt routine. This course of action has drawbacks. First, it adds to the length of the interrupt, especially if the user's routine has to be completely followed through. Secondly, it means that you have to set up your own registers for checking to see if it was the same action as the last time or not, to avoid auto-repeat. A further drawback is that if you want to use the data assigned to function keys within quotes, it is more difficult to suppress the graphic character that is generated in the quotes mode along with your phrase.

So how are we going to achieve this desirable routine of making the function keys really useful? Earlier we described the interrupt routine and how your key presses are interpreted. What we did not say was that there is a vectored jump within it. This occurs after the value from the Data Port is put into the current key registers but before it is actioned. The vector is held in addresses $028F and $0290 (655 and 656) and is known as the 'Keyboard Table Setup Vector'. If we change the address in this vector to point to a routine of our own we can process the data first. If the data concerns us we can process it jumping back to the normal interrupt routine at a point which misses out the normal key press routine. When the data does not concern us, control will be handed back to the normal flow of things.

The use of the vectors by Commodore has allowed us an easy way to program the keys. This cannot be said of the values that have been assigned to the function keys. It would have been easier if F1 had a value of 1 and F3, of 3, but this is not the case.

We are going to have 16 programmed function keys. To get this number, you have to use the keys in conjunction with the shift and logo keys as follows:

KEYS F1, F3, F5, F7	– THE KEY ONLY
KEYS F2, F4, F6, F8	– THE KEY + SHIFT
KEYS F9, F11, F13, F15	– THE KEY + LOGO
KEYS F10, F12, F14, F16	– THE KEY + SHIFT + LOGO

This gives us keys in the range of 1 to 16, but for the routine it is easier to use 0 to 15. We shall load the data into the keyboard buffer so we are limited to ten characters. We also require a marker for the end of data for each key, which will be a zero, meaning a maximum 11 bytes storage for each. It is easier, and quicker, to use 16 bytes per key. This wastes five bytes but as we are going to store the data in the RAM under the BASIC ROM this is unimportant. This will mean the value of the key needs to be multiplied by 16 to get the start of its data. Multiplying by 16 for the low numbers we are using, 0 to 15, simply involves moving the four lower bits to the four higher bits and filling the lower ones with zeros, four ASL instructions will achieve this. Sixteen bytes of data for the 16 keys will take one page, 256 bytes, exactly.

To summarise:

i) Find out if the key is a function key, yes – continue, no – go to interrupt.
ii) Calculate key number less 1.
iii) Multiply key number by 16 for table position.
iv) Get data off the data table and store in the key buffer.

ASSEMBLY LISTING

```
  9 *=$8722
 10             LDY $CB      ! CURRENT KEY PRESS
 20             CPY #$03     ! IS IT A FUNCTION KEY
 30             BCC NORMAL   ! NO
 40             CPY #$07     ! IS IT A FUNCTION KEY
 50             BCC CONT     ! YES
 60 NORMAL      JMP $EB48    ! NORMAL INTERRUPT
                               KEY ROUTINE
 70 CONT        LDA $028D    ! CURRENT SHIFT PRESS
 80             CPY $C5      ! IS CURRENT KEY=LAST
 90             BNE CONT2    ! NO
100             CMP $028E    ! IS CURRENT SHIFT=LAST
110             BEQ NORMAL   ! KEY AND SHIFT AS LAST
```

```
120 CONT2           STY $C5      ! STORE CURRENT KEY
                                   IN LAST REGISTER
130                 STA $028E    ! STORE CURRENT
                                   SHIFT IN LAST REG
140                 CPY #$04     ! IS IT F1
150                 BEQ F1+1     ! YES
160                 CPY #$05     ! IS IT F3
170                 BEQ F3+1     ! YES
180                 CPY #$06     ! IS IT F5
190                 BEQ F5+1     ! YES
200                 LDY #$07     ! IT IS F7
210 F1              BIT $01A0    ! VALUE FOR F1
220 F3              BIT $03A0    ! VALUE FOR F3
230 F5              BIT $05A0    ! VALUE FOR F5
240                 CMP #$02     ! WHAT SHIFT
250                 BCC NOCHANGE! NONE OR SHIFT -
                                   VALUES CORRECT
260                 BEQ CBM+1    ! LOGO KEY
270                 LDA #$09     ! VALUE FOR SHIFT+LOGO
280 CBM             BIT $08A9    ! VALUE FOR LOGO
290 NOCHANGE        STY $BB
300                 DEC $BB      ! ONLY WANT NO'S 0-15
310                 CLC
320                 ADC $BB      ! GET FINAL VALUES
330                 ASL A        ! MULTIPLY VALUE BY 16
340                 ASL A
350                 ASL A
360                 ASL A
370                 LDY #$A1     ! HIGH ADDR KEY TABLE
380                 STY $15
390                 LDY #$00
400                 STY $14      ! LOW ADDRESS
410                 TAY          ! TRANSFER TO Y AS
                                   POINTER
420                 LDX #$00     ! COUNTER KEY BUFFER
430 NEXT            JSR $81FB    ! SWITCH OFF BASIC
440                 LDA ($14),Y  ! GET BYTE OF DATA
450                 PHA          ! STORE TEMP
460                 JSR $8202    ! SWITCH ON BASIC
470                 PLA          ! GET BACK DATA
480                 BEQ $EXIT    ! END OF DATA
490                 CMP #$5F      ! ARROW FOR RETURN
500                 BNE $STORE   ! NO
510                 LDA #$0D     ! LOAD CODE FOR RETURN
520                 STA $0277,X  ! STORE IN KEYBOARD
                                   BUFFER
```

```
530             INX          ! INCREASE COUNTER
540             INY          ! INCREASE POINTER
550             BNE NEXT     ! FORCED-GET NEXT DATA
560 EXIT        STX $C6      ! NO OF CHARS IN KEY
                               BUFFER
570             LDA #$7F     ! RESET CIA DATA PORT
580             STA $DC00
590             RTS
```

```
875F CBM             872F CONT
873B CONT2           8791 EXIT
874E F1              8751 F3
8754 F5              8778 NEXT
8762 NOCHANGE        872C NORMAL
878A STORE
```

LINES 10–60: What we do here is to get into the Y register the value of the current key press and see if it is a function key or not. Function keys have values from 3 to 6 inclusive. Line 60 has the normal address of the Keyboard Table Setup Vector and if we do not find a function key this is where we direct the flow.

LINES 70–110: This part of the routine checks to see if the last KEY is the same as the current KEY. If it is, then off to the standard routine to avoid auto-repeat. At this point we have the current key value in the Y register and the current shift value in the accumulator.

LINES 120–130: This is part of the housekeeping. We copy the current values we have obtained into the last key registers. This is not only for our routine but also for the normal key interpreting routine.

LINES 140–230: We now take our key value, find which key it is and give a number corresponding to the number on the key itself. The BIT commands will not alter any data at all except for the status flags (which we are not testing here). They allow us to 'hide' an instruction within the address, in these cases loading the Y register, saving bytes and branch instructions. For instance, the BIT address in line 210 is $01A0 which is stored in memory as A0 01, which is the code for LDY #$01.

LINES 240–280: We now do the same for the shift value. If there is no shift or just the standard shift, there will be no need for any alterations so they would branch off in line 250. The logo key requires the value of 8 (1+8 giving key 9 and so on) and both shifts 9. We again do this using the BIT function.

LINES 290–360: Here we subtract one from the key value and then add the result to the shift value, ending up with a value between 0 and 15.

This total will be in the accumulator which is then increased 16 times by the four ASL instructions. We now end up with a value between 0 and 15 which will be the pointer to the data for that particular key.

LINES 370–420: The start position of the data table is put in registers $14 and $15. We also transfer the pointer in the accumulator to the Y register. Lastly, we initialize the X register to zero to use as a counter to the number of characters we put in the keyboard buffer.

LINES 430–550: At last we can get our data and use it. Earlier we said that we were going to put our data in the RAM under the BASIC ROM. To read it back, we have to 'remove' the ROM to access it. This we do by a call to an earlier routine in the UTILITY which you will come to later. Now we pick up a byte of data and put it on the stack for temporary safe keeping, as we require the accumulator for re-enabling the BASIC ROM. With the ROM back, and having recovered the byte, we have two checks before storing it. The first in line 480 is to see if the byte is zero, signifying that all the relevant data has been collected and we can finish up. The second is a check for the 'left arrow', which signifies the user wants a return to be included (more of this in programming the keys). If this succeeds, we will change the byte to the ASCII code for return.

The data is stored in the keyboard buffer starting at the beginning and working upwards – it will be removed in the same order. We do not need to check for overflow as we are only allowed ten characters to be programmed (see next section). Therefore, the zero, which is not stored, cannot be later than the eleventh byte.

Having stored our byte, the two registers are increased by one and we branch back to get a further byte. The branch is enforced as we will not increase Y enough to return it to a zero. The highest value Y will achieve is $FB (251 dec).

LINES 550–590: The end is near. Having stored all our data, the X register will hold a number equal to the total number of characters we put into the buffer. This is put into the register denoting how many characters are in the buffer and the operating system will only take that many off. The following two instructions are again housekeeping in that we reset the data port for collection of the next press. A return follows, but didn't we come by a JMP? This is true, but the whole key routine is entered by a JSR where the vectored jump is found. We do not now need the use of the normal key interpreting routine so we can go straight back to the main interrupt.

Key

COMMAND SNYTAX
KEY

Displays the current data assigned to the keys in a form which can be amended.

KEY[number between 1 and 15], "[data]"

Assign data to a particular key. If a return is required, type a "←' to signify this. Quotes cannot be used as data. A typical command could look like this:

KEY 7, "LIST←"

Here is a full list of the key numbers and how to achieve them:

```
KEY  1  – F1 ONLY
KEY  2  – F1 + SHIFT
KEY  3  – F3 ONLY
KEY  4  – F3 + SHIFT
KEY  5  – F5 ONLY
KEY  6  – F5 + SHIFT
KEY  7  – F7 ONLY
KEY  8  – F8 + SHIFT
KEY  9  – F1 + LOGO KEY
KEY 10  – F1 + SHIFT + LOGO KEY
KEY 11  – F3 + LOGO KEY
KEY 12  – F3 + SHIFT + LOGO KEY
KEY 13  – F5 + LOGO KEY
KEY 14  – F5 + SHIFT + LOGO KEY
KEY 15  – F7 + LOGO KEY
KEY 16  – F7 + SHIFT + LOGO KEY
```

KEY 0... will generate a SYNTAX ERROR. We had thought about using this as a way of turning off the key routines, but decided on a separate command. This makes it more of a conscious decision rather than a typing error. The command will be OFF, which is discussed later.

We have seen that we can make use of the four 'mystery' keys by getting data output on their use and in fact having 16 keys when used with the shift and logo keys. Now we have a routine to program the data, in which the user can decide what data to apply. This operation is acted upon through the keyword KEY.

KEY will perform three functions. It will 'switch' on the keys if they are off. This is performed in both of the following options. The choices are to program a key or to display the data applied to all the keys, which can then be amended on the display.

As we have said, there are two routines included in this. There is one routine to program individual keys and one routine to display the data assigned to all keys. The latter is very similar to the interrupt routine discussed earlier except that the data goes to the screen rather than a buffer. The former in many ways is the reverse: we take data from a buffer – the input buffer – and put it in a tabie.

ASSEMBLY LISTING

```
  9 *=$864D
 10            LDA $805B    ! CHECK IF INTERRUPT
                              SET FOR KEYS
 20            CMP #$87
 30            BEQ START    ! YES
 40            LDA #$87
 50            STA $805B
 60            LDA #$22
 70            STA $8056
 80            JSR $8054    ! SET INTERRUPT
 90 START      JSR $0079    ! GET LAST BYTE AGAIN
100            BEQ DISPLAY
110            JSR $81F5    ! GET PARAMETER
120            JSR $AEFD    ! CHECK FOR COMMA
130            LDA $14
140            BEQ SYNTAX   ! NO KEY0
150            CMP #$11     ! HIGHEST KEY IS 16
160            BCS SYNTAX
170            DEC $14
180            LDA $14      ! SET TO CALCULATE
                              POINTER
190            ASL A        ! CALCULATING POINTER
200            ASL A
210            ASL A
220            ASL A
230            TAY
240            LDA #$A1     ! HI ADD FOR KEY TABLE
250            STA $15
260            LDA #$00     ! SET LO ADD FOR KEYS
270            STA $14
280            LDX $0A      ! COUNTER MAX NO OF
                              CHARS
290            JSR $0079    ! GET LAST BYTE AGAIN
300            CMP #$22     ! IS IT A QUOTE
310            BEQ CONT2    ! YES
320 SYNTAX     JMP $AF08    ! PRINT SYNTAX ERROR
330 CONT2      JSR $0073    ! GET NEXT BYTE
340            BEQ ZERO     ! END OF DATA INPUT
350            CMP #$22     ! IS IT A QUOTE
```

```
360          BEQ ZERO     ! END OF DATA INPUT
370          STA ($14),Y  ! STORE DATA IN
                                 TABLE
380          INY          ! INC TABLE POINTER
390          DEX          ! DEX CHAR COUNT
400          BNE CONT2    ! IF ZERO MAX NO
                            CHARS REMAINDER
                            IGNORED
410 ZERO     LDA #$00     ! END OF WORD MARKER
420          STA ($14),Y
430          JSR $0073    ! GET NEXT BYTE
440          RTS          ! FINISHED
450 DISPLAY  LDX #$00     ! SET COUNTER
460          STX $5F
470          INX
480          LDA #$20     ! SPACE AS NO TEN'S
                            DIGIT
490          STA $22
500          LDA #$31     ! ASCII FOR ONE
510          STA $23
520          LDA #$00     ! LO BYTE OF DATA TABLE
530          STA $14
540          LDA #$A1     ! HI BYTE OF DATA TABLE
550          STA $15
560 PD1      JSR PRINT
570          INC $23      ! INCREASE NUMERAL
580          INC $5F      ! INCREASE KEY COUNT
590          INX
600          CPX #$0A     ! HAVE WE DONE KEYS1-9
610          BCC PD1      ! NO
620          LDA #$31     ! NOW HAVE A TEN DIGIT
630          STA $22
640          LDA #$30
650          STA $23
660 PD2      JSR PRINT
670          INC $23
680          INC $5F
690          INX
700          CPX #$11     ! HAVE WE DONE 16
710          BCC PD2      ! NO
720          RTS          ! YES
730 PRINT    LDY #$05     ! COUNTER
740 NEXTA    LDA PDATA,Y  ! PRINT " KEY "
750          JSR $FFD2
760          DEY
770          BNE NEXTA
```

```
780          LDA $22
790          JSR $FFD2     ! PRINT TEN'S
                             NUMERAL OR SPACE
800          LDA $23       ! PRINT LOW NUMERAL
810          JSR $FFD2
820          LDA #$2C
830          JSR $FFD2     ! PRINT COMMA
840          LDA #$22
850          JSR $FFD2     ! PRINT QUOTE
860          LDA $5F       ! CALC TABLE POINTER
870          ASL A
880          ASL A
890          ASL A
900          ASL A
910 CONT     TAY           ! PUT POINTER IN Y
920 NEXT     JSR $81FB     ! SWITCH OFF BASIC
930          LDA ($14),Y   ! GET CHAR OFF TABLE
940          PHA           ! TEMP STORE
950          JSR $8202     ! SWITCH ON BASIC
960          PLA           ! RETRIEVE CHAR
970          BEQ EXIT      ! FOUND END OF WORD
980          JSR $FFD2     ! PRINT CHAR
990          INY
1000         BNE NEXT      ! ENFORCED
1010 EXIT    LDA #$22
1020         JSR $FFD2     ! PRINT A QUOTE
1030         RTS
1040 PDATA   BYT $20,'Y,'E,'K,$20,$0D
```

```
8691 CONT2        86A8 DISPLAY
8716 EXIT         8704 NEXT
86E0 NEXTA        86BD PD1
86D1 PD2          871C PDATA
86DE PRINT        8661 START
868E SYNTAX       86A0 ZERO
```

LINES 10–80: Earlier in the UTILITY, a routine will exist that is used when the extension is initialized or when STOP/RESTORE is used. This sets the Keyboard Table Setup Vector to where we want it to point to. These addresses can be changed by the OFF command. Here we look to see if the high byte of the address is pointing to our interrupt routine. If not, we change the address in the setting routine to point to our interrupt routine and then call the setting routine to initialize.

LINES 90–270: A call first to the CHRGOT routine to get the byte after the

KEY token. This is necessary as we have used the accumulator and so overwritten the byte. If the byte has set the zero flag, then there are no further parameters and a display of the key data is required. The program in that case branches to the display which starts at line 450.

Knowing we have got some parameters, off we go to our 'GET PARAMETER' routine (Chapter 6) and to a ROM coding to see if the byte after the key number is a comma. This coding will not only update the CHRGET address but will generate a SYNTAX ERROR if a comma is not found.

The parameter we want is now held in location $14 – the key number. This value is put in the accumulator and checked for two things. If it is zero or greater than sixteen, it is out of bounds, so an error message is required, and therefore we branch off to get this printed. As in the interrupt routine it is easier to work in numbers 0 to 15 rather than 1 to 16 so we decrease the value in $14 by one and then reload back into the accumulator.

To get the pointer to the required position in the data table, the number is multiplied by 16 with the ASL instructions. The Y register will be the pointer so the value is transferred to it. Next we load two registers with the address of the data table start. Now we are in a position to get, and store, the data.

LINES 280–440: The data generated by using the function keys will be placed in the keyboard buffer. This buffer is only ten characters in length so we have to limit the input to that number. This is achieved by setting the X register to ten ($0A). We said earlier that the comma check updates the CHRGET address so a call to the CHRGOT routine will get the next byte we want. This should be a quote; if not a SYNTAX ERROR is generated (remember that CHRGET skips spaces).

Now to get the data and store it. To get the data we make use of CHRGET. If the zero flag is set, the end of the command has been reached with either a colon or a zero placed by the BASIC input routine. The second quote is checked which also signifies the end of data input. If any of these are found, we branch off to end the routine at line 410. We can now store our data in the table under the BASIC ROM. We do not have to disable the ROM as you cannot store data in ROM so it is automatically stored in the RAM underneath. We increase the Y register which points to the table position. We decrease X which checks for overflow of data. If X reaches zero, the maximum number of characters has been stored. The flow only branches back to get the next byte if X is greater than zero.

To finish off, we store a zero after the last byte of data. This will help when retrieving the data to signify all data has been gathered.

We do another visit to CHRGET to get the next byte as BASIC expects this. This will cause a SYNTAX ERROR if you have input more than ten characters of data though the first ten bytes will have been logged.

The RTS returns us to BASIC for further operations.

DISPLAYING THE KEY DATA

LINES 450–550: These instructions set up the registers used in the display itself. The X register is again used as a counter. Location $5F will have the value of the key number less one and will be used to calculate the pointer for data collection. Locations $22 and $23 hold the ASCII values of the key number. As keys up to and including 9 have only one digit location $22 is loaded with the ASCII for a space character. $23 starts with the ASCII for 1 and will be incremented. Finally, we load up the address of the start of the data table into registers $14 and $15.

LINES 560–610: Call the coding to print key data for keys 1 to 9. After calling, the ASCII value in $23 is increased along with the key number register $5F. Register X is also increased and checked to see if it has reached $0A (ten). If so, we would have to reset the ASCII numbers before printing further data. If X has not reached this value, we branch back to call the print coding for the next key.

LINES 620–720: First we reset locations $22 and $23. Key numbers from 10 to 16 have to be displayed so we have two digit numbers, the first always being one. Therefore, $22 is loaded with $31, the ASCII for one. The other is initialized to zero in ASCII format. We now continue to print out the key data, incrementing $5F, and X each time, until X reaches the value of 17 just after being incremented. This value of X signifies we have finished the display so we exit from the routine and hand control back to BASIC.

The Print Routine to Display the Key Data

This part of the command is entered 16 times in total to print the data to the screen. The value for calculating the pointer, held in $5F, is set before these lines are implemented, as are the ASCII values of the key number. We use the KERNAL routine at $FFD2 to print a character to the screen. The data is printed out in the same format as it was entered. This is done so that it can be changed, just like normal screen editing, if required.

LINES 730–770: The start of every key display line will be the same. These lines will print this from the area of data at the end of the routine (Line 140). We start with a return so it starts on a new line, then a space to give better clarity if the border and screen are different colours, especially if the border and text colours are the same. KEY is printed next, followed by a space for presentation.

LINES 780–850: The key number is printed, followed by a comma and the first set of quotes.

LINES 860–910: The key number, less one, is taken from $5F and increased 16 times with the now familiar four ASL instructions. The result is transferred to the Y register for the data pointer.

LINES 920–1030: Get the key data. First we switch off BASIC to get the data underneath. After returning the BASIC, we print the data as long as it is not the 'end of data' zero. Printing finished, we update the pointer and go back to get the next byte. When the zero is found we exit and print the closing quote. Then it's back to the main key display routine.

OFF – Turn off the keys

COMMAND SYNTAX
OFF
There are no parameters with this command.

If you want to use the function keys within a program simply as keys, you will want to be able to disable the programming they have been given. The command that enables you to do this is OFF. All we do is to alter the addresses in the routine that sets the Keyboard Table Setup Vector back to its normal address. Once changed, we call the routine to change them in the BASIC work area. Do not forget that they can be re-enabled with any KEY command.

ASSEMBLY LISTING

```
 9 *=$8799
10          LDA #$48
20          STA $8056    ! CHANGE LOW ADD IN
                           SETTING ROUTINE
30          LDA #$EB
40          STA $805B    ! CHANGE HIGH ADD IN
                           SETTING ROUTINE
50          JSR $8054    ! CALL SETTING ROUTINE
60          RTS
```

Stand alone programmable function keys

Perhaps this chapter would have been better located between Chapters 6 and 7. It was difficult to decide on its position as it also uses information from both Chapters 2 and 3, but will not work as it stands.

To provide programmable function keys without using the keyword enable routine, the 'get parameter' and 'switch off BASIC' routines have to be copied from Chapter 6. The whole routine may then be relocated and the actions of KEY and OFF performed using SYS commands.

5 Utilities in BASIC

General

This chapter includes many of the utilities in the form of BASIC subroutines and programs. You do not really need any of that which follows if you load-up the UTILITY each time. In time we suspect that the simple routines contained here will not only prove useful, but will also give you plenty of ideas of your own.

A number of the utilities require that you generate an ASCII file of a program on tape or disk. This produces a file in the same format as would be received at a printer or the screen itself. The resulting sequential file contains the program in 'un-tokenized' form. To do this, output must be directed to the desired device with an OPEN and CMD sequence. For a tape this is:

```
OPEN 1,1,1,"PROGRAM":CMD1:LIST[from – to]
PRINT#1:CLOSE 1
```

and for a disk:

```
OPEN 2,8,2,"PROGRAM,S,W":CMD2:LIST[from – to]
PRINT#2:CLOSE 2
```

Most of the utilities given here are in the form of subroutines and have been numbered in the 60000s to allow them to be easily added on to your own programs as and when appropriate. They may be included in whole, or in part, by a suitable merge or append technique. You may wish to combine a number of them together to form useful modules which in the future may save many hours of repetitive work. This you can easily do by using the mini-renumber and merge programs given. Many of the routines can be extended, but they have deliberately been kept as short as possible. Always try to adopt a 'house' format to simplify the creation of future programs. This may only be a simple line numbering sequence where: the working part of your program lies between lines 100 and 9999; the specific subroutines lie between 10000 and 50000; and your library routines are from 50000 on.

The information upon which much of the following is based is contained in Chapter 1 and we refer you to that chapter. The utilities that follow are arranged in alphabetical order.

Keyword – APPEND

Append 1
Function: To append two BASIC programs in memory (nose-to-tail)

In the past, whenever you have loaded a program, it has erased the one currently in memory. This need not be the case. BASIC can start at any address in memory and need not always be the default of 2049 ($0801). The pointer (TXTTAB) to tell the 64 where BASIC begins is held in RAM and can therefore be changed. It is even possible to have two BASIC programs resident in memory concurrently by changing the necessary zero page pointers, though only one could be running at any time. We can manipulate these pointers to allow us to append one program to another.

With a program in memory change TXTTAB to point to its end (VARTAB–2) by:

```
A=PEEK (45) :POKE 43,A-2:POKE 44,PEEK(46)+A<2
```

The program to be appended will now be loaded at the end of the existing one. Resetting the start of BASIC will make the 64 see both programs as one by:

```
POKE 43,1:POKE 44,8 (assumes original start was 2049/$0801)
```

The resulting program may then be edited or saved in the usual way. Many texts say the appended program should have line numbers higher than the original. This is not essential, but some confusion can result if this is not so. Try appending when the second program does not have higher line numbers and see.

The combined program will run correctly until a GOTO or GOSUB references a line which occurs twice. By virtue of the way these commands work, the branch will always be taken to the first occurrence of a line.

Append 2
Function: To append two programs on disk (BASIC or machine code)

Program files on disk store an image of the memory which the program occupied. The first two bytes record the load address and the last byte is a zero to mark the end of file. They can, however, be read and written in a sequential manner. This allows us to append files in much the same way as we did above, but this time performing the operation solely on disk. The following program will append two programs which will load at the address of the first:

LINE ACTION

130 Open up 'Program' files for read and write.
140 Read first program and make a byte by byte copy
TO in the combined file. Skip the terminating zero byte
180 and jump to read the second program.
200 Read the load address and discard it.
210– Copy the remainder through to produce the combined file.

```
100 INPUT"FRONT PROGRAM";F$
110 INPUT"[2SPC]END PROGRAM";E$
120 INPUT"FINAL PROGRAM";R$
130 OPEN 2,8,2,F$+",P,R":OPEN 3,8,3,R$+"
,P,W"
140 GET#2,A$
150 B$=A$:GET#2,A$:IF ST AND 64 GOTO 180
160 IF A$="" THEN A$=CHR$(0)
170 PRINT#3,B$;:GOTO 150
180 CLOSE 2
190 OPEN 2,8,2,E$+",P,R"
200 GET#2,A$:GET#2,A$
210 GET#2,A$:IF ST AND 64 GOTO 240
220 IF A$="" THEN A$=CHR$(0)
230 PRINT#3,A$;:GOTO 210
240 PRINT#3,CHR$(0);
250 CLOSE 3:CLOSE 2
```

Append 3
Function: To reopen an existing closed sequential file on disk and continue writing data from the previous end of file.

This is a standard disk command which is not made clear in the disk manual. Its format is:

OPEN 2,8,2,"TEST,A"

Subroutine keyword – AUTO NUMBER

Function: To automatically generate line numbers as code is entered.

Initiation: RUN 60000

This allows the start line and increment to be set. The line number is printed, followed by any characters typed. When RETURN is pressed the program enters the line, resets the line number variables (as an edit destroys all variables) and reruns itself by forcing two RETURNS into the keyboard buffer. As written, the program will not accept any line not followed by BASIC code (equivalent of delete line).

LINE ACTION

60010 Position cursor to 3rd line down and print line number in black.
60020 Generate a flashing cursor – not normally present on a GET
60040 Watch out for null lines
60060 Print line to – GOTO 60010, reset variables, restart program and move to HOME.
60070 Set NDX for two characters in keyboard buffer. Put two returns in k/b buffer (KEYD). On END KEYD will be emptied and the returns will enter the line and execute line from 60060

```
60000 INPUT "START[4SPC]";LN: INPUT "INC
REMENT";I%
60010 B$="":PRINT CHR$(147);CHR$(17);CHR
$(17);CHR$(17);CHR$(144);LN;CHR$(154);
60020 POKE 204,0:POKE 207,0
60030 GET A$:IF A$="" GOTO 60020
60040 PRINT A$;:IF B$="" AND ASC(A$)=13
GOTO 60010
60050 B$=A$:IF ASC(A$)<>13 GOTO 60020
60060 PRINT "LN=";LN+I%;":I%=";I%;":GOTO
 60010";CHR$(19)
60070 POKE 198,2:POKE 631,13:POKE 632,13
:END
```

The version below is a little more flexible. It will not only delete an existing line if RETURN is pressed after its number, but also allows you to change the printed line number to any value. Subsequent line numbers will increment from the new value until it is again changed. The main difference is the addition of the code to evaluate the current line number ((60060 and 60070). This is done by reading from the start of the fourth screen line until a non-numeric code is encountered and reassigning the line number variable 'LN'.

```
60000 INPUT "START[4SPC]";LN: INPUT "INC
REMENT";I%
60010 PRINT "[CLS][3CD][BLK]";MID$(STR$(
LN),2);"[L BLU]";
60020 POKE 204,0:POKE 207,0
60030 GET A$:IF A$="" GOTO 60020
60040 IF ASC(A$)<>13 THEN PRINT A$;:GOTO
 60020
60050 PRINT:B$="":I=1143
60060 I=I+1:IF PEEK(I)>47 AND PEEK(I)<58
 THEN B$=B$+CHR$(PEEK(I)):GOTO 60060
```

```
60070 LN=VAL(B$):PRINT "LN=";LN+I%;":I%=
";I%;":GOTO 60010[HOM]";
60080 POKE 198,3:POKE 631,13:POKE 632,13
:POKE633,13:END
```

Program keyword – BASES

Function: To convert hex to decimal, binary to decimal and vice versa

This program contains four useful inter-base conversion subroutines. The hex to decimal is most useful if you wish to use hex rather than decimal values in the DATA statements for a machine code BASIC loader. The *Programmer's Reference Guide*, Chapter 3 uses binary patterns for the sprite data in the 'BALLOON' program but pictorial data is also enlightening when setting up user-defined characters and makes for easier editing.

No explanation is given as the program is easy to follow.

```
100 PRINT"1 HEX/DEC":PRINT"2 DEC/HEX":PR
INT"3 BIM/DEC":PRINT"4 DEC/BIN"
110 PRINT:INPUT"SELECT ";N
120 ON N GOSUB 150,240,330,400
130 GOTO100
140 ON N GOSUB 150,240
150 PRINT:INPUT"HEX[4SPC]";A$
160 IF LEN(A$)<4 THEN A$=LEFT$("0000"+A$
,4-LEN(A$))+A$
170 A=ASC(A$)-48
180 B=ASC(MID$(A$,2,1))-48
190 C=ASC(MID$(A$,3,1))-48
200 D=ASC(MID$(A$,4,1))-48
210 E=256*(16*(A+7*(A>9))+B+7*(B>9))+16*
(C+7*(C>9))+D+7*(D>9)
220 PRINT:PRINT"$ ";A$;" = D";E:PRINT
230 RETURN
240 PRINT:INPUT"DEC[4SPC]";G:A=INT(G/256
):B=G-A*256:IF G<0 OR G>65535 GOTO 240
250 C=INT(A/16):D=A-16*C
260 C$=CHR$(48+C):IF C>9 THEN C$=CHR$(C+
55)
270 D$=CHR$(48+D):IF D>9 THEN D$=CHR$(D+
55)
280 E=INT(B/16):F=B-16*E
290 E$=CHR$(48+E):IF E>9 THEN E$=CHR$(E+
55)
```

```
300 F$=CHR$(48+F):IF F>9 THEN F$=CHR$(F+
55)
310 PRINT:A$=C$+D$+E$+F$:PRINT "D";G;" =
 $ ";A$:PRINT
320 RETURN
330 PRINT:INPUT"BIN[4SPC]";A$
340 A=0:A$=RIGHT$("0000000000000000"+A$,
16)
350 FOR I=16 TO 1 STEP -1
360 B$=MID$(A$,I,1):IF B$="1" THEN A=A+2
^(16-I)
370 NEXT I
380 PRINT:PRINT"B ";LEFT$(A$,8);" ";RIGH
T$(A$,8);" = D";A:PRINT
390 RETURN
400 PRINT:INPUT"DEC[4SPC]";A:IF A>65535
OR A<0 GOTO 400
410 B$="":D=A:FOR I=15 TO 0 STEP -1
420 B=INT(A/2^I):IF B=1 THEN B$=B$+"1":G
OTO 440
430 B$=B$+"0"
440 A=A-B*2^I:NEXT I
450 PRINT:PRINT"D";D;"= B ";LEFT$(B$,8);
" ";RIGHT$(B$,8):PRINT
460 RETURN
```

Program keyword – DATALINES

Function: To generate BASIC data statements for machine code programs.

Once again the keyboard buffer is used to generate program lines. This time there are more variables in use than would conveniently fit on a single assign line so they have been put 'out of the way' in the cassette buffer. Only variables in the normal BASIC variable storage area are lost by an edit. The resulting data values are generated to the nearest ten bytes.

LINE	ACTION
60000–	Data input.
60060–	POKE values to TBUFFR.
60090	Recycle from here. Re-read next line number,
60100	step,
60110	start address,
60120	end address for current line,

60130 and end address of program. If finished STOP program.
60140– Print line number, DATA, the values and GOTO 60090.
60210– Increment line number, address, and set up k/b ready for END.

```
60000 INPUT"START ADDRESS";B
60010 INPUT"END ADDRESS[2SPC]";E
60020 F=B:L=F+10
60030 INPUT"START LINE[3SPC]";S
60040 INPUT"LINE INC[5SPC]";T
60050 PRINT"[4CD]"
60060 POKE831,INT(E/256)
60070 POKE832,E-INT(E/256)*256
60080 POKE828,T:GOTO60160
60090 S=PEEK(826)*256+PEEK(827)
60100 T=PEEK(828)
60110 L=PEEK(829)*256+PEEK(830)
60120 E=PEEK(831)*256+PEEK(832)
60130 IFL>=EGOTO60270
60140 F=L+1:L=L+10
60150 PRINT"[CU][14SPC]"
60160 PRINTS;
60170 PRINT"DATA";
60180 FORP=FTOL:PRINTPEEK(P);"[CL],";:NE
XTP
60190 PRINT"[CL][3SPC]"
60200 PRINT"GOTO60090[4CU]";
60210 POKE198,2:POKE631,13:POKE632,13
60220 S=S+T
60230 POKE826,INT(S/256)
60240 POKE827,S-INT(S/256)*256
60250 POKE829,INT(L/256)
60260 POKE830,L-INT(L/256)*256:END
60270 STOP
```

Subroutine keyword – DELETE

Function: To remove unwanted program lines *en masse*

Two delete routines follow. Both use the link address and line number storage at the start of a BASIC line during execution to perform the deletion. The first deletes line numbers as they are encountered whereas the second only deletes one line as the final step in the process. The first line of each routine reads TXTTAB to find out the current start of BASIC.

Delete 1
This routine deletes lines using the all-too-familiar keyboard sequence and as such requires no explanation.

```
60000 TX=PEEK(43)+PEEK(44)*256
60010 INPUT"DELETE FROM";LL:M=256:INPUT"
[7SPC]TO[2SPC]";UL
60020 IF PEEK(TX+2)+PEEK(TX+3)*M<LLTHENT
X=PEEK(TX)+PEEK(TX+1)*M:GOTO 60020
60030 POKE 828,UL-INT(UL/M)*256:POKE 829
,UL/M:GOTO 60050
60040 M=256:TX=PEEK(830)+PEEK(831)*M:UL=
PEEK(828)+PEEK(829)*M
60050 IF PEEK(TX+2)+PEEK(TX+3)*M>UL OR P
EEK(TX)+PEEK(TX+1)*M=0 THEN END
60060 PRINT "[CLS][3CD]";PEEK(TX+2)+PEEK
(TX+3)*M:PRINT"GOTO 60040[HOM]"
60070 POKE830,TX-INT(TX/M)*M:POKE831,TX/
M:POKE198,2:POKE631,13:POKE632,13:END
```

Delete 2

This is, perhaps, a more refined way to carry out the task. It takes fullest advantage of the way programs are stored in RAM and in particular the function of link addresses. The routine scans the line numbers until the start of the block to be removed is found. It records the address of this link address and then continues to scan for the end line number for the delete. Once a line number equal or greater is found, this link address is substituted at the start of the block link. One very large line has thus been created in memory. A simple keyboard program is then used to remove the start line and all others go with it. This is without doubt a lot faster than the first method, but has the disadvantage that you cannot see the lines as they go.

```
60000 TX=PEEK(43)+PEEK(44)*256
60010 INPUT"DELETE FROM";LL:INPUT"[7SPC]
TO[2SPC]";UL
60020 L=PEEK(TX+2)+PEEK(TX+3)*256
60030 IF L<LL THEN TX=PEEK(TX)+PEEK(TX+1
)*256:GOTO 60020
60040 IF L=0 THEN PRINT"LOWER LIMIT";LL;
"NOT FOUND":END
60050 LL=L:D=TX
60060 L=PEEK(TX+2)+PEEK(TX+3)*256
60070 IF L=0 THEN PRINT"UPPER LIMIT";UL;
"NOT FOUND":END
60080 IF L<UL THEN TX=PEEK(TX)+PEEK(TX+1
)*256:GOTO 60060
60090 POKE D,PEEK(TX):POKE D+1,PEEK(TX+1
)
60100 PRINT"[CLS][3CD]";LL;"[HOM]":POKE
198,1:POKE 631,13:END
```

Subroutine keyword – DUMP

Function: To display the current values of all simple numeric, string and function variables

Initiation: Type GOTO 60000

This routine will display the values of all simple variables in use at the time of calling. The variables will be displayed in the order in which they were created by the program. The routine will not handle arrays nor will it work if editing has been carried out prior to its being called (simply because all variable pointers will be reset to the end of program). It also displays the values of the variables it uses – SV, V$, and so on. As these are the last variables to be created they will be the final ones to be displayed. Output may be directed to a printer by a simple:

OPEN 4,4:CMD 4:GOTO 60000

The display may be stopped by holding down any key and will resume on the release of that key. Pressing the STOP key will 'break' into the program and allow you to use the cursor keys to move up and change values. If you resume program execution with a GOTO, then the amended values will be used. A simple CONT would re-enter the dump subroutine at the break and dump any remaining variables.

The routine makes extensive use of the information contained in Chapter 1 on the storage of BASIC variables. Remember the first two bytes are the variable name adjusted for its type. The following is a description of the routine:

LINE	ACTION
60030	Read the current value of VARTAB.
60040	Do the same for ARYTAB.
60050	Default values.
60055	If equal then no simple variables, edit used, or finished. If not equal more variables exist so continue.
60060	Read the seven bytes used for variable.
60070 to 60100	Determine the type from the two name bytes and GOTO the appropriate subroutine, these being real, integer, string or function. The name bytes must be changed back to their unmodified ASCII values by the subtraction of 128, as necessary, and '%' or '$' suffixes printed where required.
60105	Pause if key held down (64=no key at SFDX. Note this is the current key not LSTX as in the *Programmer's Reference Guide*).
60110	Increment 7 bytes to next variable and recycle.
61000	Subroutine to convert 5 floating point binary bytes to decimal.
61500	Subroutine to convert the 2 of the 5 bytes used to a signed integer.

62000 Subroutine to read string length and location then find and build string.
62005 Avoids the single pass through FOR/NEXT if null string.
62020 Surround a string with quotes – required for changing its value on a break.
62500 Subroutine to detect a function and simply acknowledge the fact as its current value will be picked up by one of the other routines.

```
60000 :
60010 :REM DUMP VARIABLES
60020 :
60030 SV=PEEK(45)+PEEK(46)*256 :REM STAR
T OF VARIABLES
60040 SA=PEEK(47)+PEEK(48)*256 :REM STAR
T OF ARRAYS
60050 V$="":VV$="":V=0:VV=0:REM DEFAULTS
60055 IF SA=SV THEN END: :REM NO SIMPLE
VARIABLES OR EDIT USED
60060 FOR V=0 TO 6:V(V)=PEEK(SV+V):NEXT
V:REM READ VARIABLE NAME AND VALUE
60070 IF V(0)<128 AND V(1)<128 THEN GOSU
B 61000:REM REAL
60080 IF V(0)>128 AND V(1)>127 THEN GOSU
B 61500:REM INTEGER
60090 IF V(0)<128 AND V(1)>127 THEN GOSU
B 62000:REM STRING
60100 IF V(0)>128 AND V(1)<128 THEN GOSU
B 62500:REM FUNCTION
60101 IF PEEK(203)<>64 GOTO 60101
60110 SV=SV+7:GOTO 60040:REM INCREMENT C
OUNTER AND DO NEXT
61000 V$=CHR$(V(0))+CHR$(V(1)):REM REAL
NAME
61010 V=(-1)^(V(3)AND128)*2^(V(2)-129)
61020 VV=(1+((V(3)AND127)+(V(4)+(V(5)+V(
6)/256)/256)/256)/128)
61030 V=V*VV:PRINT V$;"=";V:RETURN
61500 V$=CHR$(V(0)-128)+CHR$(V(1)-128)+"
%":REM INTEGER NAME
61510 V=(V(2)AND127)*256+V(3)+(V(2)>127)
*32768
61520 PRINTV$;"=";V:RETURN
62000 V$=CHR$(V(0))+CHR$(V(1)-128)+"$":R
EM STRING NAME
62005 IF V(2)=0 GOTO 62020
```

```
62010 FOR V=1 TO V(2):VV$=VV$+CHR$(PEEK(
V(3)+V(4)*256+V-1)):NEXT V
62020 PRINT V$;"=";CHR$(34);VV$;CHR$(34)
:RETURN
62500 RETURN:REM FUNCTION PICKED UP BY O
THER ROUTINES
```

An alternative approach might be to use the technique in RENUMBER (see below). Namely, print a line which reads: PRINT the variable name and GOTO the point at which program execution should be resumed. If we get the cursor movements right and POKE a RETURN into the keyboard buffer, a dump could be performed. To tidy up, we should really clear the line which says 'PRINT and GOTO' with more cursor movements and spaces, and so on.

An obvious extension would be to include arrays. The logic involved in determining and printing the values of subscripted variables is identical to the above and, with care, the same subroutines could be used. The tricky bit is deciphering the array header to determine the number of dimensions and the size of each dimension. If you do decide to try this, do remember integer array values are stored in only two bytes and string pointers in three bytes, unlike their simple variable counterparts. You must also check that arrays do exist by examining STREND and comparing it with ARYTAB+1. Array headers have also been covered in Chapter 1. Including arrays will greatly increase the size of DUMP and in applications where memory is tight, prove impracticable. It is also difficult, so do not worry if your efforts are not rewarded immediately as a simple error in the logic can cause some very unexpected results.

Program keyword – LISTER

Function: To produce dated, paged and neatly formatted listings

The version given below has been written for an RS232 printer operating at 300 baud, 1 stop bit and no parity (see *Programmer's Reference Guide*, Chapter 6: 'Input/Output Guide'). The printer used also required a carriage return/line feed sequence to be generated at the end of each line. Therefore, the logical file number used has to be greater than 127, in this case #129. When using any RS232 device, it is advisable to OPEN-up the file at the start of the program to allocate the input and output buffers. For other printers, the OPEN and PRINT# statements below will have to be amended to suit.

If your printer does not support the CBM special characters, the program to be listed should first be run through CODER before generating the ASCII file. With a cassette, the OPEN command to read sequential data on line 210 should read OPEN 2,1,0,A$(I).

The listing produced is ideal for permanent record, though as the process takes a little time it is not recommended for intermediate listings. The final listing will have all text inset to column 7 and any wrap-around lines will also be inset. Specifying a line width less than the maximum available has the benefit of allowing space for comments (can save a lot of time in the future). A brief description follows:

LINE ACTION

100 See above.
110– Set parameters.
160– Allocate files to be listed to array A$().
210 See above.
220 This line is included to get any leading returns. The number of these will depend on exactly how the ASCII file was generated. Once a CMD has been issued all returns normally sent to the screen will go to the file. Typically this will be two for the LIST. If zeros appear on your output then you will have to adjust the program or the way you generate the file.
230 Create bottom margin.
250 Build one line into string A$.
260 Same problem as 220 at end of file. Assume a line number of zero is the end.
270 Reset line for text to start in col 7.
290 If length<max then print it.
300– Else split it and print first part. Recycle each time adding 6 leading spaces to continuations.
340 Print blank lines to next top of form before next program.

```
100 OPEN 129,2,0,CHR$(6)
110 PRINT"LISTER UTILITY":PRINT
120 INPUT"DATE[10SPC]";D$
130 INPUT"LINES/PAGE[4SPC]";LP:IF LP=0 T
HEN LP=66
140 INPUT"MAX CHARS/LINE";CP
150 INPUT"NO.OF PROGS[3SPC]";N:DIM A$(N)
160 PRINT:FOR I=1 TO N
170 INPUT"PROGRAM[7SPC]";A$(I)
180 NEXT I
190 I=0
200 I=I+1:LC=0:IF I>N THEN END
210 Z=1:OPEN 2,8,2,A$(I)+",S,R"
220 Z=1:GET#2,A$:GET#2,A$:GOSUB 320
230 IF LC>=LP-8 THEN FOR J=1 TO LP-LC:PR
INT#129,"":NEXT J:GOSUB 320
240 J=0:B$=""
250 J=J+1:GET#2,A$:IF A$<>CHR$(13) THEN
B$=B$+A$:GOTO 250
```

```
260 IF VAL(B$)=0 THEN GOSUB 340:GOTO200
270 L$=STR$(VAL(B$)):B$=MID$(L$+"[6SPC]"
,2,6)+MID$(B$,LEN(L$))
280 L=LEN(B$)
290 IFL<=CPTHEN PRINT#129,B$:LC=LC+1:GOT
O 230
300 L$=LEFT$(B$,CP):PRINT#129,L$:LC=LC+1
310 B$="[7SPC]"+MID$(B$,CP+1):GOTO 280
320 PRINT#129,"PROGRAM ";A$(I);" LISTED
ON ";D$;" LISTING PAGE";Z:LC=2:Z=Z+1
330 PRINT#129,"":RETURN
340 FOR J=1 TO LP-LC:PRINT#129,"":NEXTJ:
CLOSE2:RETURN
```

Subroutine keyword – MERGE

Function: To merge two BASIC programs

In all the following where line numbers are common to both the program in memory and the merging program those of the latter will take precedence.

Merge 1
Where a program is less than 22 screen lines when listed, it may be merged very easily indeed. Simply load the short program and list it. Type NEW and move the cursor to the line below the last line of the list. LOAD the main program and then move up and simply press RETURN on all lines to be included in the final program.

This is the reason for having short keyword routines, to allow the above technique to be used on many of them.

Merge 2
The following subroutine will merge programs of any length. The program (or part of) to be merged must be stored as an ASCII file on disk or tape. The program resident in memory must, of course, include the merge subroutine.

Initiation: RUN 60000

The resulting program will be an amalgamation of the two programs and unlike APPEND the lines will be in the correct numerical sequence. At the completion of the merge an 'OUT OF DATA' or 'SYNTAX ERROR' will be displayed depending on how the ASCII file was generated and which program had the highest line number, but who cares, as the result is exactly what we wanted. The program may then be saved in the normal way (after deleting lines 60000– if they are no longer needed). The version given is for disk and the necessary changes for cassette have been included in the description below, but should be only too familiar by now.

The program uses the keyboard programming technique for the most part. There is one problem and that is that whenever an edit is performed all OPEN files are CLOSEd. So in theory only one line may be read from the file. Any further attempts to obtain input will result in a 'FILE NOT OPEN' error. The solution is simple. BASIC is made to believe a file is open even though an edit has been carried out by POKEing the necessary values into the zero page file registers for current logical file (LA), secondary address (SA) and device number (FA).

LINE	ACTION
60010	For tape OPEN 1,1,0,F$
60020	Get bytes until numeric code. This overcomes the problem in LISTER and perhaps should also be used in that program.
60030	Set file parameters by poking into LA, SA and FA. For tape use 2, 0 and 1 (0=read 1=cassette).
60050	As the first numeric character has been found, mustn't forget to print it. – B$
60060–	As all other programs using keyboard.
60080	Set up k/b buffer on END to enter printed line and GOTO 60030, the cycle repeating until all done.

```
60000 INPUT"PROGRAM ";F$
60010 OPEN2,8,2,F$+",S,R"
60020 GET#2,B$:IF VAL(B$)<1 GOTO 60020
60030 POKE 184,2:POKE 185,2:POKE 186,8:P
OKE152,1
60040 PRINT"[CLS][3CD]";
60050 PRINTB$;:B$=""
60060 GET#2,A$:PRINTA$;:IF A$<>CHR$(13)
GOTO 60060
60070 PRINT"GOTO 60030[HOM]"
60080 POKE 198,2:POKE 631,13:POKE632,13:
END
```

Merge 3 (tape only)

This is the cleverest tape merge we have seen. It was originally worked out by J. Butterfield and B. Templeton for the PET and all we have done is to modify it for the 64.

Again, the program to be merged must be on tape in ASCII format. The statements may be typed in direct mode or, as in this case, be a subroutine. In direct mode the contents of the quotes should be typed after performing the necessary cursor moves and RETURN pressed at the end. Line 60030 is needed only in program mode.

Initiation: RUN 60000

The key to this is the POKE 153,1 (DFLTN) which changes the default input device after each line has been merged from the usual default of 0 (the keyboard) back to the cassette (1).

```
60000 INPUT "PROGRAM ";F$
60010 POKE 19,1:OPEN 1,1,0,F$
60020 PRINT"[CLS][3CD]POKE 153,1:POKE 19
8,1:POKE 631,13:PRINT CHR$(19)"
60030 POKE 198,1:POKE 631,13:PRINT "[HOM
]":END
```

The most common problem with merge is if a program line is in excess of 80 characters when listed (possible if abbreviations have been used). The merge will be unsuccessful as the cursor movements will be incorrect and also BASIC's input buffer will overflow.

Program keyword – OLD

Function: To recover NEWed programs

The command NEW does not actually erase the program in memory, it simply changes the first line's link address to 00 00 (2049 and 2050) and therefore fools BASIC into thinking that there isn't a program present. In addition, all variable pointers are reset to the end of the program, which in this case is the start of BASIC itself (action of CLR). The following uses these facts to recover the program by resetting the necessary pointers.

To use OLD, the start of BASIC (TXTTAB) must be set above the end of the NEWed program and TXTTAB−1 set to zero by:

POKE 43,01:POKE 44, no. of pages:POKE (no. of pages) *256,0:NEW

OLD may then be loaded and run. The erased program will be recovered and you are back in business. As a point of interest OLD is still present higher in memory and will remain so until overwritten by variable data or a larger program.

The program works by hunting from the input value of TXTTAB+4 (ignore first three zeros) for three consecutive zero bytes which mark the end of the erased program. *En route* the first link is changed to point to the second line. Once found, TXTTAB is changed to point to the specified start and VARTAB, to the end address. A CLR then tidies up and the original program is LISTed.

```
60000 INPUT"TXTTAB(2049)";TX:MX=256
60010 POKE 828,TX-INT(TX/MX)*MX:POKE 829
,TX/MX
60020 X=TX+4+J:IF PEEK(X)<>0 THEN J=J+1:
GOTO 60020
60030 POKE TX,X+1-INT((X+1)/MX)*MX:POKE
TX+1,(X+1)/256:TX=X+1
60040 X=PEEK(TX)+PEEK(TX+1)*MX:IF X<>0 T
HEN TX=X:GOTO 60040
60050 TX=TX+1:POKE 830,TX-INT(TX/MX)*MX:
POKE 831,TX/MX
60060 POKE 43,PEEK(828):POKE 44,PEEK(829
):POKE 45,PEEK(830):POKE 46,PEEK(831)
60070 CLR:LIST
```

Subroutine keyword – PLOT

Function: To position the cursor to a specified screen location

There are many ways of positioning the cursor. The most common way is to include the necessary control characters inside quotation marks. This can be expensive on memory if a lot of cursor movement is used. The movement is also relative to the current location and not absolutely fixed to some reference point unless a clear screen or home cursor is first issued. Many micros have TAB(x,y), POS(x,y) or HTAB x and VTAB y functions within their BASICs to position the cursor. The following are just two ways of doing this on the 64 with its unmodified BASIC.

Plot 1

This uses a simple subroutine into which are passed the line and column position. First, two strings are defined – preferably at the start of the program as they remain unchanged throughout the run for speed of access. They are:

1 Y$="[24CD]":X$="[40CR]"

and have been allocated line number 1. To position to any location, the X and Y coordinates are passed to the subroutine which simply homes the cursor then prints the appropriate number of cursor downs and rights.

In the example below, lines 100 to 130 have been included for demonstration purposes.

```
1 Y$="[23CD]":X$="[40CR]"
100 INPUT"COLUMN";X
110 INPUT"[3SPC]ROW";Y
120 GOSUB 1000:PRINTX;",";Y
130 GOTO 100
1000 PRINT"[HOM]";RIGHT$(Y$,Y);RIGHT$(X$
,X);:RETURN
```

The top left of the screen is considered as '0,0'. The semicolon at the end of the print in 60000 is included to hold the cursor at the set location.

The idea of holding frequently used character patterns, control characters, and so on as string variables can reduce memory usage and also makes for easier-to-read code.

Plot 2

This second plot routine uses the same zero page locations as the KERNAL function PLOT see *Programmer's Reference Guide*. These are PNT (209/210), PNTR (211) and TBLX (214). If you look at the memory map in Chapter 5 of the PRG or Appendix K of this book, you'll notice locations from 200 to 245 all relate to the screen in some way or other. We are not going to run through them all, but try experimenting with them and see what happens. If in trouble, turn off the 64.

Let us look at the three locations we are going to use to accomplish PLOT in a little more detail.

PNT: contains the address of the start of the current line in low/high format. With the screen at its default start (1024–2023), this will hold a value 1024+40*row where row is in the range 0–24. Unusual results are produced if this does not correspond to the start of a physical screen line.

PNTR: holds the offset from the address held in PNT. It is the absolute screen column (0–39) when PNT holds the start of line address.

TBLX: holds the current physical screen row.

Using only PNT and PNTR, we can position the cursor to any X,Y location. The next print would occur at the specified point. However, when the cursor returns after the print, it reappears at or below the line it was on before PNT and PNTR were set. This is difficult to put into words and much easier to see. For example, if an input took place on line 23 and the cursor was then moved to line 10, column 10 and a PRINT took place without a semicolon, the cursor would reappear at the start of line 24 and not 11 as might be expected. To avoid this, we simply also set TBLX and all will be well. The routine below has the same effect as the first PLOT routine given. Again, lines 100 to 130 are included for demonstration only.

```
100 INPUT"COLUMN";X
110 INPUT"[3SPC]ROW";Y
120 GOSUB 1000:PRINTX;",";Y
130 GOTO 100
1000 POKE 214,Y:Y=1024+Y*40
1010 POKE 209,Y-INT(Y/256)*256:POKE 210,
INT(Y/256)
1020 POKE 211,X
1030 RETURN
```

Subroutine keyword – PRINT USING

PRINT USING is a very powerful output formatting command available in some BASIC languages. It allows numbers to be right or left aligned to a specified number of decimal places, or to be expressed in exponential format and much more. There are equally as many possibilities for strings. A routine to duplicate all the facilities would be very long, so here we have only considered the problem of formatting numbers.

Very quickly everybody picks up on the idea of:

X=INT(X*10^W+.5)/10^W

to get numbers to a set number of decimal places, where W is the number of places. Unfortunately, due to the way numbers are stored, this is not guaranteed to produce the expected result. By way of a trivial example, try printing .01*649 and 649*.01 and see the difference. The result of any calculation is very much dependent on the order in which it was evaluated. To overcome the problem we have to resort to strings as these are the only type of variable we can fully format.

The following routine will format numbers not in scientific notation and will avoid the XX.X00001 type occurrence by not using any division. The value returned is right aligned to W decimal places and padded with leading spaces to a set width of L. The variable transferred is in X and the string X$ is returned.

```
1 INPUT "X";X :INPUT"W";W:INPUT"L";L: GO
SUB 60000:PRINTX$
2 GOTO 1
60000 X$=STR$(INT(X*10^W+.5)):LE=LEN(X$)
60010 SZ$=".00000000000000":S2$="[31SPC
]"
60020 IF LE<W+2 THEN X$=LEFT$(X$,1)+MID$
(SZ$,1,W+2-LE)+RIGHT$(X$,LE-1)
60030 IF LE>=W+2 THEN X$=LEFT$(X$,LE-W)+
"."+RIGHT$(X$,W)
60040 X$=RIGHT$(S2$+X$,L):RETURN
```

To illustrate its use, the following display

```
  COL1      COL2 COL3     COL4
  99.000   100.00 .999    9.51456
 100.091    98.22 .010   11.00000
```

would be produced by the program lines; where C1–4 represent the values to be PRINTed in cols 1–4 (set elsewhere within your own program).

```
1 C1=99.00001:C2=100.00123:C3=.99888:C4=
9.514569:REM EXAMPLES
10 L=10:W=3:X=C1:GOSUB 60000:PRINT X$;
20 W=2:X=C2:GOSUB 60000:PRINT X$;
30 L=5:W=3:X=C3:GOSUB 60000:PRINT X$;
40 L=14:W=5:X=C4:GOSUB 60000:PRINT X$
```

Where numbers are very large or very small, simply raise them to an appropriate power of ten prior to calling the routine and head the output '*10^N'.

Subroutine keyword – RENUMBER

Function: To renumber a specified section of a program

It is not possible to write a full renumber program in BASIC which does not use ASCII disk files (somebody will no doubt wish to disprove this statement). There are many problems, the biggest of which is in renumbering GOTOS, GOSUBS and THENS line destinations. It is relatively easy, albeit slow, to hunt these out by their token values. The problem arises in correcting destinations which are held in ASCII form. For example, GOTO100 is held as 137 49 48 48 ($89 $31 $30 $30 in hex). If during the renumbering process the destination changes by a magnitude of ten or more (the overall length changes), we have to move all code from the byte following the reference up or down in memory, recalculating link addresses as we go. If all references are entered as five figures as standard, this problem is eliminated, for example, GOTO00100. Entering line numbers in this way is rather tedious and is considered impractical. Machine code renumber programs use the 'crunch tokens' routine and the necessary memory moves are performed as part of this routine when a line is added or removed. See RENUM in Chapter 7.
The program below only renumbers the lines. It will renumber all or only a set block. The new line numbers need not even be in sequence with the rest of the program, though problems will arise if they are referenced. The user will have to manually change all GOTOS, etc. This subroutine is really intended to allow you to put together a number of the shorter routines in this chapter.

```
60000 TX=PEEK(43)+PEEK(44)*256:MX=256
60010 INPUT"RENUMBER FROM";LL:INPUT"[9SP
C]TO[2SPC]";UL
60020 INPUT"[5SPC]NEW LINE";S:INPUT"[9SP
C]STEP";I
60030 IF PEEK(TX+2)+PEEK(TX+3)*MX<LL THE
N TX=PEEK(TX)+PEEK(TX+1)*MX:GOTO 60030
60040 S=S+J*I:IF TX=0 OR PEEK(TX+2)+PEEK
(TX+3)*MX>UL THEN END
60050 POKE TX+2,S-INT(S/MX)*MX:POKE TX+3
,S/MX:TX=PEEK(TX)+PEEK(TX+1)*MX
60060 J=J+1:GOTO 60030
```

Subroutine keyword – SQUASH

Function: To increase the speed of execution of BASIC programs

Many 'crunch' or 'compactor' programs are available, both commercially and in various journals. Their function is to increase the speed of execution of a BASIC program by the removal of redundant code.

There are many reasons why code is slower than it need be. Much of this code is useful at the time of developing the program, but is not required at run-time. Some examples have been given at the end of Chapter 1, but there are many more. Listing the more obvious:

Line numbers: When they are the reference for a GOTO, GOSUB or THEN they are held in ASCII form. The shorter they are (that is, the lower the number), the quicker they are converted to numeric form. Therefore, a renumbering with an increment of 1 is advantageous.
REM: These are ignored at run-time and need only be retained if they are a destination. REMS also use valuable memory.
Spaces: Including spaces in a program makes for easier reading, but is unnecessary and wasteful (this is true only outside quotes).
Variable names: One-character names use less space and are found quicker.
Destinations: See Chapter 1 (page 23)
Screen: See Chapter 2 (page 32)
Print: Semicolons separating print lists are sometimes superfluous. They must be retained after a numeric variable and at the end of a PRINT list if a carriage return is to be inhibited.
Line length: Short lines use an extra five bytes each time (link=2 line=2 end=1) and also take time in working out the next line's details. Lines which are not destinations can be strung together, taking due care of the logic of any IF statements. Lines may be of any length, but are difficult to edit or generate once they exceed 80 characters (even if all the possible abbreviations are used, there is a limit to BASIC's input buffer).

FOR/NEXT loops: A surprising increase in speed is gained by omitting the variable on the NEXT statement. This eliminates the look-up operation for the variable name. Try timing:

FORI=1TO255:FOR J=1TO255:NEXTJ:NEXTI

and FOR......NEXT:NEXT

Operating system: Once spaces have been eliminated, CHRGET itself may be modified to get rid of the test for spaces. (See Chapter 3).

The subroutine below will remove all unnecessary spaces, semicolons and REMs. Renumbering is left up to you. Once again, an ASCII file must first be generated of the program. The program is based on Merge 2 (see page 110) and only the differences from that program are described below:

LINE ACTION

60030 L is set to 1 to account for B$ in first line.
60090 As Merge 2 line 60060, returning to k/b bit once a return found.
60100 Once a REM found, ignore all chars except return.
60110 Flag to indicate in or out of quote mode.
60120 Ignore spaces out of quotes.
60130 Keep spaces in quotes.
60140 Semicolons out of quotes require careful checking and this is carried out at 60210 on.
60150 Semicolons in quotes – keep.
60160 If not an 'M', don't look for REM.
60170 Else see if preceding two chars were 'RE'.
60180 If they were, replace by a ':' and set RE to ignore everything following (see 60100).
60190 Build line to be printed.
60210 Handle the semicolon when out of quotes and eliminate if possible. Do this by getting next byte and if the list continues check for a preceding string or opening quote. Finally, re-enter the main body of the program where appropriate.

Initiation: RUN 60000

```
60000 INPUT"PROGRAM ";F$
60010 OPEN2,8,2,F$+",S,R"
60020 GET#2,B$:IF VAL(B$)<1 GOTO 60020
60030 L=1
60040 POKE 184,2:POKE 185,2:POKE 186,8:P
OKE152,1
60050 PRINT"[CLS][3CD]";
60060 GOSUB 60090
60070 PRINT"GOTO 60040[HOM]"
```

```
60080 POKE 198,2:POKE 631,13:POKE632,13:
END
60090 GET#2,A$:IF A$=CHR$(13) GOTO 60200
60100 IF RE=1 GOTO 60090
60105 IF A$="T" AND Q=0 AND RIGHT$(B$,4)
="PRIN" THEN P=-1
60106 IF A$=":" AND Q=0 THEN P=0
60110 IF A$=CHR$(34) THEN Q=NOT(Q)
60120 IF A$=" " AND Q=0 GOTO 60090
60130 IF A$=" " AND Q=-1 GOTO 60190
60140 IF A$=";" AND Q=0 AND P=-1 GOTO 60
210
60150 IF A$=";" AND Q=-1 GOTO 60190
60160 IF A$<> "M" GOTO 60190
60170 IF MID$(B$,L-1,2)<>"RE" GOTO 60190
60180 B$=LEFT$(B$,L-2)+":":RE=1:GOTO 600
90
60190 B$=B$+A$:L=L+1:GOTO 60090
60200 PRINTB$:RETURN
60210 GET#2,C$:IF C$=":" THEN A$=A$+C$:L
=L+1:P=0:GOTO 60190
60220 IF C$=CHR$(13) THEN B$=B$+A$:GOTO
60200
60230 IFRIGHT$(B$,1)="$"ORRIGHT$(B$,1)=C
HR$(34)ORC$=CHR$(34)THENA$=C$:GOTO60100
60240 A$=A$+C$:L=L+1:GOTO 60190
```

Conclusion

We hope that this chapter has given you food for thought. By way of a project, why not write a routine to recover as much of the data as possible after an edit or NEW has been performed?

6 Routines old and new

Introduction

In Chapter 4 we gave listings in machine code to make use of the function keys. These are actioned by keywords. At the present time, BASIC will not understand these. All the functions of the UTILITY, the remainder of which are in the following two chapters, require some sort of 'driving mechanism'. That is, routines which will not only recognize the keywords, but will action them. Those routines are the PRINT tokens, DISPATCH BASIC CHARS and BASIC EVALUATION. In Chapter 3 these were fully discussed, so we are only supplying in this chapter the coding that is particular to the UTILITY.

To initialize the UTILITY we need to change the addresses in certain locations. These fall into three categories. First, we have to change the vector addresses so that BASIC will go to our token routines; secondly, we need to protect the UTILITY from being overwritten by programs and strings; and lastly we need to retain its operation during a Non-Maskable-Interrupt, that is when RUN/STOP RESTORE is pressed.

There are certain subroutines which will be used by more than one command, so we include them in this chapter. These deal with getting parameters, the switching in and out of the BASIC ROM and memory moving.

That has dealt with the new, and now for the old. A few of the resident ROM routines are useful. Many of them will be covered when describing our new commands. The later part of this chapter describes some more.

Initialization

When you start up the UTILITY with SYS 32768 these instructions will be the first to be actioned. They will set up and protect the UTILITY. At the end of the four subroutines we return control back to you, with a screen message, and the UTILITY in operation.

ASSEMBLY LISTING

```
 9 *=$8000
10             JSR VECTOR ! CHANGE BASIC VECTORS
20             JSR KEYS   ! SET KEYBOARD VECTOR
30             JSR NMI    ! SET NMI AND BRK VECTORS
```

```
 40                JSR MEM      ! SET TOP OF MEMORY
 50                JMP $9200    ! CLR AND MESSAGE
 60 VECTOR         LDA #$09
 70                STA $0304    ! ICRNCH LOW
 80                LDA #$BC
 90                STA $0306    ! IQPLOP LOW
100                LDA #$02
110                STA $0308    ! IGONE LOW
120                LDA #$29
130                STA $030A    ! IEVAL LOW
140                LDA #$82
150                STA $0305    ! ICRNCH HIGH
160                STA $0307    ! IQPLOP HIGH
170                LDA #$83
180                STA $0309    ! IGONE HIGH
190                STA $030B    ! IEVAL LOW
200                RTS
210 MEM            LDA #$FF
220                STA $37      ! MEMSIZ LOW
230                STA $33      ! FRETOP LOW
240                LDA #$7F
250                STA $38      ! MEMSIZ HIGH
260                STA $34      ! FRETOP HIGH
270                RTS
280 NMI            LDA #$7E
290                STA $0316    ! BRK LOW
300                LDA #$61
310                STA $0318    ! NMI LOW
320                LDA #$80
330                STA $0317    ! BRK HIGH
340                STA $0319    ! NMI HIGH
350                RTS
360 KEYS           SEI
370                LDA #$22
380                STA $028F    ! KEYLOG LOW
390                LDA #$87
400                STA $0290    ! KEYLOG HIGH
410                CLI
420                RTS
425 ! NMI ROUTINE
430                PHA
440                TXA
450                PHA
460                TYA
470                PHA
480                LDA #$7F
```

```
490             STA $DD0D  ! CIA INTERRUPT
                             CONTROL REG
500             LDY $DD0D
510             BPL PLUS
520             JMP $FE72  ! RESET RS232
530 PLUS        JSR $F6BC  ! SETS STOP AND RVS FLAG
540             JSR $FFE1  ! CHECK STOP KEY
550             BEQ BREAK
560             JMP $FE72  ! RESET RS232
565 ! BRK ROUTINE
570 BREAK       JSR $FD15  ! KERNAL RESET
580             JSR $FDA3  ! INITIALIZE I/O
                             CIA CHIPS
590             JSR $E518  ! INITIALIZE I/0
600             JSR KEYS
610             JSR NMI
620             JMP ($A002)


807E BREAK           8054 KEYS
8034 MEM             8041 NMI
8073 PLUS            800F VECTOR
```

We feel that this listing up to 430 is fairly self-explanatory, especially with a memory map. The remaining lines are dealt with in the next section.

BRK and NMI routines

These are included in the listing of the previous section, lines 430 to 620. When either of these are initiated, it will be to these lines they will come. The majority of these routines are copies of the equivalent ROM routines, plus a couple of directions to our set up routines to keep the UTILITY in service.

NMI

The NMI is initiated by the use of the RESTORE key (although there are means to initiate it through the cartridge slot). Not only does it tell the microprocessor it has been actioned, but it also sets a flag in the CIA #2. The processor will not action it immediately, but will wait until the present instruction is complete. The processor then saves the program counter and the status register on the stack. It will load the address stored at $FFFA and $FFFB into the program counter. This is normally $FF43. At this address it sets the interrupt flag, so that the other interrupt does not interfere, and then jumps to the vector address that we have changed. Note that the routine has so far not stored the A, X and Y registers.

The NMI in the UTILITY will end up at our routine, which is a series of subroutines. After saving the processor registers on the stack, it clears the NMI flag in the Interrupt Control Register of the CIA#2 chip, which deals with inputs and outputs of the computer. It then loads that location back into Y and if the NMI flag is still clear then it jumps ahead, missing out for the time being an RS232 reset. The following routine checks the STOP and RVS flags at location $91. A call to the KERNAL routine to check for STOP follows. If on exit the accumulator is zero, then the STOP was initiated and we go to the BRK routine. Finally, we jump to the routine to reset the RS232 locations.

BRK

The first subroutine resets the KERNAL set up vector from $0314 to $0333 to their default values from a list held in the KERNAL ROM itself. This will reset two we have changed, the BRK and NMI vectors. The following routine will service the two CIA interface chips, by restoring them to their setup levels.

The routine at $E518 performs the remaining functions of a BRK. It restores the output device to the screen and the input device to the keyboard. The video chip is next for the restoration treatment. The screen and character set are returned to their default positions, and sprite graphics turned off. After this it is the keyboard's turn, with the buffer, delays and set-up vector all returned to default values. The routine finishes off by resetting the input/output flags, clearing the screen, setting the colours and putting the cursor in the home position.

We now put in two calls ourselves so we can reset the NMI, BRK and keyboard vectors to those we require. Finally, there is the indirect jump of A002 which sets the stack pointer to its start, prints 'READY' and gives control back to the user.

Routine vectors and keywords

There is sufficient space, using the existing token system, for 51 further keywords. These will be split up into an area for command keywords and an area for function keywords. In the UTILITY we are supplying 34 commands and 1 function. Between the last command keyword vector and that of the function keyword there is space for a further nine commands (token values 238 to 246 ($EE–$F6)). Seven extra functions could be added within the space available. The vector table is positioned at $8090 to $80F5.

The keyword table is exactly 255 bytes long. Out of that our keywords use up 155 plus a zero byte to mark the end of the table. The amount of space available to you if wish to extend it is 58 bytes for command keywords and 41 bytes for functions. Remember that the last letter of each word has $80 (128) added to it. In our table, the space between our last command, TROFF, and the only function, DEEK, has been filled with bytes $5A and $EA to make up the nine unused token values.

MEMORY DUMP

```
.:8090 98 87 4C 86 B2 83 9F 84'........
.:8098 EB 84 36 85 BE 85 14 84'........
.:80A0 51 83 A6 83 AE 8F B4 8F'........
.:80A8 80 83 AC 83 51 8E C4 89'........
.:80B0 43 8F A6 87 92 8B 2D 84'........
.:80B8 D1 8F 3A A9 D1 A8 30 86'........
.:80C0 B6 91 39 8D 10 86 B5 92'........
.:80C8 8C 91 80 91 4D 90 FB 85'........
.:80D0 6E 88 60 8D FF FF FF FF'........
.:80D8 00 FF FF FF F6 FF B6 F7'........
.:80E0 00 60 00 00 D6 83 D6 83'........
.:80E8 00 00 00 68 00 00 00 40'........
.:80F0 00 00 00 40 00 40 4F 46'......OF
.:80F8 C6 4B 45 D9 44 4F 4B C5'fKEyDOKe
.:8100 54 45 CE 54 57 CF 48 45'TEnTWoHE
.:8108 D8 42 49 CE 4F 4C C4 43'xBInOLdC
.:8110 4F 4C 4F 55 D2 57 52 49'OLOUrWRI
.:8118 54 C5 43 47 4F 54 CF 43'TeCGOToC
.:8120 47 4F 53 55 C2 50 4C 4F'GOSUbPLO
.:8128 D4 45 4E 54 45 D2 44 55'tENTErDU
.:8130 4D D0 52 45 4E 55 CD 44'MpRENUmD
.:8138 45 4C 45 54 C5 4D 45 52'ELETeMER
.:8140 47 C5 43 4F 44 45 D2 41'GeCODErA
.:8148 55 54 CF 50 52 4F C3 44'UToPROcD
.:8150 50 52 4F C3 45 50 52 4F'PROcEPRO
.:8158 C3 50 4F D0 51 55 49 D4'cPOpQUIt
.:8160 54 52 41 43 C5 52 45 53'TRACeRES
.:8168 45 D4 43 48 41 49 CE 4C'EtCHAInL
.:8170 4F 4D 45 CD 48 49 4D 45'OMEmHIME
.:8178 CD 49 4E 4B 45 59 A4 4D'mINKEY M
.:8180 45 CD 41 50 50 45 4E C4'EmAPPENd
.:8188 54 52 4F 46 C6 5A 5A 5A'TROFfZZZ
.:8190 5A 5A EA 5A 5A 5A 5A 5A'ZZ.ZZZZZ
.:8198 5A EA 5A 5A 5A 5A 5A 5A'Z.ZZZZZZ
.:81A0 5A EA 5A 5A 5A 5A EA 5A'Z.ZZZZ.Z
.:81A8 5A 5A 5A EA 5A 5A 5A 5A'ZZZ.ZZZZ
.:81B0 EA 5A 5A 5A 5A 5A 5A EA'.ZZZZZZ.
.:81B8 5A 5A 5A 5A 5A EA 5A 5A'ZZZZZ.ZZ
.:81C0 5A 5A 5A 5A 5A 5A EA 44'ZZZZZZ.D
.:81C8 45 45 CB 00 FF FF FF FF'EEK.....
```

```
.:81D0 FF FF FF FF FF FF FD FF´........
.:81D8 FF FF FF FF FF 7F FF FF´........
.:81E0 00 00 00 00 00 00 00 00´........
.:81E8 00 00 00 08 01 00 00 00´........
.:81F0 00 00 00 00 00 20 8A AD´..... ..
```

This has been produced in upper case mode and as such the end shifted letter of each command is printed in lower case. If putting it into your computer in a way other than the dump, remember that they are shifted. The last letter in location $817E is a shifted $, giving the keyword INKEY$.

Getting parameters and controlling BASIC

ASSEMBLY LISTING

```
  9 *=$81F5
 10          JSR $AD8A   ! GET INPUT
 20          JMP $B7F7   ! CHECK AND TRANSFER
 30          LDA $01     ! 6510 I/O PORT
 40          AND #$FE    ! TURN OFF BIT 0
 50          STA $01     ! BASIC OFF
 60          RTS
 70          LDA $01
 80          ORA #$01    ! SET BIT 0
 90          STA $01     ! BASIC ON
100          RTS
```

Parameters

Lines 10 and 20 hold the only two instructions that we need to incorporate, but they do a lot of work in getting our numeric parameters. Let us look at the instructions one at a time.

JSR $AD8A

The first action of this is to call the evaluate expression routine at $AD9E.

This is a complex routine which deals not only with numeric data, but also with strings. After setting the CHRGET pointer back one place, it proceeds to start picking up data after the command keyword. It will then go through checking to see whether a mathematical operator or a function keyword (such as PEEK), a variable or simply a number has been obtained. From the information obtained it will (after calculating if necessary) store the result or findings in the FAC#1. For numbers up to $FFFF, the relevant numbers will be in locations $64 and $65 of this accumulator.

We now return to our original subroutine at $AD8A, where we check to see if the data received was numeric or not. The evaluate expression will set a flag in the zero page location $0D. The value of $FF indicates string data, whilst zero designates numeric data. If this subroutine finds the former, a 'TYPE MISMATCH' error is generated and the command, and program, is terminated.

JMP $B7F7

We have our numeric parameter. This routine will do two checks and then transfer our data. The checks are to make sure that neither a negative number nor one over 65535 ($FFFF) was given. In either case, failure will result in the 'ILLEGAL QUANTITY' error. The data is now transferred from $64 and $65 to locations $14 and $15. The reason for this is that the FAC#1 is used for many applications. The RTS at the end of this routine will return us to the place that called our complete GET PARAMETERS routine, that will most likely be a command routine.

The BASIC switch

As we said, when dealing with the function keys, the area of RAM under the BASIC ROM is a useful place for hiding data, or indeed routines which do not use the BASIC interpreter. To use this area, BASIC must be 'removed'. We have no trouble writing to the RAM as the computer, through its decoding logic, will select it when the processor sends a write signal. When reading, the ROM has priority unless we tell the electronics that it is not there. The main difference between the 6510 processor in the 64 and the normal 6502 is that the former has input/output ports. The user can control these using locations $0000 and $0001. The first deals with the direction of the data, that is, whether the ports, of which there are six, are going to be input or output. The second location deals with the data itself, one bit for each port, either a one or a zero which gives a switching mode. Three of the ports are connected to the cassette port. The other three control three ROMS: BASIC, KERNAL and the Character ROM. A zero will switch all of these off. The one we are concerned with, BASIC, uses bit 0 of the data register and so by changing this bit, making sure not to disturb the others, we can remove or replace as required.

Lines 30 to 60 perform the switching-out of BASIC. We load the register and set bit 0 to zero. The AND instruction will do this without changing any other bit. After placing the result back, the ROM is no longer present as far as the computer is concerned.

Lines 70 to 100 reverse the process by using the ORA code which will only affect the bits according to the data with the instruction.

To switch off BASIC – JSR $81FB
To switch in BASIC – JSR $8202

Dealing with the keywords

In Chapter 3 the routines that BASIC uses to deal with keywords and tokens were fully described. Below are the listings to use with the UTILITY, which require no further explanation.

ASSEMBLY LISTING – CRUNCH TOKENS

```
9 *=$8209
10                LDX $7A
20                LDY #$04
30                STY $0F
40  ANOTHER       LDA $0200,X
50                BPL SPACE
60                CMP #$FF
70                BEQ STORE
80                INX
90                BNE ANOTHER
100 SPACE         CMP #$20
110               BNE STORE
120               STA $08
130               CMP #$22
140               BEQ QUOTE
150               BIT $0F
160               BVS STORE
170               CMP #$3F
180               BNE NUMBER
190               LDA #$99
200               BNE STORE
210 NUMBER        CMP #$30
220               BCC CONT
230               CMP #$3C
240               BCC STORE
250 CONT          STY $71
260               LDY #$00
270               STY $0B
280               DEY
290               STX $7A
300               DEX
310 NEXTLETTER    INY
320               INX
330 CONT1         LDA $0200,X
340               SEC
350               SBC $A09E,Y
360               BEQ NEXTLETTER
370               CMP #$80
380               BNE NEXTWORD
390 STOREA        ORA $0B
400 FOUND         LDY $71
410 STORE         INX
420               INY
430               STA $01FB,Y
440               LDA $01FB,Y
450               BEQ EXIT
460               SEC
470               SBC #$3A
480               BEQ COLON
490               CMP #$49
500               BNE DATA
510 COLON         STA $0F
520 DATA          SEC
530               SBC #$55
540               BNE ANOTHER
550               STA $08
560               LDA $0200,X
570 LINE          BEQ STORE
580               CMP $08
590               BEQ STORE
600 QUOTE         INY
610               STA $01FB,Y
620               INX
630               BNE LINE
640 NEXTWORD      LDX $7A
650               INC $0B
660 FIND          INY
670               LDA $A09D,Y
680               BPL FIND
690               LDA $A09E,Y
700               BNE CONT1
710               LDY #$FF
720               DEX
730 NEXT          INY
740               INX
750 NEXTB         LDA $0200,X
760               SEC
770               SBC $80F6,Y
780               BEQ NEXT
790               CMP #$80
800               BNE NEXTNEW
810               BEQ STOREA
```

```
870              LDA $80F6,Y
880              BNE NEXTB
890              LDA $0200,X
900              BPL FOUND
910 EXIT         JMP $A609
```

```
820 NEXTNEW      LDX $7A
830              INC $0B
840 NEXTA        INY
850              LDA $80F5,Y
860              BPL NEXTA
```

```
820F ANOTHER        8269 COLON
8239 CONT           8245 CONT1
826B DATA           82B9 EXIT
8286 FIND           8254 FOUND
8275 LINE           8294 NEXT
82A9 NEXTA          8296 NEXTB
8243 NEXTLETTER     82A5 NEXTNEW
8282 NEXTWORD       8231 NUMBER
827B QUOTE          821B SPACE
8256 STORE          8252 STOREA
```

ASSEMBLY LISTING – PRINT TOKENS

```
  9 *=$82BC
 10              BPL ROM1
 20              CMP #$FF
 30              BEQ ROM1
 40              BIT $0F
 50              BMI ROM1
 60              CMP #$CC
 70              BCC CBMTOKEN
 80              SEC
 90              SBC #$CB
100              TAX
110              LDA #$F6
120              STA $22
130              LDA #$80
140              STA $23
150              BNE START
160 CBMTOKEN     SEC
170              SBC #$7F
180              TAX
190              LDA #$9E
200              STA $22
210              LDA #$A0
220              STA $23
230 START        STY $49
240              LDY #$FF
250 NEXTWORD     DEX
```

```
260                 BEQ WORDFOUND
270 NEXTCHAR        INY
280                 LDA ($22),Y
290                 BPL NEXTCHAR
300                 BMI NEXTWORD
310 WORDFOUND       INY
320                 LDA ($22),Y
330                 BMI ROM2
340                 JSR $AB47
350                 BNE WORDFOUND
360 ROM1            JMP $A6F3
370 ROM2            JMP $A6EF
```

```
82D8 CBMTOKEN           82EB NEXTCHAR
82E8 NEXTWORD           82FC ROM1
82FF ROM2               82E4 START
82F2 WORDFOUND
```

ASSEMBLY LISTING – DISPATCH AND EVALUATION

```
  9 *=$8302
 10                 JSR $0073
 20                 CMP #$CC
 30                 BCC ROM3
 40                 CMP #$EE
 50                 BCS ROM3
 60                 JSR DISPATCH
 70                 JMP $A7EA
 80 DISPATCH        SEC
 90                 SBC #$CC
100                 ASL A
110                 TAY
120                 LDA 8091,Y
130                 PHA
140                 LDA $8090,Y
150                 PHA
160                 JMP $0073
170 ROM3            JSR $0079
180                 JMP $A7E7
190                 LDA #$00
200                 STA $0D
210                 JSR $0073
220                 CMP #$F7
230                 BCC ROM4
240                 CMP #$F8
```

```
250                 BCS ROM4
260                 JSR DISPATCH1
270                 RTS
280 DISPATCH1       SEC
290                 SBC #$F6
300                 ASL A
310                 TAY
320                 LDA $80E5,Y
330                 PHA
340                 LDA $80E4,Y
350                 PHA
360                 JMP $0073
370 ROM4            JSR $0079
380                 JMP $AE8D


8313 DISPATCH          833C DISPATCH1
8323 ROM3              834C ROM4
```

The start up message

This is the final subroutine called during the initialization of the UTILITY. It performs a CLR, to set all the variable addresses, changes the screen and text colours, and finally puts a message on the screen indicating that the UTILITY is in operation.

ASSEMBLY LISTING

```
  9 *=$9200
 10                JSR $A663      ! CLR
 20                LDA #$93       ! CLEAR SCREEN
 30                JSR $FFD2
 40                LDA #$00       ! SET COLOURS TO BLACK
 50                STA $D020      ! BORDER
 60                STA $D021      ! BACKGROUND
 70                LDA #$05
 80                STA 0286       ! GREEN TEXT
 90                LDX #$0A
100                LDY #$09
110                JSR STARS
120                LDX #$0C
130                LDY #$09
140                CLC
```

```
                 JSR $FFF0     ! SET CURSOR
                 LDX #$15      ! CHARS TO PRINT
     CONT        LDA DATA,X
                 JSR $FFD2     ! PRINT
                 DEX
                 BPL CONT      ! NOT FINISHED
                 LDX #$0E
                 LDY #$09
                 JSR STARS
                 STA $05C1     ! FILL IN MISSING CHARS
                 STA $0611
                 STA $05D6
                 STA $0626
                 LDA #$05      ! COLOUR MAP VALUE
                 STA $D9C1
                 STA $DA11
                 STA $D9D6
                 STA $DA26
                 LDA #$0D      ! PRINT RETURN
                 JSR $FFD2
                 JMP $A474     ! READY FOR BASIC
     STARS       CLC
                 JSR $FFF0     ! SET CURSOR
                 LDA #$2A      ! ASCII FOR *
                 LDX #$16      ! NUMBER TO PRINT
     NEXT        JSR $FFD2
                 DEX
                 BNE NEXT
                 RTS
     DATA        TXT "* YTILITU CISAB  NAP *"


9226 CONT              9267 DATA
9260 NEXT              9258 STARS
```

Memory moving

RENUMber and CODER, described in Chapter 7, both require some manipulation of memory in the form of either gaining space or removing unnecessary bytes. This section deals with the two subroutines, CLOSE and OPEN, which perform these operations. CLOSE is self-contained whilst OPEN uses a ROM routine for the actual moving of memory. In the BASIC interpreter there are routines to both open and close up a BASIC program, used when you insert or delete lines, but we can only really use the opening routine. It is a subroutine on its own whereas the closing-up is integral with the inputting of a BASIC line. We have written coding that is virtually identical to the one in ROM as it is efficient enough.

Having moved the program about, all the link addresses, from the line the move started, will now be wrong by the amount of the move. There is a subroutine in the interpreter which changes the link addresses but we have not used it. The reason for this is one of speed as during the course of using CODER or RENUM, these subroutines may be called several times and would prove to be very slow.

The ROM routine for rechaining the lines goes through the whole program, byte by byte, to calculate the link addresses and store them. It has been done this way as it is a multi-purpose routine, catering for the lengthening and shortening of code. What we have done is to write separate routines for each direction of movement and place them immediately after the moving instructions. These will only rechain from the program line in which the alteration occurred. In addition, we only need to look at the link addressess as we know by how much they have changed so we can subtract or add as required.

To set the scene, as they say, here are the locations that need to be set before calling these subroutines:

$FB and $FC–	The address of the start of the current BASIC line.
$49–	The number of the current position on that line. This will be where the replacement code will start.
$FD and $FE–	The address of the next BASIC line, that is, the link address of the line in $FB and $FC.
$3E–	The number of bytes in the original code to changed.
register–	The number of bytes in the replacement code.

ASSEMBLY LISTING

```
  9 *=$888B
 10          STX $C2
 20          LDA $3E      ! FIND HOW MANY
                            BYTES TO REMOVE
 30          SEC
 40          SBC $C2
 50          STA $BB
 60          CLC
 70          LDA $FB
 80          ADC $49      ! FIND START OF
                            BLOCK TO MOVE
 90          STA $5F
100          LDA $FC
110          ADC #$00
120          STA $60
130          LDA $5F
```

```
140             ADC $BB
150             STA $5A      ! START + AMOUNT OF
                               REDUCTION
160             LDA $60
170             ADC #$00
180             STA $5B
190             LDA $2D      ! END OF PROG
200             SEC
210             SBC $5A      ! CALCULATE TOTAL
                               AMOUNT TO MOVE
220             STA $58
230             TAY          ! NO OF BYTES OF
                               INCOMPLETE PAGE
240             LDA $2E
250             SBC $5B
260             TAX          ! NO OF PAGES TO
                                   MOVE
270             INX          ! FOR EASIER CHECKING
280             TYA
290             BEQ PAGE     ! NO SEPARATE BYTES
300             LDA $5A      ! MOVE SEPARATE
                               BYTES FIRST
310             CLC
320             ADC $58
330             STA $5A
340             BCC NOINC
350             INC $5B
360             CLC
370 NOINC       LDA $5F
380             ADC $58
390             STA $5F
400             BCC NOINCA
410             INC $60
420 NOINCA      TYA
430             EOR #$FF
440             TAY
450             INY
460             DEC $5B
470             DEC $60
480 PAGE        LDA ($5A),Y
490             STA ($5F),Y
500             INY
510             BNE PAGE
520             INC $5B
530             INC $60
540             DEX          ! POINTER - COMPLETION
```

```
550            BNE PAGE
560            SEC
570            LDA $2D        ! SET END OF PROG
580            SBC $BB
590            STA $2D
600            BCS RECHAIN
610            DEC $2E
620            SEC
630 RECHAIN    LDY #$00
640            LDA $FD        ! GET LINK
650            SBC $BB        ! CALC NEW ADDRESS
660            STA $FD
670            STA ($FB),Y    ! STORE IN LINE
680            STA $57
690            LDA $FE
700            SBC #$00
710            INY
720            STA $FE
730            STA $58
740            STA ($FB),Y
750 NEXT1      DEY
760            LDA ($57),Y    ! GET LINKS
770            STA $B9        ! STORE THEM
780            INY
790            LDA ($57),Y
800            STA $BA
810            BEQ EXIT       ! COMPLETED RECHAINING
820            DEY
830            SEC
840            LDA $B9
850            SBC $BB        ! CALC NEW LINK ADDS
860            TAX            ! TEMP STORE
870            STA ($57),Y
880            LDA $BA
890            SBC #$00
900            INY
910            STA ($57),Y
920            STA $58
930            TXA
940            STA $57
950            JMP NEXT1      ! GET NEXT LINE
960 EXIT       RTS
970            TXA
980            SEC
990            SBC $3E        ! CALCULATE NO OF
                                SPACES REQUIRED
```

```
1000           STA $BB
1010           CLC
1020           LDA $49
1030           ADC $BB
1040           BCS ERROR    ! >255 CHARS IN LINE
1050           CMP $FE
1060           BCC CONT     ! ONLY 254 ALLOWED
                              -NEED END MARKER
1070 ERROR     LDX #$17
1080           JMP $A437    ! ERROR STRING TOO LONG
1090 CONT      LDA $2D
1100           ADC $BB      ! ENOUGH MEMORY?
1110           TAX
1120           LDA $2E
1130           ADC #$00
1140           CMP $38
1150           BNE CONT2    ! ENOUGH MEMORY
1160           CPX $37
1170           BCC CONT2
1180           JMP $A435    ! ERROR OUT OF MEMORY
1190 CONT2     CLC
1200           LDA $2D      ! SET ADDS FOR MOVE
1210           STA $5A
1220           ADC $BB
1230           STA $58
1240           LDA $2E
1250           STA $5B
1260           ADC #$00
1270           STA $59
1280           LDA $FB
1290           ADC $49
1300           STA $5F
1310           LDA $FC
1320           ADC #$00
1330           STA $60
1340           JSR $A3BF    ! ROM ROUTINE TO
                              OPEN UP MEMORY
1350           CLC
1360           LDY #$00
1370           LDA $2D      ! SET NEW END OF PROG
1380           ADC $BB
1390           STA $2D
1400           BCC CONT3
1410           INC $2E
1420           CLC
1430 CONT3     LDA $FD
```

```
1440           ADC $BB
1450           STA $FD
1460           STA $57
1470           STA ($FB),Y
1480           LDA $FE
1490           ADC #$00
1500           INY
1510           STA $FE
1520           STA $58
1530           STA ($FB),Y
1540 NEXT3     DEY
1550           LDA ($57),Y
1560           STA $B9
1570           INY
1580           LDA ($57),Y
1590           STA $BA
1600           BEQ EXIT2
1610           DEY
1620           CLC
1630           LDA $B9
1640           ADC $BB
1650           TAX
1660           STA ($57),Y
1670           LDA $BA
1680           ADC #$00
1690           INY
1700           STA ($57),Y
1710           STA $58
1720           TXA
1730           STA $57
1740           JMP NEXT3
1750 EXIT2     RTS
```

```
CONT      8949     CONT2     895D
CONT3     898B     ERROR     8944
EXIT      892F     EXIT2     89C4
NEXT1     890E     NEXT3     89A0
NOINC     88CA     NOINCA    88D4
PAGE      88DD     RECHAIN   88F7
```

CLOSE ROUTINE

LINES 10–270: Before we can move a block of memory, we have to determine three values: the start address of the block to move, the new start address and the amount of code to move. The first thing we work out is the number of redundant bytes. This is done, obviously, by subtracting from the original amount of data to be changed the number of bytes of the replacement code. The resultant value is stored in location $BB. We shall need this number later for rechaining the lines. The new start of the block will be obtained by adding the line pointer, $49, to the address of the current BASIC line. To this value is added the contents of $BB which will give us the location of the first byte in the block to be moved.

To get the amount of data to be moved, the result of the last calculation is taken away from the end of program address, held in $2D and $2E. The answer will be held in the processor registers, the high byte in the X and the low in the Y. A page of memory is 256 bytes so the X register is therefore the number of pages to be moved, increased by one for easier checking on completion. We move a complete page and then decrease X. X will be zero when all done, checking immediately after decreasing. To summarize, we have found the amount to move, its current start and its destination.

LINES 280–470: This is the hardest part of the routine to follow, and we hope that we succeed in explaining it clearly.

We transfer the Y register to the accumulator. To recap briefly, this will be the number of bytes, other than complete pages, of memory to move. If the value now in the accumulator is zero, only complete pages require moving, so we skip this section completely. In closing up memory we start from the low addresses, move them, and work to the higher end addresses. We do this by setting the address of the page and moving it up, using the Y register as a pointer. If we have an odd number of bytes to start with, this causes a slight problem. For example, if we have $10 bytes and the Y is set thus we would move 246 bytes by increasing Y. To compensate for this, what we do is to produce the 2's complement of the value. This is done in lines 430 to 450. The EOR #$FF will change all the bits set to one to zero and vice versa. One is then added. So instead of $10, we should now have $F0. This means that if we now increase Y from $F0 until it becomes zero it will have been incremented $10 times.

For the same reasons we have to alter the address of the start of the block and its new start address. We add to these the original number of odd bytes, held in $58. Finally, we decrease the high byte of the address by one. The next effect of these changes is a stalemate as the locations along with the Y pointer value are equivalent to the original values but now allow us to increase Y the required amount.

LINES 480–550: Having set all the values we move the data, byte by byte, until both X and Y registers are zero. We simply load a byte from its position and store its new lower location.

LINES 560–620: The end of the BASIC program will now be shorter by the value of location $BB. The original end address is adjusted and reset.

LINES 630–960: All that remains is to change the values of the link addresses from the current BASIC line onwards. First, we change the links in the current line and as these are also held in $FD and $FE, used by the calling routine, we change these also.

We proceed through the lines gathering the addresses, subtracting the value in $BB, and then we restore them. The end of the program is indicated when the MSB of a link address is zero. Finally, we return to the calling program, such as CODER.

OPEN ROUTINE

LINES 970–1080: We calculate the space required by subtracting the value in $3E, the length of the old code, from the value in the X register, the length of the new code, and store the result in $BB.

As a BASIC line may not exceed 255 bytes (to allow for a zero at the end making a maximum of 256), we check this by adding the line marker to the $BB value. A set carry flag will mean the maximum has been exceeded. We then check that there will be room for the end of line zero. Failure of either of these will generate the syntax error 'STRING TOO LONG'.

LINES 1090–1180: As we are creating space we must check that there is sufficient room available in the BASIC program area. These lines do just that by checking that we will not exceed the values in $37 and $38, which indicate its limit. If we do go over, we call a BASIC routine to generate the 'OUT OF MEMORY' error message.

LINES 1190–1340: Next on the agenda is to set the registers for the interpreter's OPEN-up memory routine at $A3BF. On leaving this routine:
$5A and $5B – This will hold the address of the end of the block to move. It will be the same as the end of program address before the move.
$58 and $59 – These registers will hold the address of the end of the new block. It will also be the end of the BASIC program after the move. It is arrived at by adding the amount of move to the address in $5A and $5B.
$5F and $60 – The start of the block to move. These hold the location of the first byte of the code to be changed. It is calculated by adding the line marker to the address of the current BASIC line to be processed.

LINES 1360–1420: Now that the data has been moved, we reset the end of program address to its new value.

LINES 1430–1750: A replica of the rechaining in lines 630 to 960, except that here we increase the addresses instead of reducing them.

This concludes the new routines that we planned to introduce in this chapter. The remainder are descriptions of some of the ROM routines we use (and hope that you will come to use).

RECHAINING THE LINES

During our memory move routine, we did not use the ROM routine to rechain the link addresses because for our purposes it was inefficient due to the number of calls required. However, we do use the subroutine, in DELETE for instance, where only one call is required. It serves another purpose in that from the addresses it exits with, one can calculate and set the end of program/start of variable registers.

ROM LISTING

```
A533 A5 2B        LDA $2B
A535 A4 2C        LDY $2C
A537 85 22        STA $22
A539 84 23        STY $23
A53B 18           CLC
A53C A0 01        LDY #$01
A53E B1 22        LDA ($22),Y
A540 F0 1D        BEQ $A55F
A542 A0 04        LDY #$04
A544 C8           INY
A545 B1 22        LDA ($22),Y
A547 D0 FB        BNE $A544
A549 C8           INY
A54A 98           TYA
A54B 65 22        ADC $22
A54D AA           TAX
A54E A0 00        LDY #$00
A550 91 22        STA ($22),Y
A552 A5 23        LDA $23
A554 69 00        ADC #$00
A556 C8           INY
A557 91 22        STA ($22),Y
A559 86 22        STX $22
A55B 85 23        STA $23
A55D 90 DD        BCC $A53C
A55F 60           RTS
```

The routine commences by getting the program start address and placing it in registers for its own use. The carry flag is cleared for addition. The first byte of a line that it picks up is the high byte of the link address and it tests for the end of the program (a zero). The Y register is loaded again so as to skip the addresses and line number. It now proceeds through the line, searching for the end of line zero marker. When this is discovered, the Y register will contain one less than the number of bytes in the complete line. This is immediately rectified by incrementing Y by one. This value is added to the line start address and placed as the link address of the line. As this is also the address of the next line, it is loaded into the locations used by the routine. The flow now branches back, (the carry flag will be clear), to process the next line. Every BASIC line will be processed until the end of the program.

On exiting, the program locations $22 and $23 will hold the address of the two end zero bytes. If this address is increased by two then the end of program address can be derived, and hence the start of variables, as they are one and the same thing VARTAB.

Opening up memory

In our memory move routine we made use of a ROM routine when we required more space in a BASIC program. It will move a block up in memory even if its new start is within the original block. Six locations have to be set before entering the routine, which are, in low/high byte order:

$5A and $5B– End address of present block
$5F and $60– Start address of present block
$58 and $59– End address of the new block

ROM LISTING

```
A3BF 38          SEC
A3C0 A5 5A       LDA $5A
A3C2 E5 5F       SBC $5F
A3C4 85 22       STA $22
A3C6 A8          TAY
A3C7 A5 5B       LDA $5B
A3C9 E5 60       SBC $60
A3CB AA          TAX
A3CC E8          INX
A3CD 98          TYA
A3CE F0 23       BEQ $A3F3
A3D0 A5 5A       LDA $5A
A3D2 38          SEC
A3D3 E5 22       SBC $22
A3D5 85 5A       STA $5A
A3D7 B0 03       BCS $A3DC
A3D9 C6 5B       DEC $5B
A3DB 38          SEC
```

```
A3DC A5 58        LDA $58
A3DE E5 22        SBC $22
A3E0 85 58        STA $58
A3E2 B0 08        BCS $A3EC
A3E4 C6 59        DEC $59
A3E6 90 04        BCC $A3EC
A3E8 B1 5A        LDA ($5A),Y
A3EA 91 58        STA ($58),Y
A3EC 88           DEY
A3ED D0 F9        BNE $A3E8
A3EF B1 5A        LDA ($5A),Y
A3F1 91 58        STA ($58),Y
A3F3 C6 5B        DEC $5B
A3F5 C6 59        DEC $59
A3F7 CA           DEX
A3F8 D0 F2        BNE $A3EC
A3FA 60           RTS
```

The immediate action is to calculate the number of bytes to move. The number of low bytes is placed in the Y register and location $22. The number of pages to move, the difference of the high bytes, is placed in the X register and immediately increased by one. This will be the counter where the zero state is checked to determine completion. As it is decreased before being checked, increasing by one will ensure that all pages will be done. If X was zero and was not incremented, then you would end up going around the circuit 256 times before a zero was discovered in the X register.

The low byte result is checked again; if there is no value, then a large chunk of instructions can be skipped. The bytes between addresses $A3D0 and $A3E7 deal with cases where there is an element of an incomplete page of data to move. These lines reduce the two end addresses by the number of low bytes to move. This will not effect the move as the data is loaded and stored with respect to Y and this has the number that was the reduction. The incomplete page is moved first.

Except when Y is zero, all the bytes are transferred within addresses $A3E8 and $A3EE. The Y register will start at a high value and be decremented to zero. When that is reached, the next bytes are moved separately, before the high addresses are decreased. After this has been achieved, the X counter is reduced and checked, and if it is not zero, it's back to move the next page of data.

From this it can be seen that the transfer is done by taking the high addresses and moving them first. This means that the program will not overwrite itself.

Find a line

This routine finds the start address of a BASIC line, given the line number. We shall use it in our RENUMber and DELETE. It uses all three processor registers and locations $5F and $60. On top of that the entry requirement is the line number in low/high byte form in locations $14 and $15.

ROM LISTING

```
A613 A5 2B          LDA $2B
A615 A6 2C          LDX $2C
A617 A0 01          LDY #$01
A619 85 5F          STA $5F
A61B 86 60          STX $60
A61D B1 5F          LDA ($5F),Y
A61F F0 1F          BEQ $A640
A621 C8             INY
A622 C8             INY
A623 A5 15          LDA $15
A625 D1 5F          CMP ($5F),Y
A627 90 18          BCC $A641
A629 F0 03          BEQ $A62E
A62B 88             DEY
A62C D0 09          BNE $A637
A62E A5 14          LDA $14
A630 88             DEY
A631 D1 5F          CMP ($5F),Y
A633 90 0C          BCC $A641
A635 F0 0A          BEQ $A641
A637 88             DEY
A638 B1 5F          LDA ($5F),Y
A63A AA             TAX
A63B 88             DEY
A63C B1 5F          LDA ($5F),Y
A63E B0 D7          BCS $A617
A640 18             CLC
A641 60             RTS
```

Locations $5F and $60 are loaded with the start of BASIC. The high link address is again picked up first to see if the end of program has been reached. The high byte of the line number is checked first. If the value is greater than the required value, the carry will clear and the subroutine is left. If the two values are the same, we go forward to test the low byte values. Failure of either of these checks means that we have not reached the required line and have to go ahead and get the address of the next line. When the low bytes are checked if they are equal, or the carry flag clear, the routine is terminated. On failing to find the desired line, or on finding a higher one, the link addresses are gathered in and we branch back to check the next line.

Due to the way the checks are made, the routine can be left in one of two states. In the first, the exact line number has been found, in which case the address in $5F and $60 will be what you require. The second state will be that there is no such line number and the routine returns the address of the next highest line. These conditions can be tested in the calling routine by examining the carry flag on return. If the carry is set then the actual line number was found, and if clear it was not.

7 Programming aid routines

Introduction

In Chapter 5 we gave routines to help in the preparation and editing of BASIC programs. These routines were themselves in BASIC, so were slow and had to be tagged onto the end of the resident program. This chapter not only puts these routines into 6502 machine code, but also extends their capabilities. In addition the following are included: OLD, RENUM, DELETE, MERGE, APPEND, DUMP, TRACE, CODER, HEX, BIN, TEN, TWO, AUTO, and MEM.

Our object has been to show you that with a little thought and perseverance, adding new BASIC commands is well within your grasp. Most of the routines start with an explanation of what we wish to achieve and how it is possible to do it. This is followed by the assembly listing and the label addresses used. These are provided for assemblers which do not allow the use of labels (Supermon) and with relocation in mind. Finally, a byte-by-byte explanation of the routine is given.

At the beginning of each routine, the command name and parameters are given for use in the UTILITY.

Renumber

COMMAND SYNTAX
RENUM start line number or 0, increment, new start line value

Using 0 as the first parameter will indicate that the whole program requires renumbering. If a start line is set, it will renumber from that line to the end of the program:

for example, RENUM 0,10,100
RENUM 100,10,200

Later in the chapter we will discuss an AUTO routine. This is of use when typing in programs where the line numbers are sequential and of a fixed step. Renumbering a program makes it easier to read and opens up space to incorporate new lines.

The system we are going to use is known as a two pass system. The first pass will renumber commands that have line numbers associated with them. This is not as straight-forward as it might at first appear as the commands THEN and RUN have optional line numbers.

There are cases where we do not need to look for a 'renumbering command'. These will be after a DATA or REM token is encountered, or when inside quotes. In the latter case, we just loop until the next quote or the end of the line is found, whichever is soonest. The procedure on finding the tokens is simply to go to the next statement.

On finding a line number after a command, we convert it from its stored ASCII form to a two byte number. If it is less than the 'start line number', renumbering is not required. When it is not, we calculate its new value, convert it to ASCII, and overwrite the original.

Once all the directive line numbers have been dealt with, the simple task of actually changing the line numbers themselves is carried out.

We will be using many zero page locations in the routine and so to help you to follow the routine, a list of the main ones and what they control is given below:

$FB and $FC	Address of the current BASIC line being worked on
$FD and $FD–	Address of the next line – the links of the current line
$49–	Stores the position in the current line, the line marker or Y register
$C9 and $CA	Line number of first line to be renumbered
$41 and $42–	Address of the first line to be renumbered
$BC–	Value of increment between new line numbers
$BD and $BE–	Value of the new start line number
$B9 and $BA–	Starts with the same values as $BD and $BE and is changed whilst calculating the new line number for directives after keywords.
$58 and $59–	Starts with the same values as $41 and $42 and is incremented to give the actual new number of a directive command

ASSEMBLY LISTING

```
  7 OPEN          =  $8933
  8 CLOSE         =  $888B
  9 *=$89C5
 10                JSR $81F5    ! GET PARAMETER
 20                JSR $AEFD    ! CHECK COMMA
 30                LDA $14
 40                STA $C9
 50                LDA $15
 60                STA $CA
 70                JSR $81F5    ! GET PARAMETER - INC
 80                JSR $AEFD    ! CHECK COMMA
 90                LDA $14
100                STA $BC
110                JSR $81F5    ! GET PARAMETER
                                  - NEW START LINE #
```

```
120            LDA $14
130            STA $BD
140            LDA $15
150            STA $BE
160            LDA $2B
170            STA $FB
180            LDA $2C
190            STA $FC
200            LDA $CA
210            BNE FINDS
220            LDA $C9
230            BNE FINDS    ! IS START LINE INPUT 0
240            LDY #$02
250            LDA ($FB),Y
260            STA $C9      ! GET FIRST PROG LINE #
270            INY
280            LDA ($FB),Y
290            STA $CA
300 FINDS      LDA $C9
310            STA $14
320            LDA $CA
330            STA $15
340            JSR $A613    ! FIND START LINE ADD
350            BCS STORE    ! LINE FOUND
360            LDX #$15
370            JMP $A437    ! ERROR - ILLEGAL DIRECT
380 STORE      LDA $5F      ! STORE START LINE ADD
390            STA $41
400            LDA $60
410            STA $42
420 START      LDY #$00
430            LDA ($FB),Y
440            STA $FD      ! GET LINKS TO NEXT LINE
450            INY
460            LDA ($FB),Y
470            STA $FE
480            BNE CONT     ! NOT END OF BASIC PROG
490            JMP RENUM    ! CHANGE LINE NUMBERS
500 CONT       INY          ! SKIP LINE NUMBERS
510            INY
520 NEXT       INY
530            LDA ($FB),Y  ! GET CHAR OF LINE
540            BNE CONT1    ! NOT END OF LINE
550 LINE       LDA $FD      ! PUT NEXT LINE IN
                              LINE REGISTERS
560            STA $FB
```

```
570              LDA $FE
580              STA $FC
590              BNE START    ! ENFORCED - NEXT LINE
600  CONT1       CMP #$22     ! IS IT A QUOTE
610              BNE CONT2    ! NO
620  QUOTE       INY
630              LDA ($FB),Y  ! LOOK FOR NEXT
                                QUOTE OR LINE END
640              BEQ LINE     ! END OF PROG LINE
650              CMP #$22     ! QUOTE?
660              BNE QUOTE    ! NO
670              BEQ NEXT     ! YES - NEXT CHAR
680  CONT2       CMP #$8F     ! REM TOKEN?
690              BEQ LINE     ! YES - NEXT LINE
700              CMP #$83     ! DATA TOKEN?
710              BEQ LINE     ! YES - NEXT LINE
720              CMP #$A7     ! THEN TOKEN?
730              BEQ THEN     ! YES
740              CMP #$8A     ! RUN TOKEN?
750              BEQ THEN     ! YES
760              CMP #$89     ! GOTO TOKEN?
770              BEQ CONT3
780              CMP #$CB     ! GO TOKEN?
790              BNE NOGO     ! NO
800  SPACE       INY
810              LDA ($FB),Y
820              CMP #$20
830              BEQ SPACE
840              CMP #$A4     ! TO TOKEN AFTER GO?
850              BEQ CONT3
860  NOGO        CMP #$8D     ! GOSUB TOKEN
870              BEQ CONT3    ! YES
880              CMP #$E6     ! RESET TOKEN
890              BEQ CONT3
900              BNE NEXT     ! NO RELEVENT TOKEN
910  THEN        INY
920              LDA ($FB),Y  ! GET NEXT BYTE
930              CMP #$20     ! IS IT A SPACE
940              BEQ THEN     ! YES - SKIP IT
950              CMP #$30     ! LOOK FOR A NUMBER
                                IN ASCII
960              BCS NUMBER   ! FOUND A NUMBER?
970              DEY
980              BNE NEXT     ! NOT LINE # AFTER THEN
990  NUMBER      DEY
1000             CMP #$3A     ! IS  IT A NUMBER
```

```
1010           BCS NEXT     ! NO LINE #
1020 CONT3     INY
1030           LDA ($FB),Y  ! GET NEXT BYTE
1040           CMP #$20     ! A SPACE?
1050           BEQ CONT3    ! YES
1060           STY $49      ! STORE LINE MARKER
1070           DEY
1080           LDX #$00     ! COUNTER FOR NO OF
                              CHARS IN NUMBER
1090 DIGITS    INY
1100           LDA ($FB),Y
1110           CMP #$30     ! NUMBER IN ASCII ?
1120           BCC CONT4    ! NO END OF LINE#
1130           CMP #$3A     ! NUMBER IN ASCII ?
1140           BCS CONT4    ! NO END OF LINE#
1150           STA $0200,X  ! STORE IN INPUT BUFFER
1160           INX
1170           BNE DIGITS   ! ENFORCED GET NEXT BYTE
1180 CONT4     LDA #$3A     ! STORE COLON AS
                              END MARKER
1190           STA $0200,X
1200           STX $BF      ! NO OF CHARS IN LINE
1210           LDA #$02     ! SET CHRGET TO
                              START OF BUFFER
1220           STA $7B
1230           LDA #$00
1240           STA $7A
1250           JSR $81F5    ! CHANGE INTO REAL NOS
1260           LDA $14
1270           STA $C3
1280           LDA $15
1290           STA $C4
1300           CMP $CA
1310           BEQ CHECK2
1320           BCS CONT5    ! > START LINE
1330 NORE      LDA $49      ! NO RENUM OF DIRECTIVE
1340           ADC $BF
1350           TAY
1360           JMP COMMA    ! CHECK FOR ON COMMAND
1370 CHECK2    LDA $C3
1380           CMP $C9
1390           BCC NORE
1400 CONT5     LDA $BD
1410           STA $B9      ! TRANS START ADD TO
                              WORKING REGISTER
```

```
1420            LDA $41       ! TRANS START ADD TO
                                WORKING REGISTERS
1430            STA $58
1440            LDA $BE
1450            STA $BA
1460            LDA $42
1470            STA $59
1480 FINDL      LDY #$00
1490            LDA ($58),Y   ! SEARCH FOR LINE NO
1500            STA $5A       ! SAVE LINKS
1510            INY
1520            LDA ($58),Y
1530            STA $5B
1540            BNE CONT6     ! NOT END OF PROG
1550            LDY #$02
1560            LDA ($FB),Y   ! GET LINE# FOR
                                ERROR MESSAGE
1570            STA $39
1580            INY
1590            LDA ($FB),Y
1600            STA $3A
1610            LDX #$11
1620            JMP $A437     ! ERROR - UNDEF'D
                                STATEMENT
1630 CONT6      INY
1640            LDA ($58),Y   ! GET AND STORE LINE NO
1650            STA $B7
1660            INY
1670            LDA ($58),Y
1680            CMP $C4       ! COMPARE FOR SAME LINE
1690            BNE NEXTLINE! NOT SAME
1700            LDA $B7
1710            CMP $C3
1720            BEQ FOUNDL
1730 NEXTLINE   LDA $B9
1740            CLC           ! INC REGS TO CALC
                                NEW LINE NO
1750            ADC $BC
1760            STA $B9
1770            BCC NOINC
1780            INC $BA
1790 NOINC      LDA $5A
1800            STA $58       ! PUT NEXT LINE ADD
                                IN CURRENT REG
1810            LDA $5B
1820            STA $59
```

```
1830            BNE FINDL    ! ENFORCED - CHECK
                               NEXT LINE
1840 FOUNDL     LDX $B9
1850            LDA $BA      ! MSB OF NEW LINE
                               DIRECTIVE
1860            JSR $847F    ! CONVERT TO ASCII
                               - INPUT BUFFER
1870            LDA $BF
1880            STA $3E
1890            CPX $3E      ! DOES MEM HAVE TO MOVE
1900            BEQ NOMOVE
1910            BCS OPENUP
1920            JSR CLOSE    ! REQUIRES LESS
                                   SPACE
1930            JMP NOMOVE
1940 OPENUP     JSR OPEN     ! REQUIRES MORE
                                   SPACE
1950 NOMOVE     LDY $49
1960            LDX #$00
1970 NEXTF      LDA $0200,X  ! GET NEW NO IN ASCII
1980            BEQ COMMA    ! END OF NUMBER
1990            STA ($FB),Y  ! STORE IN PROG
2000            INY
2010            INX
2020            BNE NEXTF    ! ENFORCED
2030 COMMA      LDA ($FB),Y  ! COMMA MEANS ON USED
2040            CMP #$2C
2050            BEQ ANOTHER
2060            DEY
2070            JMP NEXT     ! GET NEXT TOKEN
2080 ANOTHER    JMP CONT3    ! NEXT LINE - ON COMMAND
2090 RENUM      LDY #$00
2100            LDA ($41),Y  ! GET AND STORE LINKS
2110            STA $5A
2120            INY
2130            LDA ($41),Y
2140            STA $5B
2150            BNE CONT8    ! NOT  END OF PROGRAM
2160            PLA          ! REMOVE RETURN
                                   ADDRESS
2170            PLA
2180            JMP $A474    ! GOTO READY FOR BASIC
2190 CONT8      INY
2200            LDA $BD      ! NEW LSB LINE NO
2210            STA ($41),Y  ! CHANGE PROG
2220            INY
```

```
2230                LDA $BE       ! NEW MSB LINE NO
2240                STA ($41),Y
2250                CLC
2260                LDA $BD
2270                ADC $BC       ! INC NEW LINE# BY
                                    INCREMENT
2280                STA $BD
2290                BCC CONT9
2300                INC $BE
2310 CONT9          LDA $5A
2320                STA $41
2330                LDA $5B
2340                STA $42
2350                BNE RENUM     ! ENFORCED NEXT LINE
```

```
8B5F ANOTHER        8AD2 CHECK2
888B CLOSE          8B55 COMMA
8A2D CONT           8A3E CONT1
8A4D CONT2          8A8D CONT3
8AAA CONT4          8AD8 CONT5
8B05 CONT6          8B74 CONT8
8B89 CONT9          8A99 DIGITS
8AE8 FINDL          8A03 FINDS
8B2C FOUNDL         8A34 LINE
8A2F NEXT           8B4A NEXTF
8B17 NEXTLINE       8A70 NOGO
8B22 NOINC          8B46 NOMOVE
8ACA NORE           8A88 NUMBER
8933 OPEN           8B43 OPENUP
8A42 QUOTE          8B62 RENUM
8A65 SPACE          8A1D START
8A15 STORE          8A7A THEN
```

LINES 10–150: The parameters are gathered here and put into their registers. Commas separating the inputs are also checked, giving 'SYNTAX ERRORS' if not present.

LINES 160–190: The start of the BASIC program is now put into the current line registers, as it is also the address of the first line.

LINES 200–290: If the first parameter input was zero, indicating a full program RENUMber, then the first line number is found and stored in its appropriate register.

LINES 300–410: Although we do not need this at the moment – here we find the address of the start line. We use the ROM routine FIND BASIC LINE (see Chapter 6, page 142). If the carry flag is set, it will mean that the start line requested was not found and an 'ILLEGAL DIRECT' error will be printed. The address, if found, will be stored in $41 and $42.

LINES 420–1020: The byte-by-byte search for the appropriate keywords starts here. We start at the beginning of the BASIC program, no matter what the start line requested. As soon as the link addresses are collected and stored, the end of the program is checked for. Passing this means that only the actual line numbers require changing, so it is off to the final section of the whole routine, which is described later.

To continue finding the tokens, we skip the line numbers as they are not required here. Lines 540 to 590 set the values for the next line, after the end of line zero is discovered, and branches back to process the next line.

There is nothing of interest to us in quotes so on finding one we go into a loop to find a second quote or the end of the line. This is carried out in lines 620 to 670.

The next two tokens checked for are DATA and REM. Encountering these indicates that we can proceed to the next line as there will be nothing further to renumber in these lines.

There are two keywords – RUN and THEN – that may, or may not, have line numbers. These will therefore branch to check this possibility before proceeding. The standard Commodore directive commands are next in line: GOTO, GO TO and GOSUB. The centre keyword is checked in two stages, first for the GO. A loop is then set up to skip over spaces and then the TO token is looked for. All three keywords on being found will cause the routine flow to branch further ahead than RUN or THEN, as it assumes they will have line numbers. If not, then unless you have a line number of 0, an error will be detected later.

The last keyword to be checked for is one of our new ones: RESET.

If line 900 is reached then we have not found a relevant keyword and therefore branch back to get the next program byte.

The last part in this section is to check if the next significant byte after RUN or THEN is a number, in ASCII. Spaces are skipped over and checks are made for values between $30 and $39 inclusive to continue.

LINES 1030–1390: The line numbers after the keywords will be in ASCII form and the line itself in two byte form. To do our calculations, we want both in the same two byte format. ASCII numerals therefore have to be changed.

After skipping spaces, we store our Y register (so we know where to write our new line number from), which is the line marker. Proceeding, we pick up bytes until a non-numeral is found, and store them in the input buffer. The X register is used to count the number of digits and is stored for later use. To convert the ASCII into the form we require, we use the GET PARAMETER routine. For this to work, we perform two operations. First, we make sure that after the last line number digit there is a non-numeric character by storing a colon there. Secondly, we set CHRGET to point to the first numeral – $0200.

The converted result is taken from registers $14 and $15 and stored in the two we have designated.

We only wish to renumber from the start line in the command. Lines 1290 to 1390 check this by comparing the two values. If no renumber of that particular line is required, we retrieve our line marker and increase it by the number of digits in the directive number, as we do not require to check them again. This is then transferred to the Y register. It will actually point to the byte following the last digit but this is taken into account in what follows. We jump further ahead to a position noted as COMMA (described later) starting at line 2030.

LINES 1400–1830: Having got this far, we have found a number which requires a different value. To find the new value, we have to go through the program from the designated start line and find the line that it points to (remember we have not changed the actual line numbers yet). At the same time, we calculate the new value.

To start with, we take the address of the start line and store it in $58 and $59. We then take the new value for the start line, the third parameter in the command, and place it in $B9 and $BA. The line number we are checking for is held in $C3 and $C4.

As before, we get and store the link addresses, but here if we discover the end of the program has been reached (high link address of zero), an error is present as the line number of the directive was not found. The error produced is the same as when RUNing a program – 'UNDEF'D STATEMENT'. To make it easier for you, we also print out the line number with the error.

As long as this is not encountered, collecting a line number and comparing it for a match comes next. If it is not the one we want, the new start line number is increased by the increment value. This will calculate the value for the following line. The value is only increased on not finding a match, which conversely means that when the line searched for is located, the value is ready and waiting. After incrementing, we transfer the links to the line registers and branch back to check further lines.

LINES 1840–1940: We now have our new line number and need only to insert it into the line after the token, overwriting the original directive.

The new value is in two byte form and so requires converting to ASCII form. We do this by using a routine earlier in the UTILITY. This requires the accumulator to be loaded with the high byte and the X register with the low byte. We then call our 'convert to ASCII' at $847F. This will do the conversion and store the answer in the input buffer, with a zero signifying the end. Also returned is the number of characters in the X register. If this value is the same as the original number, the value stored in location $BF, we can just overwrite with no problems and proceed to line 1950.

The number of characters is transferred to location $3E, via the accumulator. This location is for the memory move routine described in Chapter 6. By comparing the value in the X register (the new number of digits) with the accumulator figure (the old number of digits), we determine if a move is required. If the X value is less, the CLOSE routine is called; if it is greater, the OPEN routine is called; if it is the same, no move is required.

LINES 1950–2020: This leaves us one thing to do which is to write in the new number. First, we reload the Y register with the line marker. This points to the position of the first digit in the number. By increasing X, starting at zero, and Y, we take the digits from the input buffer and store them into the program line. This is repeated until we collect a zero from the buffer, when the branch in line 2020 will fail.

LINES 2030–2080: When we went through the lines checking for tokens, we did not look for the ON statement. The reason for this is that the only time a comma should be used after a line directive following a GOSUB or GOTO is when the ON keyword has been used.

On entering these lines the Y register will point to the byte following the last one checked or stored. We load the accumulator with that byte to see if a comma is present. Finding it means that we branch back to the position just after the token search where it will commence with gathering in the line number directive for processing.

If the comma is not present, the Y register is decreased and we go back to start the search for the next appropriate token. The Y has to be decreased in case it was the end of the line, otherwise it would not have mattered.

LINES 2090–2350: RENUMBERING THE LINE NUMBERS

All the directive line numbers have now been checked and processed where required. The only thing which remains is to renumber the program lines themselves.

This is started from the line number requested in the first parameter of the command and whose address is held in $41 and $42. Its new starting value is held in $BD and $BE, and these will be incremented by the value in $BC for each line. We progressively go through the program inserting the new numbers, in two byte form, until the end of the program is reached. After each line the number value is incremented ready for the next line.

When the renumbering is completed, we take the return address from the stack and jump to the start of BASIC to await further instructions. We do this to break out of the program in the unlikely event of RENUM being initiated within it.

A WORD OF WARNING

We cannot stress strongly enough that a copy of the original program should be saved to tape, or disk, prior to renumbering. We all make mistakes and if the RENUM finds a non-existent line number after, say, a GOTO, then an error is produced. This leaves the program only partially renumbered.

Auto

COMMAND SYNTAX
AUTO first line number, increment

To escape from AUTO simply press RETURN immediately after the line number, as if deleting a single line.

This command removes the need to type in line numbers. The user just decides the start line number and the increment between consecutive lines. To achieve this we want a place to break into the normal flow of BASIC. This is made possible as every time an input line is processed it goes to a vectored jump before it is ready to receive the next. This is called the BASIC Warm Start Vector which is at $0302 and $0303. By changing the address in this vector to a routine of our own we can calculate the line number, put it into ASCII form and then insert it into the keyboard buffer just as if you typed it yourself. Having completed this, we then return to the input routine for you to make up the program line.

ASSEMBLY LISTING

```
*=$842E
            JSR $81F5    ! GET PARAMETER
            JSR $AEFD    ! CHECK COMMA
            LDA $14      ! 1ST LINE# - LOW
            STA $FB      ! SPARE ZERO PAGE
                                LOCATION
            LDA $15      ! 1ST LINE# - HIGH
            STA $FC      ! SPARE ZERO PAGE
                           LOCATION
            JSR $81F5    ! GET PARAMETER
            LDA $14      ! INCREMENT
            STA $FD      ! SPARE ZERO PAGE
                           LOCATION
            LDA #<AUTO
            STA $0302    ! START BASIC VECT LOW
            LDA #>AUTO
            STA $0303    ! START BASIC VECT HIGH
AUTO        LDA $0200    ! 1ST CHAR IN BUFFER
```

```
150            BEQ EXIT      ! TURN OFF AUTO
160            LDX $FB
170            LDA $FC       ! NEXT LINE NUMBER
180            JSR ASCII     ! PUT LINE # IN ASCII
190            STX $C6       ! NO OF CHARS IN
                               KEYBOARD BUFFER
200 NEXT       LDA $0200,X   ! PICK UP ASCII
210            STA $0277,X   ! PUT IN KEY BUFFER
220            DEX
230            BPL NEXT
240            CLC
250            LDA $FB
260            ADC $FD       ! INCREMENT LINE NUMBER
270            STA $FB
280            BCC NOINC
290            INC $FC
300 NOINC      JMP $A483     ! READY FOR BASIC
310 EXIT       LDA #$83
320            STA $0302
330            LDA #$A4
340            STA $0303
350            JMP ($0302)
360 ASCII      STX $63       ! LOW BYTE
370            STA $62       ! HIGH BYTE
380            LDX #$90
390            SEC
400            JSR $BC49
410            JSR $BDDF
420            JSR $B487
430            JSR $B6A6
440            LDX #$00
450 AGAIN      LDA $0100,X
460            STA $0200,X   ! PUT ASCII CHARS IN
                               INPUT BUFFER
470            BEQ FINISH    ! ZERO CHAR WAS FOUND
480            INX
490            BNE AGAIN
500 FINISH     RTS
```

```
8494 AGAIN        847F ASCII
844D AUTO         8472 EXIT
849F FINISH       845B NEXT
846F NOINC
```

LINES 10–130: These lines are only used when the AUTO command is active. We take the parameters of the command and place the start line number in $FB and $FC and the increment in $FD. As we only take the low byte of the increment, the multiples over 256 are ignored. The only syntax check is made in line 20, where we check for a comma between the two parameters. Lastly in this section we change the vector address to point to the AUTO numbering which starts in line 140.

LINES 140–300: The first thing is to see if the first character in the input buffer is zero. This will signify no BASIC coding was inserted in the line and that you want to cancel the AUTO routine. When a BASIC line is typed, the line number is put into the input buffer. During the processing stage of inserting the line into the program BASIC takes out the line number and moves the rest of the line back up the buffer to overwrite it. This means that if the first input after a line number was a RETURN (BASIC inserts a zero for that as an end of line marker) then the first character in the buffer will be a zero after the line number has been removed. Therefore, on finding a zero we will branch off to exit the AUTO mode.

Assuming that we are still auto-numbering, we take the values in $FB and $FC and go off to convert them to ASCII form. This we will come across shortly. On returning from that subroutine the X register will have the number of ASCII characters in the line number and they will be in the input buffer. The X value is stored in the register which tells the operating system how many characters will be in the keyboard buffer. Having done that, we transfer the characters from the input buffer to the keyboard buffer.

We now set the line number for next time by adding the increment in $FD to the values in $FB and $FC.

That is all there is to do, so we return you to the normal BASIC flow where the input routine will take the line number from the keyboard buffer, place it in the input buffer and print it on the screen.

LINES 310–350: These lines will be operated when you want to exit from AUTO. All we do is restore the BASIC Warm Start Vector to its initial value and then return you to BASIC to wait for your next instruction.

LINES 360–500: CONVERT TO ASCII: This subroutine will also be used by other commands when they require a one or two byte number converted into ASCII form. The subroutine is entered with the low byte of the number in the X register and the high byte in the accumulator.

The conversion is carried out by four ROM routines, but before we can call them we have four items to set. First, the number to be converted is transferred to locations $63 (the low byte) and $62. These are part of the floating point accumulator #1 (FAC#1) which is the main number manipulation area for BASIC. The other two prevent certain actions in the conversion process. Setting the carry flag will bypass a routine that will perform the complement of the number and loading X with $90 will set the Exponent byte of FAC#1 to avoid getting an answer in exponent form.

The first ROM routine visited clears all the bytes in the FAC#1 (or sets them to default values) which we have not dealt with. The next routine does the actual conversion. The remaining routine puts the result into a string and places it at the bottom of the stack area. The last byte placed there will be zero to mark the end.

We cannot leave it there as BASIC often uses this area. We therefore transfer it to the input buffer where it can be taken and used by the coding calling this routine. On exit the value in the X register will be the number of ASCII characters in the conversion.

Merge and append – combining BASIC programs together

COMMAND SYNTAX
MERGE "program title",device
APPEND "program title",device

The default device is tape and if there is no program title, the first program found on the tape will be used.

Merging programs together means that they are weaved together in program line order. The result is as if you typed in the lines of the merging program at the keyboard. This also means that if the programs both have lines with the same number, the ones in the merging program will overwrite the original.

Appending a program to another is simply a process of tagging it onto the end of the one in memory, irrespective of line numbers.

Merging is the more complicated of the two programs, but is not really complicated in itself. Both programs are initially loaded at the end of the current memory program, APPEND overwriting the last two bytes whilst MERGE comes just after them. The last two bytes of the program are the unique link address of zero, signifying the end of a BASIC program. By overwriting them on APPEND, we achieve our aim immediately and all that remains is to amend the link addresses to continue the program flow and to reset the end of program pointer.

In merging we take each new line in turn and insert it in the main program – if we had overwritten the original end links we would both merge and append which we do not want. To incorporate the new lines we make use of the normal BASIC input routine. After you input a program line and press RETURN, BASIC takes off the line number and then tokenizes all the BASIC keywords. At this point the line number is in two registers, in a two byte form rather in than the ASCII form typed in, and the line's content has been moved to the beginning of the input buffer. There is a counter of the number of bytes in the line which is four greater to incorporate the line number and link address. BASIC therefore knows the total space required for the line. We will enter the ROM routine at this point with the appropriate data set. Unfortunately, it is not a subroutine but finishes up waiting for an input. We therefore have to change the same vectors as in the AUTO routine (the BASIC Warm Start) to point us back to continued merging until we reach the end.

ASSEMBLY LISTING

```
  9 *=$87A7
 10              JSR SETADD     ! SET ADDRESSES
                                  FOR MERGE PROG
 20              STX $2B        ! SET MERGE PROG START
 30              STY $2C
 40              LDY #$00       ! PUT ZERO AT 1ST
                                  LOCATION
 50              TYA
 60              STA ($2B),Y
 70              JSR LOAD
 80              STX $2D        ! END OF MERGE PROG
 90              STY $2E
100              JSR $A533      ! RECHAIN MERGE PROG
110              JSR RESET1     ! RESTORE POINTERS TO
                                  ORIG PROG
120              LDA #<MERGE
130              LDX #>MERGE
140              STA $0302      ! CHANGE WARM START
                                  VECTOR TO MERGE
150              STX $0303
160 JOIN         LDA #$01
170              STA $7B
180              LDA #$FF
190              STA $7A        ! SET UP CHRGET
200              LDY #$00
210              LDA ($FB),Y    ! GET AND STORE LINKS
220              STA $FD
230              INY
240              LDA ($FB),Y
250              STA $FE
260              BEQ EXIT       ! END OF MERGE PRG
270              INY
280              LDA ($FB),Y    ! GET AND STORE LINE NO
290              STA $14        ! BASIC EXPECTS THEM
                                  IN THESE LOCATIONS
300              INY
310              LDA ($FB),Y
```

```
320             STA $15
330             LDX #$04      ! SET COUNTER TO
                                INCLUDE LINKS AND
                                   LINE#
340 NEXT        INX
350             INY
360             LDA ($FB),Y   ! GET LINE DATA
370             STA $01FB,X   ! STORE IN INPUT BUFFER
380             BNE NEXT      ! NOT END OF LINE 0
390             TXA
400             TAY           ! PUT COUNTER IN Y
410             JSR $A4A2     ! BASIC AFTER CRUNCH
                                ROUTINE
420 MERGE       LDA $FD       ! BASIC WARM START
                                             POINTS HERE
430             LDX $FE       ! PUT LINKS IN LINE
                                REGISTERS
440             STA $FB
450             STX $FC
460             BNE JOIN      ! ENFORCED AS $FE
                                CHECKED FOR 0 EARLIER
470 EXIT        LDA #$83
480             LDX #$A4
490             STA $0302     ! RESTORE WARM START
500             STX $0303
510             JSR $A474     ! READY FOR BASIC
520 LOAD        JSR $E1D4     ! GET LOAD PARAMETERS
530             LDA #$00      ! FOR RELOCATED LOAD
540             STA $B9       ! IN CASE A SEC ADD
                                IN COMMAND
550             LDX $2B
560             LDY $2C       ! SET LOAD START
570             JSR $FFD5     ! KERNAL LOAD
580             BCS ERROR     ! BAD LOAD
590             JSR $FFB7     ! READ I/O STATUS WORD
600             AND #$BF
610             BEQ EXIT1     ! GOOD LOAD
620             JSR RESET     ! RESET POINTERS TO
                                ORIGINAL PROG
630             LDX #$1D
640             JMP $A437     ! LOAD ERROR
650 EXIT1       RTS
660 ERROR       PHA           ! SAVE FOR ERROR
670             JSR RESET
680             PLA
690             JMP $E0F9     ! ERROR DEPENDING ON A
```

```
700 RESET       LDA $FB
710             SEC
720             SBC #$02
730             STA $14
740             LDA $FC
750             SBC #$00
760             STA $FC
770             LDA #$00
780             TAY
790             STA ($14),Y   ! RESTORE TWO ZEROS
                                AT END OF PROG
800             INY
810             STA ($14),Y
820 RESET1      LDA $FD
830             LDX $FE
840             STA $2B
850             STX $2C
860             LDA $FB
870             LDX $FC
880             STA $2D
890             STX $2E
900             RTS
910 SETADD      LDA $2B
920             STA $FD
930             LDA $2C
940             STA $FE
950             LDX $2D
960             LDY $2E
970             STX $FB
980             STY $FC
990             RTS
1000 ! APPEND ROUTINE
1010            JSR SETADD
1020            TXA
1030            SEC
1040            SBC #$02
1050            STA $2B
1060            TYA
1070            SBC #$00
1080            STA $2C
1090            JSR LOAD
1100            STX $FB
1110            STY $FC
1120            JSR RESET1
1130            JSR $A533
1140            RTS
```

8830 ERROR	8803 EXIT
882F EXIT1	87CA JOIN
8810 LOAD	87F9 MERGE
87EB NEXT	8838 RESET
884D RESET1	885E SETADD

This time we are not going to describe the program in line number order. There are three subroutines in the body of the program used both by MERGE and APPEND and we will deal with these first.

LINES 910–990: SETADD: This simply takes the start and end addresses of the original program and temporarily stores them. On coming out of the subroutine, the Y register will contain the high byte of the end address and the X register, the low value.

LINES 700–900: RESET: The first 12 lines will only be encountered when there is an error in loading the secondary program. These simply ensure that the end of program zeros are at the end of the original program. This will mean that when exiting from either command your original program is intact before starting.

The remaining lines are the reverse of SETADD, that is, they take the values in the temporary registers and place them in the program end and start registers. These last lines are called in the assembly listing as RESET1.

LINES 520–690: LOAD: The first thing this subroutine does is to call one resident in the BASIC ROM used by the standard LOAD and SAVE commands. It gathers up the parameters and sets various registers according to that information, and as it is there we also make use of it. We are going to do a relocated load and if a secondary address is present this will override our objective. To correct this we load location $B9 with zero to bring back the state for a relocated load.

The KERNAL LOAD routine expects the start address of the load in the two processor registers, X and Y, with the former holding the low byte of that address. The accumulator is the flag for either a load or a verify operation. The value for load is zero, the other being one, which was set whilst confirming the secondary address. The KERNAL LOAD routine is situated at $FFD5. Error checking comes now in the order of operations – you may have put in the wrong tape or disk. The first indicator to a bad load is the carry flag being set; if this is so, then we branch off to deal with it. We have to check the I/O status word if the carry is clear. This is achieved by calling another KERNAL routine, at $FFB7. The result coming out of this call is ANDed with the value $BF and everything is fine if the zero flag is set. The error given for any other outcome is 'LOAD ERROR'.

On the first check we go to line 660 if the carry was set. The value in the accumulator will be used for the error so we temporarily store this on the stack. The reason for this is to reset all the pointers we altered to give you back your original program. This is done by a call to RESET, described above. Once done we retrieve the accumulator value and jump to part of the BASIC loading routine for an error to be generated, based on the accumulator value.

MERGE

LINES 10–150: First of all we call the SETADD routine. From this we can set the start address of the merging program to immediately after the memory program. BASIC expects the first byte to be zero and therefore we oblige it. Having done that, we load the program we want to merge. The address that is returned from loading is placed in the end of program address. At this point as far as BASIC knows the only program in memory is your merge program. The link addresses have to be set up so we know where the lines start. This is done by a call to the ROM routine at $A533 which will do this for us. From now on we want BASIC to respond to the original program and therefore we reset all the registers back to their original values through RESET1.

To merge the lines into the master program we use the program line input routine in the BASIC ROM. This is not a stand alone routine but ends up at the BASIC Warm Start after each line has been processed. It follows that, as in the AUTO routine, we will have to alter the vector pointing to that position to divert to this routine until all lines are merged. This is done in the lines 120 to 150. The addresses will be in the location of line 420.

LINES 160–460: These are the instructions which actually combine your two programs. First, CHRGET is set to a position one place before the start of the input buffer. We now turn our attention to the merging program lines. The address of the first line will still be in locations $FB and $FC following the SETADD routine called at the beginning. Using this information, we get the link addresses and store them for later. We use this system a lot in our routines but in this case it is vital. We have left no room between the programs so that when the line is transferred the master program will be longer, overwriting the needed link addresses. The check is also made for the end of the merge program, the usual high byte zero link address.

Next we take the two byte line number and place it in registers so that BASIC will know where to find them, $14 and $15. The contents of the line, including the zero byte end marker, are now transferred to the input buffer starting at $0200. The listing shows $01FB but there is a reason for this. The X register will come out with a value of the number of bytes in the line, but to account for the four bytes holding the link addresses and the line number, it starts with a value of 4. This is one more than required but the ROM routine will compensate. This means that with the initial value of $04 in X the first location written to will be

$0200. The ROM routine we are using wants the number of bytes in the Y register rather than the X, so we oblige by transferring them via the accumulator.

We are now ready to use the ROM to merge our line into the main BASIC program. We join the ROM just after the coding that turns the keywords into tokens – ours are in that state already. As far as the ROM is concerned you have typed in the line and will put it in the program as such.

The BASIC Warm Start Vector will bring us back after inserting the line to LINE 420 in the above listing. We now put the address of the next line into the working registers and branch back to deal with it. The branch instruction will always succeed as we have checked previously for a zero value in $FE.

These lines will be repeated until all the merge program lines have been assigned to the master program.

LINES 470–510: The merge is complete so we restore the BASIC Warm Start Vector to its normal setting and return to BASIC where it will await further commands.

APPEND

LINES 1010–1140: The first thing, as in MERGE, is to call the SETADD routine. On coming out though, it is slightly different. The new BASIC program start has to be reduced by two so will overwrite the end of memory program links. The appending program will load directly after the final line of the master program. On completion of the load, we store the loading end addresses (not in the end of program registers), but overwrite the original stored values, set by SETADD. This means that on resetting the values, by RESET1, we end up with the original start and the end marker corresponding to end of the appended lines. The final thing to do is to set the link addresses to follow as one program. This is carried out by the ROM routine at $A533. The two programs have been joined together with our own form of 'superglue'.

Delete

COMMAND SYNTAX
DELETE first line number,[second line number]

The first line number to be deleted and the comma are essential and if missing will give errors. The second line number is optional in that if you want to delete to the end of the program you omit it; otherwise insert a number.

Deleting a line of BASIC program is easy, you just type in the line number followed by a return. There is no real hardship in deleting one or two lines but longer blocks become tedious and time-consuming.

DELETE will rid you of a block of lines with one command. To do this we use the same ROM routine as if deleting one line. What the BASIC ROM does is to take the address of the line to be deleted and the link address within that line. It then takes the program starting at that link address to the end of the program and moves it to the address of the line to be deleted. For example, if we have a line whose address is $0901, its link address is $0925 (the start of the next line), and the end of the program is $0A45, the block to move is $0925 to $0A45 with its new starting address of $0901. Hence, the line at $0901 is overwritten or deleted. By the way, the variables and the arrays are also moved along with the block.

It therefore goes to show that if we get the address of the next line after the 'second line number' and place it in the link address of the 'first line number', a whole block of lines will be overwritten at once. Where there is no 'second line number' we take the end of program address and deduct two from it. This will give us the address of the two zero bytes at the end of a program. The first line number is placed in the input buffer, with a zero at the end of it signifying no further data, and goes to ROM as if you typed it in.

Two listings follow for this command. The reason for this is explained later.

ASSEMBLY LISTING1

```
  9 *=$8F44
 10               BCC DEL       ! PARAMETER A NUMBER
 20 SYNTAX        JMP $AF08     ! GENERATE SYNTAX ERROR
 30 DEL           JSR $81F5     ! GET 1ST LINE NUMBER
 40               JSR $A613     ! FIND LINE LOCATION
 50               BCS FOUND     ! LINE NUMBER FOUND
 60               LDX #$15      ! ILLEGAL DIRECT ERROR
 70               JMP $A437     ! ERROR ROUTINE
 80 FOUND         LDA $5F       ! PUT LINE ADDRESSES
 90               STA $FB       ! IN STORAGE
100               LDA $60
110               STA $FC
120               JSR $0079     ! CHECK FOR COMMA
130               CMP #$2C
140               BNE SYNTAX    ! NOT FOUND
150               JSR $0073     ! GET NEXT BYTE
160               BNE NUMERAL   ! A SECOND LINE NUMBER
170               SEC           ! PREPARE FOR SUBTRACT
180               LDA $2D       ! END OF PROGRAM
190               SBC #$02      ! DEDUCT BY TWO
200               STA $5F       ! READY FOR DELETION
210               LDA $2E       ! END OF PROGRAM
```

```
220              SBC #$00      ! IN CASE OF PAGE
                                 CROSSING
230              STA $60       ! READY FOR DELETION
240              BNE CONT      ! ENFORCED $2E CAN'T
                                 BE ZERO OR 1
250 NUMERAL      BCS SYNTAX    ! NO NUMBER
260              JSR $81F5     ! GET 2ND LINE NUMBER
270              INC $14       ! SO WE GET
                                 FOLLOWING LINE
280              JSR $A613     ! FIND LINE
290              LDA $FC       ! CHECK IF 1ST NO IS
                                 SMALLER THAN 2ND
300              CMP $60
310              BCC CONT
320              BNE SYNTAX
330              LDA $FB
340              CMP $5F
350              BCS SYNTAX
360 CONT         LDY #$00
370              LDA $5F
380              STA ($FB),Y   ! STORE ADRESS
390              INY
400              LDA $60
410              STA ($FB),Y
420              INY
430              LDA ($FB),Y   ! GET 2 BYTE LINE NO
440              TAX
450              INY
460              LDA ($FB),Y
470              JSR $847F     ! CONVERT TO ASCII
480                            ! AND PUT INTO
                                 START OF INPUT BUFFER
490              PLA           ! REMOVE RETURN ADDRESS
500              PLA
510              LDX #$FF      ! TO INITAILIZE
                                 INPUT BUFFER
520              LDA #$01
530              JMP $927D     ! WILL DELETE LINE
                                 AND RETURN TO BASIC


8F91 CONT           8F49 DEL
8F56 FOUND          8F79 NUMERAL
8F46 SYNTAX
```

ASSEMBLY LISTING2

```
  9 *=$927D
 10               LDA $0302      ! SAVE WARM START
 20               STA ADD+1
 30               LDA $0303
 40               STA HADD+1
 50               LDA #<ADD      ! CHANGE WARM START
 60               STA $0302
 70               LDA #>ADD
 80               STA $0303
 90               JMP $A486      ! DELETE BLOCK
100 ADD           LDA #$83       ! RESTORE WARM START
110               STA $0302
120 HADD          LDA #$A4
130               STA $0303
140               JSR $A533      ! RECHAIN LINES
150               CLC            ! PREPARE FOR ADD
160               LDA $22
170               ADC #$02       ! INCREASE FOR
                                   VARIABLE START
180               STA $2D        ! START OF VARIABLES
190               LDA $23
200               ADC #$00       ! IN CASE OF CARRY
210               STA $2E
220               JSR $A660      ! CLR
230               JMP $A474      ! READY FOR BASIC


9296 ADD            929B HADD
```

LISTING1

LINES 10–110: These instructions deal with the 'first line number'. The routine first checks the carry flag, set or unset by CHRGET on entering, and if set a SYNTAX ERROR is generated as the first byte after the delete token was not a numeral. A call to our GET PARAMETERS routine is next, immediately followed by a visit to the ROM routine FIND BASIC LINE. The result from GET PARAMETERS is in the registers used to call FIND BASIC LINE. On returning from the latter, if the carry is not set, then the line was not found and we therefore generate a further error. The address of the first line is placed in locations $FB and $FC.

LINES 120–280: The remaining parameter is now dealt with. CHRGET is positioned to where the comma should be, so we call CHRGOT to see if it is there. A call to CHRGET now will get the first byte of the second line number. If no line number is present, the zero flag will be set. In that case we gather in the address of the end of the program, deduct two from it, and store the result in registers $5F and $60.

If the zero flag was not set, we make a further check as earlier to see if the byte picked up by CHRGET was a numeral. GET PARAMETER is called to get the second line number and the low byte result in $14 is increased by one. This is done as we do not require the address of that line but rather the one following it. After the visit to FIND BASIC LINE the address in $5F and $60 will be the next line, whether its line number is one or ten greater than the 'second line number'.

LINES 290–350: These instructions check to see that the address of the 'second line number' is higher than that of the first, otherwise a SYNTAX ERROR is given.

LINES 360–480: Here we insert the second address we found into the link address position of the first line. We then get the line number, in its two byte format, putting the low byte in the X register and the high into the accumulator. We now call another routine which we coded at $847F in the UTILITY where the number will be converted to ASCII and placed in the input buffer, starting at $0200, with a zero at the end.

LINES 490–530: From the stack we remove the return addresses which were placed there on entering DELETE. The X register and the accumulator are given the address of the input buffer less one which will be the CHRGET address. The final thing is to jump to the second listing.

LISTING2

LINES 10–230: The ROM routine that we will use is not a subroutine but ends up at the BASIC Warm Start Vector. We want to return here so we first store its present values and replace them to point back to these lines. We now go to ROM where it will treat the number in the input buffer as if you were deleting a single line from the keyboard, but as we have changed the link address it will delete more than one.

On returning, we restore the BASIC Warm Start Vector. We now subject the program to the rechain routine – not that it requires it, but from this routine we can calculate the end address. From the address the rechain routine ends with, we add two and set the end of program registers. A call to the CLR routine will set the remaining variable addresses. Finally, we jump to BASIC, printing 'READY' and give you back control.

The reason for two listings is due to the way in which the ROM memory moving routine sets the end of program address. We came across this when testing the UTILITY. The BASIC normally expects lines of around 80 characters and definitely no more than 255. Mainly for the latter reason the ROM routine only decreases the end address by the maximum of a page. It does not affect the deletion, but it did not make the required reduction in memory used. The second listing was added to overcome the times when the number of lines took more than 256 bytes. Thus in the second listing we were able to set the addresses ourselves.

Memory – Display number of bytes free

COMMAND SYNTAX
MEM

There are no parameters in this command. The command is available only in direct mode. If found in a program the routine is not carried out.

BASIC has a command that prints out the amount of space available to it. It is FRE(x) where x is a dummy argument. Unfortunately it returns, when used with PRINT, an integer value which means any value over 32767 ($7FFF) will be a negative number. For example, if the number of bytes free is 36500, the result printed would be -29035. If you add that, with the sign, to 65535 ($FFF), you will arrive at the true figure of 36500. We produce here a short routine to print out the correct value straight away. Having said that, with the UTILITY in place, the maximum space available is less than 32768 and so FRE(x) will always print out the correct value.

The first thing to do is call a ROM routine to do a 'garbage collection'. It is at $B526 and tidies up the variable and string area. It will reset the necessary registers after the compaction. The area of memory that is unused will be from the end of arrays to the beginning of the area used by strings. If we take the higher address from the lower, we will have the number of free bytes available.

The routine that we have used to print the result to the screen is a subroutine of the HEX command, which is described later. Suffice to say that on calling this subroutine with the low byte in the Y register and the high in the accumulator, it will convert it to ASCII and print the result to the screen.

To check whether you are in direct mode, we look at location $9D(157). This will hold $80 (128) for direct or $00 for program mode.

ASSEMBLY LISTING

```
 9 *=$85FC
10           LDA $9D      ! DIRECT OR PROGRAM
20           BNE MEM      ! DIRECT ONLY
30           RTS          ! PROGRAM NOT EXECUTED
40 MEM       JSR $B526    ! ROM COLLECT GARBAGE
50           SEC          ! PREPARE FOR SUBTRACT
60           LDA $33      ! POINTER START OF
                            STRING STORAGE
70           SBC $31      ! POINTER END OF ARRAYS
```

```
 80           TAY           ! TEMP STORE
 90           LDA $34       ! POINTER START OF
                              STRING STORAGE
100           SBC $32       ! POINTER END OF ARRAYS
110           JMP $85AD     ! CONVERT TO ASCII
                              AND PRINT TO SCREEN
```

```
8601 MEM
```

Coder

COMMAND SYNTAX
CODER
There are no parameters to this command.

How many times have you picked up a listing from a magazine and wondered what graphic symbol is in that PRINT statement? Is it a shifted N graphic or shifted L? How many have been used together, is it 2 or 3? You then come across a colour code and have to look it up in the manual to remember which colour to program. Owners of non-Commodore printers also have a problem as these symbols and graphics do not print.

We would like to introduce a routine that replaces these graphics with mnemonics. For example, the symbol for clear screen would be replaced by [CLS].

Except for one, all the codes we want to change appear within quotes. That means we have to look through the program for a quote and when found look for ASCII values that we want to change until the end of the line or the second quote appears. Having found one, we also have to look to see if it has been repeated. This done, we will either calculate the new code or find one in a data table. The codes produced will be of a different length from the original, but if it repeats, may be of shorter overall length. To accommodate this, we will use the memory move routines described in Chapter 6.

The one exception we mentioned earlier is the mathematical 'PI' (3.14159, etc). This we also found does not appear on some listings and is essential if in a mathematical equation. This is therefore coded whether in or out of quotes.

Most of the program operation is described after the assembly listing (see below), but before the listing we would like to say a word or two about the data make-up. This can be split into two sections. Graphics that are obtained by using the shift with most of the 'letter' keys can be calculated directly to the ASCII code of that particular letter. The remaining graphics and codes require the use of data tables.

We have employed two tables and stored them out of the way under the BASIC ROM. The first table, the data address table, has the three bytes for each character we are going to encode. The first byte is the ASCII value of the character and is followed by the address within the second table where the data is stored. The second table, the data table, holds all the data for those characters. The data will be the characters printed between the [ ] brackets, and may be of differing length. Because of this the first byte is the number of bytes of data.

What are we going to produce instead of all these graphics and codes? These are listed in Appendix I and are mainly self-explanatory. However, an explanation of two of them is required. If you look at the *Programmer's Reference Guide*, page 74, under 'Other Special Characters', you will see five functions available. Three of these can be achieved more easily than described in the PRG. These are SWITCH TO LOWER CASE, DISABLE CASE-SWITCHING KEYS and ENABLE CASE-SWITCHING KEYS. They can be obtained by simply holding down the CTRL key and appropriate letter. In quotes they will print the appropriate symbol. Out of quotes the action will be carried out. The remaining two 'special characters' are implemented in the way the PRG describes. We have given them codes of [CRG>M] and [CRG>N]. These stand for CTRL REVS GRAPHIC SHIFT RIGHT and the appropriate letter.

ASSEMBLY LISTING

```
  7 OPEN          =  $8933
  8 CLOSE         =  $888B
  9 *=$8B93
 10               LDX #$00      ! INITIALIZE QUOTE
                                  COUNTER
 20               LDA $2B       ! GET AND STORE
                                  START OF BASIC PROG
 30               STA $FB
 40               LDA $2C
 50               STA $FC
 60 LINKS         LDY #$00      ! SET Y TO BEGINNING
                                  OF LINE
 70               LDA ($FB),Y   ! GET ADD OF NEXT LINE
 80               STA $FD       ! STORE FOR LATER
 90               INY
100               LDA ($FB),Y
110               STA $FE
120               BNE CONT      ! NOT END OF BASIC PROG
130               PLA           ! REMOVE RETURN ADDRESS
140               PLA
150               JMP $A474     ! GOTO "READY FOR
                                  BASIC"-END OF CODER
160 CONT          INY           ! SKIP LINE NUMBER
170               INY
```

```
180 NEXT        INY
190             LDA ($FB),Y   ! GET BYTE OF PROG LINE
200             BNE CONT1     ! ZERO SIGNIFIES END
                                OF LINE
210             LDA $FD       ! GET NEXT LINE ADDRESS
220             STA $FB       ! PUT IN CURRENT
                                LINE REGISTERS
230             LDA $FE
240             STA $FC
250             LDX #$00      ! RESET QUOTE COUNTER
260             BEQ LINKS     ! X SETS ZERO FLAG -
                                BRANCH ENFORCED
270 CONT1       CMP #$FF      ! IS IT pi
280             BNE NOPI      ! NO
290             STA $3E       ! STORE VALUE
300             BEQ CONT2     ! ENFORCED
310 NOPI        CMP #$22      ! IS BYTE A QUOTE
320             BNE CHECK     ! NO GO TO SEE IF IN
                                QUOTES
330             INX           ! IT'S A QUOTE SO INC
                                COUNTER
340             CPX #$02      ! IS IT SECOND QUOTE
350             BNE INQUOTES  ! IN QUOTES CODER IN
                                ACTION
360             LDX #$00      ! RESET COUNTER
370             BEQ NEXT      ! ENFORCED
380 CHECK       CPX #$01
390             BNE NEXT      ! NOT IN QUOTES
400 INQUOTES    STA $3E       ! STORE BYTE
410             CMP #$C0      ! IS IT LESS THAN 192
420             BCC COMPARE   ! YES
430             SBC #$60      ! NO SUBTRACT 96
440 COMPARE     CMP #$60      ! IS IT > OR = TO 96
450             BCS CONT2
460             CMP #$21      ! IS IT LESS THAN 21
470             BCS NEXT      ! CHARS 21 - 95
                                DON'T REQUIRE CODING
480 CONT2       STA $3D       ! STORE REVISED CHAR
                                VALUE
490             STY $49       ! STORE LINE MARKER
500             STX $3C       ! STORE QUOTE COUNTER
510             LDX #$01
520 NEXT1       INY
530             LDA ($FB),Y   ! GET NEXT CHAR
540             CMP $3E       ! IS A REPEAT CHAR
550             BNE NEXT2     ! NO
```

```
560           INX
570           BNE NEXT1     ! ENFORCED
580 NEXT2     STX $3E       ! REG - NO OF REPEATS
590           CPX #$02
600           BCS CONT3     ! MORE THAN ONE CHAR
610           DEX
620           LDA $3D       ! GET CHAR BACK
                                 AGAIN
630           CMP #$20      ! IS IT A SPACE
640           BEQ SPACE     ! DON'T CODE SINGLE
                              SPACE
650           LDA #$00
660           STA $40       ! RESET REG WITH ASCII
                              FOR NO OF REPEATS
670           STA $3F
680           BEQ CONT4     ! ENFORCED
690 SPACE     JMP RELOAD    ! RELOAD REGISTERS
                              FOR GET NEXT BYTE
700 CONT3     LDA #$00      ! X HAS LOW VALUE -
                              NO HIGH VALUE
710           JSR $847F     ! NO OF REPEATS INTO
                              ASCII FORM
720           LDX #$00
730           LDA $0200,X   ! GET ASCII INTO REGS
740           STA $3F
750           INX
760           LDA $0200,X
770           STA $40
780 CONT4     LDA $3D
790           CMP #$61
800           BCC CONVERTB
810           CMP #$7B
820           BCS CONVERTB
830 CONVERTA  SEC
840           SBC #$20      ! REDUCE VALUE
850           STA $3D
860           LDX #$07      ! TOTAL NO OF SPACES
                              REQUIRED
870           LDA $40
880           BNE CONT5     ! MORE THAN 9 REPEATS
890           DEX
900           LDA $3F
910           BNE CONT5     ! SOME REPEATS
920           DEX
930 CONT5     CPX $3E       ! FIND HOW MUCH ROOM
940           BEQ NOMOVE    ! RIGHT AMOUNT OF SPACE
```

```
 950          BCS OPENUP    ! NEEDS SPACE IN LINE
 960          JSR CLOSE     ! GET RID OF
                              UNWANTED CHARS
 970          JMP NOMOVE    ! CONT WITH PROG
 980 OPENUP   JSR OPEN      ! NOT ENOUGH ROOM IN
                              LINE
 990 NOMOVE   LDY $49       ! GET LINE POINTER
1000          LDA #$5B      ! ASCII FOR [
1010          STA ($FB),Y   ! PUT IN LINE
1020          LDA $40
1030          BEQ CONT6     ! NO TENS DIGIT
1040          INY
1050          STA ($FB),Y
1060 CONT6    LDA $3F
1070          BEQ CONT7     ! NO REPEATS
1080          INY
1090          STA ($FB),Y
1100 CONT7    LDA #$47      ! ASCII FOR G
1110          INY
1120          STA ($FB),Y
1130          LDA #$3E      ! ASCII FOR >
1140          INY
1150          STA ($FB),Y
1160          LDA $3D       ! CHAR
1170          INY
1180          STA ($FB),Y
1190          LDA #$5D      ! ASCII FOR ]
1200          INY
1210          STA ($FB),Y
1220          LDX $3C       ! RESET QUOTE COUNTER
1230          JMP NEXT      ! NEXT BYTE
1240 CONVERTB STA $3D
1250          LDA #$50      ! LSB OF DATA
                              ADDRESS TABLE
1260          STA $62
1270          LDA #$A3      ! MSB OF D.A.T.
1280          STA $63
1290          LDX #$51      ! COUNTER MAX NO OF
                              CHARS IN DATA
                                 TABLE
1300          LDY #$00
1310 NEXT3    JSR $81FB     ! SWITCH OF BASIC
1320          LDA ($62),Y   ! GET ASCII CHAR NO
1330          PHA           ! TEMP STORE
1340          JSR $8202     ! SWITCH IN BASIC
1350          PLA           ! RETRIEVE
```

```
1360          CMP $3D      ! IS IT THE SAME
1370          BEQ FOUND    ! YES
1380          INY          ! SKIP UNWANTED ADDRESS
1390          INY
1400          INY
1410          DEX          ! DECREASE COUNTER
1420          BPL NEXT3    ! GET NEXT ASCII NO
                             UNTIL X<0
1430          LDA $3D
1440          CMP #$1B     ! SEE IF IT USED
                             WITH CTRL KEY
1450          BCC CTRL     ! YES
1460          LDX #$0D     ! NO
1470          JMP $A437    ! ERROR OUT OF DATA
1480 CTRL     ADC #$40     ! ADD $40 TO GIVE
                             ASCII LETTER
1490          STA $A448    ! STORE IN DATA TABLE
1500          LDA #$43     ! LSB OF DATA FOR
                             [CTRL?]
1510          STA $62
1520          LDA #$A4     ! MSB OF DATA FOR
                           [CRTL?]
1530          STA $63
1540          BNE CONT8
1550 FOUND    INY
1560          JSR $81FB    ! SWITCH OUT BASIC
1570          LDA ($62),Y  ! GET LSB OF DATA
                             POSITION
1580          PHA          ! TEMP STORE
1590          INY
1600          LDA ($62),Y  ! GET MSB OF DATA
                             POSITION
1610          STA $63
1620          JSR $8202    ! SWITCH IN BASIC
1630          PLA
1640          STA $62
1650 CONT8    LDY #$00
1660          JSR $81FB
1670          LDA ($62),Y  ! GET NO OF DATA CHARS
1680          STA $C1
1690          JSR $8202
1700          LDA $C1
1710          CLC
1720          ADC #$04     ! FOR BRACKETS AND
                             REPEATS
1730          TAX
```

```
1740          LDA $40
1750          BNE CONT9
1760          DEX          ! NO TENS IN REPEATS
1770          LDA $3F
1780          BNE CONT9
1790          DEX          ! NO REPEATS AT ALL
1800 CONT9    CPX $3E      ! DO WE REQUIRE A
                             MEMORY MOVE
1810          BEQ NOMOVE1  ! NO
1820          BCS OPEN1    ! MORE SPACE
1830          JSR CLOSE    ! LESS SPACE
1840          JMP NOMOVE1
1850 OPEN1    JSR OPEN
1860 NOMOVE1  LDY $49      ! GET LINE MARKER
1870          LDA #$5B     ! ASCII FOR [
1880          STA ($FB),Y
1890          LDA $3F
1900          BEQ CONTA    ! NO TENS DIGIT IN
                             REPEATS
1910          INY
1920          STA ($FB),Y  ! STORE IN PROG
1930 CONTA    LDA $40
1940          BEQ CONTB    ! NO REPEATS AT ALL
1950          INY
1960          STA ($FB),Y  ! STORE IN PROG
1970 CONTB    STY $49      ! STORE LINE MARKER
1980          LDY #$00
1990          JSR $81FB    ! SWITCH OUT BASIC
2000 DATA     INY
2010          LDA ($62),Y  ! GET DATA FROM
                                 TABLE
2020          STY $C2      ! STORE DATA MARKER
2030          LDY $49      ! GET LINE MARKER
2040          INY
2050          STA ($FB),Y  ! STORE DATA IN PROG
                             LINE
2060          STY $49      ! STORE LINE MARKER
2070          LDY $C2      ! GET DATA MARKER
2080          CPY $C1      ! HAVE WE GOT ALL DATA
2090          BNE DATA     ! NO
2100          JSR $8202    ! SWITCH IN BASIC
2110          LDY $49
2120          LDA #$5D     ! ASCII FOR ]
2130          INY
2140          STA ($FB),Y
```

```
2150                  LDX $3C       ! GET QUOTE COUNTER
2160                  JMP NEXT      ! NEXT BYTE TO PROCESS
2170 RELOAD           LDY $49       ! GET LINE MARKER
2180                  LDX $3C       ! GET QUOTE COUNTER
2190                  JMP NEXT      ! NEXT BYTE TO PROCESS
```

```
8BD7 CHECK            8889 CLOSE
8BE3 COMPARE          8BAF CONT
8BC2 CONT1            8BEB CONT2
8C15 CONT3            8C27 CONT4
8C42 CONT5            8C5E CONT6
8C65 CONT7            8CCC CONT8
8CDF CONT9            8D04 CONTA
8D0B CONTB            8C7E CONVERTB
8CAB CTRL             8D12 DATA
8CBA FOUND            8BDB INQUOTES
8B9D LINKS            8BB1 NEXT
8BF3 NEXT1            8BFD NEXT2
8C8C NEXT3            8C51 NOMOVE
8CF7 NOMOVE1          8BCA NOPI
8931 OPEN             8CF4 OPEN1
8C4E OPENUP           8D33 RELOAD
8C12 SPACE
```

LINES 10–150: These set up the routine and if necessary return control back to you through BASIC. There are no parameters included in the command to pick up as it codes the whole program. The address of the first line is taken from the start of BASIC program variables at $2B and $2C. The X register is initialized to zero and is used as a quote counter. We get the link address to the next line and if it is the end of the program we remove the return address from the stack, placed there on entering, and go back to BASIC with the program in memory coded for listing or saving.

LINES 160–390: We are going to start to look for our trigger codes – quotes or pi. We skip the line number and start to scan. If the end of the line is encountered we transfer the links to the line register and start the next line.

We check for PI (ASCII value is $FF). On finding it, we store it in a register for later use and branch further into the program. The check for the quote takes two forms. When one is found, we increase and check the X register. A two here will indicate that it is the second quote and therefore going out of the area we are interested in. It also means we go back to look for another quote.

If X is one, then the first quote has been found and we go forward to check for codes. It will fail there the result is that we return to get the next byte.

On encountering a byte other than a quote or pi, we check to see if the X register is one, indicating that we are in quotes and it will require processing.

LINES 400–470: We first store the byte. This is done as we are going to manipulate this data and possibly alter it. The original value is needed later when checking for repeat characters.

Values over $C0 (192) are reduced by $60 (96) and we are in a position to weed out characters that do not require any action. These will be values of $21 to $5F inclusive. This is the position that the first quote will end up in. These characters cause the flow to go back and get the next byte.

LINES 480–690: Our character value is stored again, as it may be different, in another register. We also store the line pointer (the Y register) and the X quote counter. The latter is stored because if pi is being changed, the X register could be zero; at other times it will always be one.

The next procedure is to see if there is more than one character of the same type consecutively in the program. The X register will be used as a counter and as it is one already, it is already initialized. The following bytes are gathered in and checked against the original value. The X register is increased until a byte of a different value is found.

The routine now splits up. Where there are two or more repeat characters, we jump ahead to CONT3 to put that number into ASCII.

Continuing along, the X value will be one but we will not print out the number one as it implied. Registers $3F and $40 are set to zero, which as we shall see shortly will hold the ASCII value for repeats.

The action taken is to check to see if the character we are coding is a space or not. We do not want to code single spaces as it would clutter the listing unnecessarily. On finding a space the flow jumps further ahead to reload the registers and go back to get the next byte. For characters not spaces, and all single characters, the routine branches forward to skip the next section.

LINES 700–770: On finding more than one of the same character we want to convert the number into ASCII format. We already have the number in the X register. To use our own conversion routine at $847F we need to set the accumulator to zero, as the high byte value. The result will be in the input buffer with a zero after the last digit. As a line of BASIC program when typed into the 64 cannot be more than 80 characters, it therefore means that the number of repeats cannot be any greater. This means that the number of ASCII digits will be two at a maximum.

We therefore pick up the first two digits from the buffer and store them in $3F and $40. If there was a single digit, that is 2 to 9 repeats, $40 will be zero.

LINES 780–820: We said at the beginning that some characters would require the use of the data tables whilst some can be coded by calculation. These few lines divide up the flow into these two areas.

We load back the value achieved in earlier calculations (lines 410–440) to the accumulator. Values of $60 and under, or $7B and over, will branch off to Conversion B, which uses data tables.

CONVERSION A

LINES 830–1230: The first task to undertake is to subtract $20 (32) from our value and store it. This is now the same value as the ASCII code of the letter of the key it shares. They will all be achieved using the shift with the key rather than the logo key.

The maximum number of characters we could insert is seven, two for the brackets, two for the number of repeats and three for the code. This number is placed into X. We check the 'repeat' digits storage for the number of numerals. A zero will indicate that there is no digit in that column. The X register will be decreased accordingly. Location $3E has the number of graphic characters to be coded and this is compared with X. From this we either open-up the program, close-up the program or leave it unchanged. The memory move routines are described in Chapter 6.

Now we are ready to insert the code in the order of:

i) The [bracket.
ii) The number of repeats if applicable.
iii) The letter G.
iv) The symbol >.
v) The letter of the key, held in location $3D.
vi) The] bracket.

Once completed, we load the quote counter back into X and jump back to get the next character to code.

CONVERSION B

LINES 1240–1640: This is where we have to use data tables to find the relevant code. This part is entered with the character value in the accumulator and is put into $3D for later use. The first table we look up is the data address table. The start address of the table is placed in locations $62 and $63. The X register has the total number of characters catered for and the Y register is used as a general pointer.

As the table is in the RAM under the BASIC ROM we have to disable that ROM, get the byte we want and then switch back the ROM. The byte is placed temporarily on the stack during the enabling of the ROM. The byte we have collected is compared for equality with our character value. Succeeding forces a branch forward. Failure means we continue the search. The first thing is to increase the Y register three times. This will skip the address of the rejected character in the data table. The Y value will be in line with the next character value. If the X register has been decreased to a value below zero, that is, $FF, then all the data address table has been checked and a match not found. There is one further chance. It could be a character which uses the CTRL key along with a letter key. These will have values no greater than $1A (26). This is checked and if it does not fall in, then an 'OUT OF DATA' error is generated and coder is exited. We think that this should never come about as we believe we have catered for all eventualities.

Supposing a CTRL value is the one found, then we add $40 (64). This simply gives the value of the letter on the key. This is stored immediately in the data table. The start address of the start of CTRL data is placed into $62 and $63.

Now back to the other characters. A match has been found in the data address table and we have arrived at line 1550. The two bytes next in line in the address table are the data address in the second table. These are placed into registers $62 and $63.

LINES 1650–2160: We have now finished with the data address table and concentrate on the data table itself. This time we only require one byte, the first byte, which will give us the number of bytes of code. To this value we add four, the brackets and the 'repeat' digits, transfer it to X and decrease it if one or both repeat digits are redundant. This final value is compared with the number of characters to be replaced to determine whether more or less space is required. This and the moves, if needed, are achieved in lines 1800 to 1850.

The insertion of data is the only thing left to do. We reload the line marker and start. The left square bracket and, if required, the repeat digits are stored first. The data insertion is slightly complicated. The line marker is stored and the Y register is re-initialized. The BASIC ROM is switched off and a byte is taken from the table. The Y register is stored in $C2 and the line marker is restored and incremented. The byte is now inserted in the program. Now the line marker is stored and the data marker placed back in Y. This is compared with the number of bytes of data in the code ($C2). If it has not reached this number, we branch back to get further bytes to insert. Once all the data has been collected and stored, the BASIC ROM is switched back in.

Finally, the right hand square bracket is inserted and we jump to get the next character to be coded, after restoring the quote counter.

LINES 2170–2190: This simply restores the line marker and quote counter, after which the routine goes back to get the next character. Single space characters, which are not coded, are sent here.

The data table – a program

After much thought, we have decided to supply the data tables for CODER in the form of a BASIC loader. This is mainly due to the fact that it is stored under the BASIC ROM which makes it hard to check and correct using a monitor. With the loader program we can put in a checksum which helps to see if you typed in the correct values.

A further item that the loader program does is to clear the area used by the KEY command (see Chapter 4) for its data. So type in the program, check it and save it.

```
10 L=41472:T=0
20 READD:IFD=-1THEN40
30 T=T+D:POKEL,D:L=L+1:GOTO20
40 IFT<>51131THENPRINT"[REV] DATA INCORR
ECT":END
50 FORL=41216TO41471:POKEL,0:NEXT
60 PRINT"[REV] DATA LOADED":END
70 DATA3,87,72,84,2,67,68
80 DATA3,82,69,86,3,72,79
90 DATA77,3,82,69,68,2,67
100 DATA82,3,71,82,78,3,66
110 DATA76,85,3,83,80,67,3
120 DATA71,62,42,3,71,62,43
130 DATA3,71,60,45,3,71,62
140 DATA45,1,126,3,71,60,42
150 DATA3,79,82,71,2,70,49
160 DATA2,70,51,2,70,53,2
170 DATA70,55,2,70,50,2,70
180 DATA52,2,70,54,2,70,56
190 DATA3,66,76,75,2,67,85
200 DATA3,79,70,70,3,67,76
210 DATA83,3,73,78,83,3,66
220 DATA82,78,5,76,32,82,69
230 DATA68,3,71,82,49,3,71
240 DATA82,50,5,76,32,71,82
250 DATA78,5,76,32,66,76,85
260 DATA3,71,82,51,3,80,85
270 DATA82,2,67,76,3,89,69
280 DATA76,3,67,89,78,5,71
290 DATA62,83,80,67,3,71,60
300 DATA75,3,71,60,73,3,71
310 DATA60,84,3,71,60,64,3
320 DATA71,60,71,3,71,60,43
330 DATA3,71,60,77,3,71,60
340 DATA92,3,71,62,92,3,71
350 DATA60,78,3,71,60,81,3
```

```
360 DATA71,60,68,3,71,60,90
370 DATA3,71,60,83,3,71,60
380 DATA80,3,71,60,65,3,71
390 DATA60,69,3,71,60,82,3
400 DATA71,60,87,3,71,60,72
410 DATA3,71,60,74,3,71,60
420 DATA76,3,71,60,89,3,71
430 DATA60,85,3,71,60,79,3
440 DATA71,62,64,3,71,60,70
450 DATA3,71,60,67,3,71,60
460 DATA88,3,71,60,86,3,71
470 DATA60,66,5,67,84,82,76
480 DATA65,5,67,84,82,76,66
490 DATA5,67,84,82,76,72,5
500 DATA67,84,82,76,73,5,67
510 DATA84,82,76,78,5,67,82
520 DATA71,62,78,5,67,82,71
530 DATA62,77,3,68,69,76,2
540 DATA80,73,255,0,0,0,0
550 DATA5,0,162,17,4,162,18
560 DATA7,162,19,11,162,28,15
570 DATA162,29,19,162,30,22,162
580 DATA31,26,162,32,30,162,96
590 DATA34,162,123,38,162,124,42
600 DATA162,125,46,162,126,50,162
610 DATA127,52,162,129,56,162,133
620 DATA60,162,134,63,162,135,66
630 DATA162,136,69,162,137,72,162
640 DATA138,75,162,139,78,162,140
650 DATA81,162,144,84,162,145,88
660 DATA162,146,91,162,147,95,162
670 DATA148,99,162,149,103,162,150
680 DATA107,162,151,113,162,152,117
690 DATA162,153,121,162,154,127,162
700 DATA155,133,162,156,137,162,157
710 DATA141,162,158,144,162,159,148
720 DATA162,160,152,162,161,158,162
730 DATA162,162,162,163,166,162,164
740 DATA170,162,165,174,162,166,178
750 DATA162,167,182,162,168,186,162
760 DATA169,190,162,170,194,162,171
770 DATA198,162,172,202,162,173,206
780 DATA162,174,210,162,175,214,162
790 DATA176,218,162,177,222,162,178
800 DATA226,162,179,230,162,180,234
810 DATA162,181,238,162,182,242,162
```

```
820 DATA183,246,162,184,250,162,185
830 DATA254,162,186,2,163,187,6
840 DATA163,188,10,163,189,14,163
850 DATA190,18,163,191,22,163,1
860 DATA26,163,2,32,163,8,38
870 DATA163,9,44,163,14,50,163
880 DATA142,56,163,141,62,163,20
890 DATA68,163,255,72,163,5,67
900 DATA84,82,76,67,-1
```

SAVING THE DATA AREA

The following listing will save the area we have used for both KEY and CODER routines. The saving of data through machine code is described in the *Programmer's Reference Guide*. The only extra coding is to switch the BASIC ROM out, so that we will save our data and not the BASIC interpreter. You could use this after setting up the function keys (see Chapter 4) so on reloading, the data is there and ready.

```
 10          LDX #$08    ! DEVICE NO (TAPE=1)
 20          LDA #$01    ! LOGICAL FILE NO
 30          LDY #$FF    ! NO SEC ADDRESS
 40          JSR $FFBA   ! SETLFS
 50          LDA #$0C    ! CHARS IN FILENAME
 60          LDX #<NAME  ! LOW ADDRESS OF NAME
 70          LDY #>NAME  ! HIGH ADDRESS
 80          JSR $FFBD   ! SETNAM
 90          LDA $01     ! SWITCH OFF BASIC
100          AND #$FE
110          STA $01
120          LDA #$00    ! STORE START ADDRESS
130          STA $FB
140          STA #$A1
150          STA $FC
160          LDX #$49    ! LOW END OF SAVE
170          LDY #$A4    ! HIGH END OF SAVE
180          LDA #$FB    ! LOCATION OF START ADD
190          JSR $FFD8   ! SAVE
200          LDA $01     ! SWITCH IN BASIC
210          ORA #$01
220          STA $01
230          RTS
240 NAME     TXT "UTILITY DATA"
```

RELOCATING THE DATA TABLES

If you relocate CODER may also want to relocate the data. Here is one suggested way. Using the BASIC loader program for the data, change the value of L in line 10 to the new data start address. The data normally starts at $A200 (41472) but the data address table starts at $A350 (41808). From this calculate the data address table new address and put its value in lines 10 and 30 of the routine below. The end of the data address table is normally $A442 (42050), so work out its new end, subtract one, and this is put in lines 210 and 240. The difference between the old address and the new address should be put in lines 80 and 120. The routine below is for a new table at a higher address; for one lower, change the addition to subtraction and set the carry instead of clearing it.

```
 10          LDA #$50      ! START OF DATA
                             ADDRESS TABLE
 20          STA $14
 30          LDA #$A3
 40          STA $15
 50 NEXT     LDY #$01      ! POINTER
 60          LDA ($14),Y   ! LOW ADD IN TABLE
 70          CLC
 80          ADC #$60      ! ADD LOW DIFFERENCE
 90          STA ($14),Y
100          INY
110          LDA ($14),Y   ! HIGH ADD IN TABLE
120          ADC #$25      ! ADD HIGH DIFFERENCE
130          STA ($14),Y
140          CLC
150          LDA $14       ! UPDATE TABLE ADDRESS
160          ADC #$03
170          STA $14
180          LDA $15
190          ADC #$00
200          STA $15
210          CMP #$A4      ! END OF DATA
                             ADDRESS TABLE
220          BNE NEXT      ! NO
230          LDA $14
240          CMP #$43
250          BNE NEXT      ! NO
260          RTS
```

Old

COMMAND SYNTAX
OLD

There are no parameters with this command.

There are four ways to 'lose' a BASIC program. The first way is by switching off and then there is absolutely no way of recovering it. Another two ways are by doing a system cold start or a reset. This is as if you have just switched on but retaining data held by the RAMS. This can be achieved by typing SYS64738 or by a reset button, if you have fitted one. The final way to lose a program is by issuing the BASIC command NEW.

To lose a program the operating system of the 64 sets the first two bytes of the BASIC program area to zero. This would normally be $0801 and $0802 (2049 and 2050) and would be the link address in the first line of a BASIC program. This means that as far as BASIC is concerned no program is present as it would encounter the zeros straight away.

Now as long as no further lines of BASIC are typed in, we can reverse the process, but will lose all the variables. The way it is done is made clear in the description of the coding.

ASSEMBLY LISTING

```
  9 *=$8415
 10          LDA #$FF
 20          LDY #$01
 30          STA ($2B),Y  ! PUT ANY LINK IN
                            1ST LINE
 40          JSR $A533    ! RECHAIN LINES
 50          LDA $22
 60          CLC
 70          ADC #$02
 80          STA $2D      ! SET END OF PROG
                            ADDRESS -LOW
 90          LDA $23
100          ADC #$00     ! IN CASE CARRY WAS
                            SET IN 70
110          STA $2E      ! SET END OF PROG
                            ADDRESS -HIGH
120          JMP $A660    ! PERFORM CLR
```

LINES 10–40: If we change the first two bytes from zero, BASIC will no longer think it is at the end of the program. We put $FF in those, and get the address from the start of BASIC variables in $2B and $2C. Now a call to the ROM routine RECHAIN LINES will achieve two things. First, it will correctly set the link address in the first line, and secondly, we will be able to set the end of program variables.

LINES 50–120: Locations $22 and $23 are set to the beginning of the two zero bytes, which mark the end of the program, when the RECHAIN routine is finished. By adding two to those, we have the end of program and can set the respective registers, $2D and $2E.

The final thing is a jump to the CLR routine. This will set all the variable addresses to coincide with the recovered program. The BASIC program is now restored to its original state.

Dump

COMMAND SYNTAX
DUMP
There are no parameters with this command. It will also only operate in direct mode. If used within a program it will just skip out of the command. Hitting STOP will break out of DUMP and allows direct editing; typing CONT will resume at the break point in the BASIC program. Holding down any other key, apart from shift, will halt the routine until it is released.

The action of DUMP is identical to the BASIC subroutine in Chapter 5 except that as the routine is in machine code it does not add to the simple BASIC variables. The logic closely follows the BASIC routine. Output may again be directed to a printer by an OPEN and CMD sequence. The major departure is in the use of one or two ROM routines to carry out the mathematical conversions and convert the number to an ASCII string to be printed.

ASSEMBLY LISTING

```
  9 *=$8E52
 10 !NOTE    REAL ASC      / ASC OR 0
 20 !      STRING ASC      / ASC+128 OR 128
 30 !     INTEGER ASC+128 / ASC+128 OR 128
 40 !    FUNCTION ASC+128 / ASC OR 0
 50 !
 60 !
 70 !TEST FOR DIRECT MODE
 80 !
 90             LDA $9D      !MSGFLG
100             CMP #$80     !DIRECT ???
110             BEQ DIRECT
120             RTS          !PROGRAM MODE SO ABANDON
130 !
140 !NOTE CURRENT VARIABLE IN FILE NAME POINTER
```

```
150 !
160 DIRECT      LDA $2D      !VARTAB
170             STA $BB
180             LDA $2E      !VARTAB+1
190             STA $BC
200 !
210 !RETURN TO HERE TO SEE IF ALL DONE
220 START       LDA $BC
230             CMP $30
240             BNE CONT     !MORE VARIABLES
250             LDA $BB
260             CMP $2F
270             BNE CONT
280             RTS          !DONE ALL OR WERE
                              NONE TO START
290 !
300 !FIND VARIABLE TYPE
310 !
320 CONT        LDY #$00
330             LDA ($BB),Y  !FIRST CHAR OF NAME
340             CMP #$80
350             BCS INTFN    !INTEGER OR FUNCTION
360 !
370 !STRINGS AND REAL
375 !
380             JSR $FFD2    !OUTPUT ASCII CHAR
390             INY
400             LDA ($BB),Y  !GET SECOND CHAR
410             CMP #$7F
420             BCS STRING
430 !
440 !REAL VARIABLE
450 !
460             JSR $FFD2    !OUTPUT SECOND ASCII
                              CHAR
470             JSR UPDATE   !PRINT '=' AND
                              UPDATE POINTER
480             LDA $BB      !POINT TO VARIABLE
                              FOR MEMORY MOVE
490             LDY $BC      !TO FPACC#1
500             JSR $BBA2    !GO DO IT
510             JSR $BDDD    !FPACC#1 TO STRING
                              AT $0100
520             JSR $AB1E    !PRINT IT
530             LDA #$FF
540             BNE NEXT     !SKIP TO NEXT VARIABLE
```

```
550 !
560 !STRING VARIABLE
570 !
580 STRING      AND #$7F    !MAKE ASCII
590             JSR $FFD2
600             LDA #$24    !'$'
610             JSR $FFD2
620             JSR UPDATE
630             LDA #$22
640             JSR $FFD2
650             LDY #$00
660             LDA ($BB),Y
670             TAX         !LENGTH OF STRING IN X
680             BEQ QUOTE   !NULL STRING
690             INY
700             LDA ($BB),Y
710             STA $22     !LSB LOCATION
720             INY
730             LDA ($BB),Y
740             STA $23     !MSB LOCATION
750             LDY #$00
760 CHAR        LDA ($22),Y
770             JSR $FFD2
780             INY
790             DEX
800             BNE CHAR
810 QUOTE       LDA #$22
820             JSR $FFD2
830             BEQ NEXT
840             BNE NEXT
850 HALFSTART   BCS START
860 !
870 !INTEGER AND FUNCTIONS
880 !
890 INTFN       AND #$7F    !CONVERT TO ASCII
900             JSR $FFD2
910             INY
920             LDA ($BB),Y
930             CMP #$7F
940             BCS INT     !INTEGER IF SECOND
                             CHAR>128
945 !
950 !FUNCTION DEFINITION
960 !
970             JSR $FFD2
980             JSR UPDATE
```

```
 990              LDA #$46     !'F'  PRINT FN
1000              JSR $FFD2
1010              LDA #$4E     !'N'
1020              JSR $FFD2
1030              BNE NEXT
1040 !
1050 !INTEGER VARIABLE
1060 !
1070 INT          AND #$7F     !CONVERT TO ASCII
1080              JSR $FFD2
1090              LDA #$25     !'%'
1100              JSR $FFD2
1110              JSR UPDATE
1120              LDY #$00
1130              LDA ($BB),Y  !MSB
1140              STA $62      !FPACC#1
1150              INY
1160              LDA ($BB),Y  !LSB
1170              STA $63
1180              LDX #$90
1190              JSR $BC44    !CONVERT TWO BYTE
                                INT TO REAL
1200              JSR $BDDD    !FPACC#1 TO STRING
                                AT $0100
1210              JSR $AB1E    !PRINT IT
1220 !
1230 !SEE IF KEY HIT AND UPDATE POINTERS
1240 !
1250 NEXT         LDA #$0D     !CARRIAGE RETURN
1260              JSR $FFD2
1270              CLC
1280              LDA $BB
1290              ADC #$05
1300              STA $BB
1310              BCC WAIT
1320              INC $BC
1330 WAIT         JSR $FFE4    !STOP
1340              JSR $FFE1
1350              BNE NOT
1360              RTS
1370 NOT          LDA $CB      !CURRENT KEY
1380              CMP #$40     !NO KEY=64
1390              BNE WAIT     !CYCLE WHILE KEY
                                HELD DOWN
1400              SEC
1410              BCS HALFSTART
```

```
!
!PRINT '=' AND SET POINTER IN $BB/BC
 TO START OF VARIABLE
!
UPDATE       LDA #$3D     !'='
             JSR $FFD2
             CLC
             LDA $BB
             ADC #$02
             STA $BB
             BCC RETURN
             INC $BC
RETURN       RTS
```

```
8EBC CHAR          8E6E CONT
8E59 DIRECT        8ECE HALF
8EEE INT           8ED0 INTFN
8F11 NEXT          8F2A NOT
8EC5 QUOTE         8F43 RETURN
8E61 START         8E97 STRING
8F33 UPDATE        8F21 WAIT
```

The routine has been written to be easily relocatable. The only change necessary is to alter all JSR UPDATES to JSR (start address + $E1).

The ROM routines used are as follows:

CHROUT ($FFD2)
A full description of this function is given in the *Programmer's Reference Guide*, 'The KERNAL B-5'. It outputs the contents of A as an ASCII character to the screen.

STOP ($FFE1)
See 'The KERNAL B–33'. Test for the stop key. UDTIM must be called before using this routine.

UDTIM ($FFEA)
See the *Programmer's Reference Guide*, 'The KERNAL B–36'. This updates the system clock.

MEMORY TO FAC#1 ($BBA2)
This routine takes a five byte real number and moves it to the floating point accumulator #1. *En route* the sign bit of the mantissa is stripped off and the sign register FACSGN ($66) set, the exponent put at FACEXP ($61) and the mantissa of FACHO ($62–65). On entry A must hold the low and Y the high byte of the address of the bytes to be moved.

FAC#1 TO STRING ($BDDD)
Converts FAC#1 to an ASCII string stored at the bottom of the stack ($0100). On exit A holds #$00 and Y holds #$01.

PRINT STRING FROM MEMORY ($AB1E)
This routine prints successive characters starting at the memory location whose address is held in A (low) and Y (high). The routine continues until a zero terminator is found (as will be the case at the bottom stack in this application). Note A and Y already hold the start address on exiting the previous routine and need not be changed.

EVALUATE TWO BYTE SIGNED INTEGER ($BC44)
Evaluates a two byte signed integer held in FAC#1 and deposits the result in floating point form back in FAC#1. Before calling X must be set to #$90, FACHO must hold the high and FACHO+1 the low byte of the integer (remember integers are held in high/low format unlike addresses). Once in this form the same routines as for real may be used to convert to ASCII and print.

LINES 1450–1530: JSR UPDATE
This will be used by all types of variables. It will be used directly after the variable name has been printed. All this does is to print out the equals sign and increase the address registers by two, so that they will point directly to the next byte to be collected – the first of the actual variable.

LINES 90–120: These check for direct mode. If program mode is discovered, then the routine is exited.

LINES 160–190: The locations we are going to use to step through the variable area are initialized with the start of variable address, which also happens to be the end of BASIC program address. We are now ready to start.

LINES 220–280: Locations $2F and $30 are the address of the end of the variable block that we are going to DUMP. By checking the values in those with our registers, we can find out if we have completed all.

LINES 320–350: The routine is divided up here and will be further divided later. When the first byte is picked up, we check if it is an integer or a function by seeing if the value is $80 (128) or over. These are dealt with further into the routine.

LINES 380–420: The first byte we have already is printed – the first letter of the variable name. The next byte is collected and this will distinguish between real and string variables.

LINES 460–540: REAL VARIABLES: Again we print what we have in A, making the whole variable name output. A call is now made to JSR UPDATE. With the address of the present position placed in the A and Y registers, we call three ROM routines, described at the start, to print out the variable to the screen or output device. The accumulator is loaded with $FF just so the branch following will succeed.

LINES 580–840: STRING VARIABLES: Before we print out the accumulator, we remove the negative bit, bit 7, so that it is the pure ASCII code of the variable letter. The dollar sign is printed and UPDATE is called. As there are no separaters between stored strings, we cannot use a similar approach to the one used in real variables. The first thing printed is the start quotes. The length of the string is the byte after the name and this is placed in the X register. Now we gather in the address of where the string is stored. From this we can print out the characters directly, decreasing X each time, until the counter is zero. Finally, the closing quotes are output. One of the following two branches must succeed so we can continue.

LINES 890–940: Integers and functions start here. In these lines we distinguish between them and act accordingly. By stripping off the negative bit we can print out the first character of the name, and do so. The second byte is loaded and this will tell us what type to deal with. A value of $80 (128) or over will signify integer.

LINES 970–1030: FUNCTION VARIABLE: We cannot print any value for the function, so after the second name character is printed, UPDATE is called, and then we just print the letters 'F' and 'N' and branch off.

LINES 1070–1210: INTEGER VARIABLES: After printing the second character of the name the integer sign of '%' is output. Once more UPDATE is visited. The next two bytes in the variable area are the integer value and these are transferred to FAC #1. With the X register set to normalize the result, we call a ROM routine at $BC44. This will convert the integer value to a real number. We then convert to ASCII and print the result with the routine described at the beginning.

LINES 1250–1410: After dealing with any of the four types of variables, the flow is directed to this part of the routine. The return character is printed so that the next variable is printed on a new line. Each variable takes seven bytes of memory and as we added two bytes to our address registers in UPDATE, we only need to add five more to get to the start of the next variable in line.

The STOP key is now tested for and if the negative flag is set then DUMP is ended, as STOP was pressed. By examining $CB we can see if any key is being pressed. As long as a key is held down we loop around here and then check for STOP.

To continue with DUMP we set the carry flag and we use a Branch with Carry Set to line 850 where the same happens, going further back to proceed dumping variables.

IMPROVEMENTS?

Obvious improvements are as for the BASIC subroutine. If you had trouble extending the BASIC subroutine to handle arrays, you haven't tried anything yet! Most dump routines (wisely) do not handle subscripted variables (probably because it is considered too difficult).

Trace and Troff

COMMAND SYNTAX
TRACE and TROFF

Speed Control

'0' reset single-step
'1' a relative delay of 2^0
'2' a relative delay of 2^1
'3' a relative delay of 2^2
'4' a relative delay of 2^3
'5' a relative delay of 2^4
'6' a relative delay of 2^5
'7' a relative delay of 2^6
'8' a relative delay of 2^7
'9' a relative delay of 2^8

The space bar operates the single-stepping.

The delay is in addition to the normal time taken by BASIC to move to a new line and execute the common trace code. The delay may be changed at any time by hitting the appropriate key. It is, however, not possible to break into program execution in single step. If you wish to do this, hit a number other than 0 first.

TRACE is a diagnostic aid which provides useful information on the path taken through a BASIC program. In this particular version the previous and the current line numbers are displayed in reverse video at the top right of the screen.

We considered it far more important to allow the user to be able to vary the speed of the trace and have single-stepping capability. When called for the first time the default will be to single stepping and thereafter at each run it will continue at the last set speed. After being disabled with TROFF it will, on being enabled, revert back to single stepping. In single step mode program execution halts until the space bar or a speed change key is pressed. The keys for speed change are given above.

ASSEMBLY LISTING

```
  9 *=8D3A
100 !TRACE ENABLE
110 !
120 !
130 ENABLE       LDA $9D    !MSGFLG CHECK FOR DIRECT
```

```
140            BEQ PMODE !$00=PROG $80=DIRECT
150            SEI       !OK-NOT IN PROG MODE
                          SO DISABLE
160            LDA #$FF  !INTERRUPT AND SET
                          SINGLE STEP
170            STA SSTEP !TO $FF FOR SINGLE STEP
180            LDA #$FF  !DO SAME FOR TRACE FLAG
190            STA TRFLAG!
200            LDX $0308 !IGONE GET LOW BYTE
                          OF TOKEN
210            STX IGONE !DISPATCH AND STORE AT
                          TEMP REG
220            LDA NLVL  !NEW LINE VECTOR LOW WHICH
230            STA $0308 !POINTS TO TRACE
240            LDX $0309 !SAME FOR HIGH
250            STX IGONE+1
260            LDA NLVH
270            STA $0309
280 PMODE      CLI       !RESET INTERRUPT
290            RTS       !AND RETURN TO BASIC
300 !
310 !TROFF = TRACE DISABLE
320 !
330 DISABLE    SEI        !REVERSE ENABLE PROCESS
340            LDA $9D    !CHECKING IN DIRECT MODE
350            BEQ PMODE  !
360            LDA #$00   !DISABLE TRACE FLAG
370            STA TRFLAG !RESET TOKEN DISPATCH
380            LDA IGONE  !TO VALUES AT THE TIME
390            STA $0308  !OF CALLING
400            LDA IGONE+1
410            STA $0309
420            CLI
430            RTS        !BACK TO BASIC WITH
                           TRACE OFF
440 !
450 !PERFORM TRACE  IGONE POINTS HERE
460 !
470 TRACE      STA AREG   !IF BASIC IS TO RESUME
480            PHP        !THEN WE MUST SAVE A,X,Y
490            STX XREG   !AND STATUS FLAGS
500            STY YREG   !TO RESTORE THEM
                           ON CONTINUING
510 !
520 !ONLY PROCEED WITH TRACE IF A PROGRAM RUNNING
530 !
```

```
540              LDA $9D
550              BEQ RUNNING
560 !
570 !RESTORE ENTRY VALUES BEFORE CONTINUING
580 !
590 BASIC        LDA AREG    !REVERSE ENTRY PROCESS
600              LDY YREG    !TO ALLOW PROG
                              TO CONTINUE
610              LDX XREG    !UNCORRUPTED
620              PLP         !DON'T FORGET FLAGS!!!
630              JMP (IGONE)!CONTINUE AT
                              TOKEN DISPATCH
640 !
650 !PROGRAM RUNNING SO CHECK IF TRACE ENABLED
660 !FROM TRACE FLAG =$FF???
670 !
680 RUNNING      LDA TRFLAG
690              BEQ BASIC   !TRACE OFF SO RESTORE
                              AND CONT
700 !
710 !TRACE IS ON SO UPDATE DISPLAY
720 !
730              SEC         !READ CURSOR POSITION
740              JSR $FFF0   !AND SAVE BY CALLING PLOT
750              STX ROW     !WITH CARRY SET
760              STY COL
770              CLC         !SET CURSOR POSITION
780              LDX #$00    !TO ROW 0 COLUMN 24
790              LDY #$18
800              JSR $FFF0
810              LDX #$0F
820 SPACE        LDA #$20    !CLEAR PREVIOUS NUMBERS
830              JSR $FFD2
840              DEX
850              BNE SPACE
860              CLC         !SET BACK TO ROW 0 COL 24
870              LDX #$00
880              LDY #$18
890              JSR $FFF0
900              LDA #$12    !TURN ON REVERSE VIDEO
910              JSR $FFD2
920              LDA OLHIGH !LOW BYTE PREVIOUS LINE
930              LDX OLLOW   !HIGH
940              JSR $BDCD   !PRINT LINE NUMBER
950              LDA #$92    !REVERSE OFF
960              JSR $FFD2
```

```
 970            LDA #$20    !BIT OF SPACE
 980            JSR $FFD2
 990            LDA #$12    !REPEAT FOR CURRENT LINE
1000            JSR $FFD2   !GETTING ITS VALUES FROM
1010            LDA $3A     !CURLIN  LOW BYTE
1020            STA OLHIGH  !NOW BECOMES OLD LINE
1030            LDX $39     !CURLIN+1
1040            STX OLLOW
1050            JSR $BDCD
1060            LDA #$92
1070            JSR $FFD2
1080            CLC         !PREPARE TO RESET CURSOR
1085 !
1090 !IGNORE THIS BIT AS ONLY TO ALLOW BRANCH TO WORK
1095 !
1100 BASIC1     BCS BASIC   !HALFWAY BRANCH TO BASIC
1110            LDX ROW     !CONTINUE RESET CURSOR
1120            LDY COL
1130            JSR $FFF0   !RESTORE CURSOR POSITION
1140 !
1150 !CHECK FOR ANY KEYS PRESSED
1160 !
1170            JSR $FFE4   !GETIN
1180 CHCHAR     BEQ SINGLE  !NOTHING IN K/B BUFFER
1190            CMP #$2F    !KEY<0???
1200            BCC SINGLE  !YES THEN OF NO INTEREST
1210            CMP #$3A    !KEY>9???
1220            BCS SINGLE  !YES - NO INTEREST
1230            SBC #$30    !BETWEEN 0 AND 9 SO -$30
1240            BNE CHDELAY!1-9
1250 !
1260 !0 PRESSED SO RESET SINGLE STEP
1270 !
1280            LDA #$FF
1290            STA SSTEP
1300            BNE SINGLE  !NO NEED TO CALC DELAY
1310 !
1320 !CALCULATE DELAY AS POWERS OF 2
1330 !
1340 CHDELAY    TAX         !PUT 1-9 IN X
1350            SEC         !1 IN CARRY
1360            LDA #$00
1370 ROLL       ROL A       !MOVE CARRY BIT X TIMES
1380            DEX         !TO SET KEY-2 BIT
1390            BNE ROLL    !TO GIVE DOUBLING DELAY
1400            STA COUNT   !STORE IT TO USE AS TIMER
```

```
1410             LDA #$00    !DISABLE SINGLE STEP
1420             STA SSTEP
1430             BEQ DELAY   !ALWAYS TAKEN
1440 !
1450 !SINGLE STEP PAUSE
1460 !
1470 SINGLE      LDX SSTEP   !IS IT ON
1480             BEQ DELAY   !IF NOT GO TO DELAY
1490             CMP #$20    !SPACE HIT ORIGINALLY???
1500             BEQ BASIC2  !YES THEN PERFORM LINE
1510 SSLOOP      JSR $FFE4   !GETIN WAIT FOR A CHAR
1520             BEQ SSLOOP  !AND KEEP WAITING
1530             CMP #$20    !SPACE???
1540             BEQ BASIC2  !YES THEN SKIP DELAY
1550             BNE CHCHAR  !NO - WAS IT A
                              SPEED CHANGE
1560 !
1570 !TIMER COUNTDOWN FOR DELAY
1580 !
1590 DELAY       LDX COUNT   !DO COUNT LOTS
1600 DLOOP1      LDY #$FF    !OF 256'S
1610 DLOOP2      DEY
1620             BNE DLOOP2
1630             DEX
1640             BNE DLOOP1
1650 !
1660 !GUARANTEED BRANCH TO HALFWAY BACK TO
      DISPATCHING LINE 1100
1670 !
1680 BASIC2      SEC         !ENSURES BRANCH BASIC1
1690             BEQ BASIC1  !Z FLAG SET ALWAYS SET HERE
1700 !
1710 !RESERVE TEMPORARY STORES AND FLAGS SET
      TO DEFAULTS
1720 !
1730 TRFLAG      BYT $00     !TRACE FLAG OFF
1740 SSTEP       BYT $FF     !SINGLE STEP ON
1750 COUNT       BYT $00     !NO DELAY
1760 AREG        BYT $00     !A ON ENTERING FROM BASIC
1770 XREG        BYT $00     !X "
1780 YREG        BYT $00     !Y "
1790 COL         BYT $00     !CURSOR WHILE
                              LINES PRINTED
1800 ROW         BYT $00
1810 OLLOW       BYT $00     !PREVIOUS LINE LOW
1820 OLHIGH      BYT $00     !              HIGH
```

```
1830 IGONE        BYT $00,$00!STORE FOR ORIG VECTOR
1840              END        !A JMP TO HERE


8E49 AREG           8D87 BASIC
8DEF BASIC1         8E43 BASIC2
8DFD CHCHAR         8E12 CHDELAY
8E4C COL            8E48 COUNT
8E38 DELAY          8D61 DISABLE
8E3B DLOOP1         8E3D DLOOP2
8D3A ENABLE         8E50 IGONE
8E4F OLHIGH         8E4E OLLOW
8D5F PMODE          8E16 ROLL
8E4D ROW            8D94 RUNNING
8E24 SINGLE         8D8F SPACE
8E2D SSLOOP         8E47 SSTEP
8D97 TRACE          8E46 TRFLAG
8E4A XREG           8E4B YREG
```

The ROM routines used are as follows:

CHROUT ($FFD2)
As DUMP (see page 190).

GETIN ($FFE4)
See *Programmer's Reference Guide*, 'The KERNAL function B–11'. This removes one character from the current input device (usually the keyboard buffer) and returns its ASCII value in A. Zero is returned if none waiting.

PLOT ($FFF0)
See *Programmer's Reference Guide*, 'The KERNAL function B–19'. Reads the cursor position with the carry set and positions the cursor when the carry is clear. Misleadingly, X is used for the row and Y for the column.

PRINT LINE NUMBER ($BDCD)
Useful little routine, this one, and well worth noting. Not only can it be used for line numbers, but also for a two byte unsigned integer ($0000 to $FFFF). Before calling it, X must hold the low and A the high byte. It also strips off the traditional leading and trailing spaces before printing.

HANDLE NEW LINE ($A734)
This routine is vectored by the page 3 vector IGONE ($0308) and $A7E4 is the 64's default setting. This is BASIC's token DISPATCH routine and is covered in great detail in Chapter 3. When used with the UTILITY, IGONE has been modified and hence the reason why IGONE has been first read and

stored. Doing it this way means the routine will work with or without the UTILITY. IGONE is called to tokenize each new line and is thus the ideal point at which to patch our trace.

LINE 130–290: TRACE ENABLE: These set up TRACE ready for when you RUN a BASIC program. A scan is made for direct operation only, and only if it is direct do we continue. During this initialization the interrupt will be disabled. The default speed is single step and its value is stored in the appropriate location at the end of the routine. The original value of IGONE, BASIC Character Dispatch Vector (see Chapter 6), are stored for safe-keeping and the start of TRACE replaces them. After clearing the interrupt, we return you to BASIC until the RUN command is issued.

LINES 330–290: TROFF – Trace Disabled: The reverse of the TRACE set up procedure.

LINES 470–690: The BASIC dispatch is used each time a BASIC command is issued whether in direct or program mode. This means that the routine can be called when not required. To avoid this, we check for program mode, after preserving the processing registers, as in the set up. If still in direct mode, we restore the registers and jump to the normal DISPATCH routine. A final check is made before operating TRACE to ensure it is enabled by looking at the TRFLAG at the end of the routine.

LINES 730–1130: Display and updating are the purpose of these lines. We print at the top of the screen so as not to disrupt your display. We locate and save the current cursor position before setting it to the start of our print, top row and column 24, and clear the area we use by printing 15 spaces to the end of the line. After turning on reverse video, we gather in the values of the previous line number and visit the ROM routine to print it. To distinguish between the line numbers, we put a space between them, after turning off the reverse video. We now repeat the operation for the current line number. At the same time as getting the current line number, we store its value in our previous line store. As TRACE is called before every BASIC command is initiated, then when more than one command is on a line the previous and the present line numbers will be identical.

The instructions in line 1080 and 1100 are little tricks. Clearing the carry will ensure that the branch will fail. The branch is there for a later instruction when it will save a JUMP command.

Finally, we restore the original cursor position.

LINES 1170–1430: Is a speed change required? To find out, we use the KERNAL routine GETIN. If there is no character or it is not between the ASCII values of $30 and $3A, we branch out of this section. On remaining here, we subtract $30 from the character value to convert from ASCII to a real value between 0 and 9.

The zero value signifies single-step mode is required again, so its flag is set to $FF.

The remainder have to be acted upon in order to gain our power of two figures for delay purposes. The value we have is to be used as a counter and so is transferred to the X register. To start with, we set the carry and initialize the accumulator to zero. The accumulator is rotated right X times. The first time, the bit set in the carry is transferred to bit zero of the accumulator. The carry from this time on will be unset, except for the last time when nine is the value of X and that will not worry us. Every consecutive rotate will shift the set bit further to the right and so increase the power of two of the value. When nine is raised to a power the accumulator will end up with a value of zero, but when the delay is explained, lines 1590–1090, you will see that this is in effect 256, that is, 2^8. Finally, we disable the single-step mode, if set, by storing zero in its flag.

LINES 1470–1530: SINGLE-STEP MODE: To check if it is in operation, we test its flag. We not only look for the space character, which operates single-stepping, but also for others by branching back to lines 1170 to 1430. This means to continue in single-step the space bar or a numeral key will in fact cause the program to continue, the latter ones ending single-step at the same time.

LINES 1590–1620: DELAY: This consists of a loop within a loop. The inner loop is completed the number of times calculated earlier, thereby giving variable time delays. When speed nine has been selected COUNT is zero, the inner loop is carried out, and the COUNT is decreased before checking for zero. When zero is decreased it becomes $FF (255) so the check will fail until it is decreased to zero once more. This therefore operates the inner loop 256 times.

Finally, we set the carry and branch back, as the accumulator will be zero, to line 1100 where the Branch with Carry Set will send it further back to jump in to the normal IGONE routine to carry out the BASIC command.

IMPROVEMENTS?

It is possible to modify the trace to list not only the previous and current line, but also to highlight the current statement being executed.

To list a line we can use the LIST routine in ROM which starts at $A69C. There are two major problems if we try to use it. The first is that LIST uses a number of zero page locations also used during a run. The second is that on completion of LIST a warm boot of BASIC is carried out. (Try putting LIST in a program and running it.) We can overcome the first by copying zero page to elsewhere in RAM before calling LIST. The second requires that on return from LIST, we must re-enter our TRACE routine at the next instruction after performing LIST which must restore zero page. To do this we must read and store the warm start vector IMAIN ($0302) and set it to the next instruction after LIST is called. TROFF must, of course, reset this vector to its original value.

To highlight the statement within a listed line places even greater demands on our ingenuity and would require the TRACE routine to be rewritten to use CHRGET which has purposely been avoided (because of DOS 5.1). If it were to use CHRGET, the line could be re-listed each time with a marker character printed at the current byte held in TXTPTR ($7A/$7B) through the use of the PRINT tokens link.

Both additions seem of little point as we can use the STOP key followed by LIST line number(s). We have not even attempted to incorporate either possibility.

Numeric conversions

In the world of the Commodore 64 we come across three main numbering systems: that of decimal, to the base of 10, hexadecimal, to the base of 16, and binary which is to the base of 2 (octal, to the base eight is less common).

The binary number system is used because there are only two numerals used: 0 and 1. This matches the type of electronics used in the computer world, digital electronics, which has only two states, either on or off. These two positions are known as logical states. Logic 1 is on and obviously logic 0 is off. These, as you can see, fit well with the binary system.

The hexadecimal system was introduced because although binary matches the electronics, it is unwieldy and is not so easy to recognize in everyday form. Hexadecimal is easier to remember, using only two digits to the binary eight, and therefore faster to type in. Hexadecimal is nearly always entered in groups of two for example, $FF.

Decimal is used in our everyday life and is therefore used in BASIC. One of its disadvantages is that numbers have varying amounts of digits. For instance, in numbers up to 255 there are one, two or three digits whereas with hex there is always two.

Some BASICS give you the option to enter numbers in forms other than decimal. The Commodore 64 BASIC does not. We are not going to rectify this but are giving you four conversion routines. These are converting decimal and binary, and decimal and hex.

TEN – *Decimal to hexadecimal conversion*
COMMAND SYNTAX
TEN decimal number [,decimal number,....]

The maximum decimal number that can be converted is 65535 and then only positive numbers can be converted. Multiple conversions can be done if they are separated by commas. The result will be a four digit hex number.

ASSEMBLY LISTING

```
  9 *=$84A0
 10 START       BEQ SYNTAX
 20             BCS SYNTAX
 30             JSR $81F5    ! GET PARAMETER
 40             LDA #$20
 50             JSR $FFD2    ! PRINT A SPACE
                               ON SCREEN
 60             LDY #$02
 70 NEXT        JSR HEX      ! CONVERT A BYTE TO HEX
 80             DEY
 90             BNE NEXT     ! TWO CONVERTS FOR
                               EACH BYTE
100             LDY $14
110             STY $15      ! PLACE LOW BYTE
                               FOR CONVERSION
120             LDY #$02
130 NEXT1       JSR HEX
140             DEY
150             BNE NEXT1
160             JSR $0079    ! GET LAST BYTE
                               OF BUFFER AGAIN
170             CMP #$2C     ! COMMA FOR MORE NUMBERS
180             BNE EXIT     ! NO MORE TO CONVERT
190             JSR $FFD2    ! PRINT COMMA AS SPACER
200             JSR $0073
210             JMP START
220 EXIT        RTS
230 HEX         LDX #$04     ! SET COUNTER
240             LDA #$00     ! INITIALISE ACC
250 AGAIN       ASL $15
260             ROL A
270             DEX
280             BNE AGAIN
290             CMP #$0A     ! IS IT 10 OR MORE
300             BCC DIGIT+1  ! ASCII ADDITION
                               FOR NUMBER
310             CLC
320             ADC #$37     ! FOR LETTER
330 DIGIT       BIT $3069
340             JSR $FFD2    ! PRINT RESULT
350             RTS
```

```
360 SYNTAX        JMP $AF08
```

```
84D5 AGAIN          84E2 DIGIT
84D0 EXIT           84D1 HEX
84AE NEXT           84BA NEXT1
84A0 START          84E9 SYNTAX
```

LINES 30–50: Here we pick up the first decimal number to convert. The high byte will be in $15 and the low, in $14. We then print a space on the screen so the first digit is a character away from the border or the last PRINTed statement.

LINES 60–90: Each part of the hexadecimal number will have two characters. As we will convert our decimal in two stages, the high byte first then the low, each will require two entries to the conversion routine. We therefore set a counter to two, in this case the Y register. After going to the conversion routine we decrease the counter. If it is zero then we have done it twice, if not we go back again.

LINES 100–110: We have now converted the high byte. As the conversion subroutine uses the high byte register in the transposition, we transfer the low byte of the decimal to that register.

LINES 120–150: This is the same as lines 40–70 but for the low byte.

LINES 160–220: The GET PARAMETERS has already picked up the byte after the last decimal digit. Here we get that byte again by a call to CHRGOT. We want to find out if more than one conversion is required. The syntax of the command is for a comma as a separator, so we check for that. If the check succeeds, we print the comma to the screen and go back to convert the number. On failing the check, it is back to BASIC *via* the RTS.

THE CONVERSION ROUTINE
We use this routine four times for every decimal number in the command, twice for both the high and low bytes. We enter with the byte in location $15. The hexadecimal number for a byte consists of two digits, one for the upper four bits and the other for the lower four. As we print on the screen from left to right, we print from the most significant hex digit and therefore want the high bits of the decimal number first.

Hex uses numerals 0 to 9 and letters A to F. Unfortunately, these do not follow in sequence in the ASCII table, as other characters lie between 9 and A. We therefore have to test for this when converting to ASCII for printing to the screen.

LINES 230–240: The X register is initialized to 4 as a counter for taking off the required bits for each hex digit. The accumulator is used to gather in the bits so is initialized to zero before we start.

LINES 250–280: This is the main part of the conversion. We use the instruction ASL to move all the bits of the decimal byte one place to the left. The most significant bit (bit 7) is moved to the carry flag. The least significant bit (bit 0) is filled with zero (although that does not worry us). We need the bit we put into the carry back in the accumulator. This is achieved by the command ROL. This moves the accumulator bit one place to the left, filling bit 0 with the carry value, which we have just set (or unset). Bit 7 of the accumulator goes to the carry and again it is of no use to us here.

Now the counter is decreased and checked to see if we have done the bit shifting four times. We have now taken the four high bits of $15 (the decimal byte) and put them in the same order in the accumulator but in the low bit positions.

LINES 290–350: The answer in the accumulator is now converted to ASCII form and printed to the screen. If it is less than $0A, it is a number so we add $30 to it. Greater than 10 means it is a letter, so we have to add $37, giving letters from A to F.

HEX – *Converting a hexadecimal number to decimal*
COMMAND SYNTAX
HEX hex number [,hex number,....]

The hex number can be of two or four digits. More conversions can be added if separated by commas. The normal '$' sign which preceeds hex numbers must not be used.

A four digit hex number can be split very conveniently into two parts. The two left digits are the high byte whilst the right are the low byte. Where a two digit conversion is required, we treat it as a low byte number. The two digits can be further split in that one represents the high nybble and the other the low nybble (a nibble is half a byte or four bits).

To do the conversion we collect two hex digits at a time and convert them to a one byte answer.

ASSEMBLY LISTING

```
  9 *=$8537
 10 START        STA $63
 20              JSR $0073     ! GET NEXT BYTE
 30              STA $62
 40              JSR DECIMAL
 50              PHA           ! PUT HIGH ANS ON STACK
 60              JSR $0073
 70              BEQ LOWPRINT! ONLY TWO BYTE HEX NO
 80              CMP #$2C      ! IS IT A COMMA
 90              BEQ COMMA     ! YES & ONLY 2 BYTE HEX
100              STA $63
```

```
110              JSR $0073
120              STA $62
130              JSR DECIMAL
140              TAY          ! PUT LOW ANS IN Y
150              PLA          ! GET HIGH ANS IN ACC
160              JSR PRINT    ! PRINT ANSWER
170              JSR $0073    ! GET NEXT BYTE
180              CMP #$2C     ! IS IT A COMMA?
190              BEQ COMMA1   ! YES
200              RTS
210 LOWPRINT     PLA          ! GET HIGH ANS
220              TAY          ! PUT IT IN LOW ANS REG
230              LDA #$00     ! SET HIGH ANS TO ZERO
240              JSR PRINT    ! PRINT ANSWER
250              RTS
260 COMMA        PLA          ! GET HIGH ANS
270              TAY          ! PUT IT IN LOW ANS REG
280              LDA #$00     ! SET HIGH ANS TO ZERO
290              JSR PRINT    ! PRNT ANSWER
300 COMMA1       LDA #$2C     ! ASCII FOR COMMA
310              JSR $FFD2    ! PRINT IT AS SPACER
320              JSR $0073    ! GET NEXT BYTE
330              JMP START
340 DECIMAL      LDY #$01     ! COUNTER
350 AGAIN        LDA $0062,Y  ! GET LOW CHAR
360              CMP #$30     ! IS IT A NUMBER
370              BCC SYNTAX   ! NOT NUMBER OR
                                    LETTER
380              CMP #$47     ! IS IT LETTER > F?
390              BCS SYNTAX
400              CMP #$3A     ! IS IT A NUMBER?
410              BCC DIGIT    ! YES
420              CMP #$41     ! IS IT A LETTER?
430              BCC SYNTAX   ! NO
440              SBC #$37     ! CONVERT ASCII LETTER
                                INTO REAL NUMBER
450              BNE NEXT     ! ENFORCED
460 DIGIT        SEC
470              SBC #$30     ! CONVERT ASCII NUMBER
                                INTO REAL NUMBER
480 NEXT         STA $0014,Y
490              DEY
500              BPL AGAIN    ! NEXT CHARACTER
510              LDY #$04     ! COUNTER
520 NEXT1        ASL $15      ! PUT HIGH ANS IN
                                HIGH BITS
```

```
530                 DEY
540                 BNE NEXT1
550                 LDA $14
560                 ORA $15      ! JOIN BOTH TOGETHER
570                 RTS
580 SYNTAX          JMP $AF08
590 PRINT           JSR $B391
600                 JSR $B401    ! CONVERT TO
                                   POSITIVE REAL NUMBER
610                 JSR $AD8D
620                 JSR $BDDD
630                 JSR $B487
640                 JMP $AB21
```

```
857E AGAIN          856A COMMA
8571 COMMA1         857C DECIMAL
8595 DIGIT          8562 LOWPRINT
8598 NEXT           85A0 NEXT1
85AD PRINT          8537 START
85AA SYNTAX
```

LINES 10–50: The routine is entered with the first digit and is stored. Calling CHRGET, we get the next byte and again store it. The decimal conversion routine is visited (this is described later), and the result comes back into the accumulator which we place on the stack.

LINES 60–90: The next byte of the command is now collected and two checks made, first to see if it is the end of the command and secondly for a comma. If the first succeeds we go off and print what we have already collected, but as a low byte answer. If it is a comma, we again print but will return to do further conversions.

LINES 100–130: This is a repeat of lines 10–50 except the result in the accumulator is put in the Y register instead of the stack.

LINES 140–200: The result of the four digit conversion is printed to the screen here. The Y register has the low byte and the high byte is pulled off the stack into A. The print routine described at the end is now called. The conversion is complete and we now check to see if further conversions are required by getting the next byte. If a comma is not present, the routine is ended.

LINES 210–250: The low byte answer is printed here. The byte is pulled off the stack and placed in the Y register and the accumulator set to zero. After printing, the HEX routine is left.

LINES 260–290: This is the same as lines 210–250 but instead of leaving, we continue as we know there was a comma present when we arrived here.

LINES 300–330: A comma is printed on the screen to separate the answers. The first byte of the next conversion is gathered and we go back to the beginning to start converting again.

LINES 340–570: THE DECIMAL CONVERSION: The two bytes will be in locations $62 and $63. They will hold the ASCII values of either a numeral or a letter between A and F. SYNTAX ERRORS are given if they fall outside these limits.

Taking each location in turn, we determine what it is and deduct from it $37 for a letter of $30 for a numeral, the value ending up between $00 and $0F (0 to 15). These are placed in registers $14 and $15. We now have to combine these into one number. Address $15 will have the high nybble but in the wrong bit positions. To get them into the upper four bits we shift the bits left four times. To join the two together, the byte in $14 is copied to the accumulator and is ORed with location $15. With the final result in the accumulator, we exit the subroutine.

LINES 590–640: PRINT RESULT TO SCREEN: Six subroutines are called here where the result of numeric calculations are converted to a string of ASCII characters and printed to the screen which except for one are all ROM routines. The one exception is a subroutine in the DEEK routine (see Chapter 8). For convenience, we reproduce it below. We enter this PRINT routine with numeric data, the Y register holding the low and the accumulator the high byte.

ROUTINE $B391 – FIX TO FLOAT

This sets the data flag in $0D to zero signifying numeric data. The number we wish to convert is placed in FAC#1 registers $62, meaning that numbers over 32768 ($80) will be output as negative numbers.

ROUTINE $8401 – CONVERT TO POSITIVE

```
10          LDA $66
20          BPL EXIT
30          LDY #>DATA
40          LDA #<DATA
50          JSR $BA8C
60          JSR $B86A
70 EXIT     RTS
80 DATA     BYT $91,$00,$00,$00,$00
```

We check register $66 of the FAC#1 to see if it is negative. If so we load FAC#2 with zero and set for no exponent. This is done through the ROM routine $BA8C, entering with the data start address in A and Y.

Now by adding the two FACs together we will end up with a result in FAC#1 which is a real whole number; $B86A will achieve this.

ROUTINE $AD8A – CHECK

This just checks that the data is numeric, otherwise a 'TYPE MISMATCH' error is given.

ROUTINE $BDDD – FAC#1 TO STRING

This converts the contents of FAC#1 to an ASCII string and places it at the bottom of the stack. The Y and A registers will hold this address when the routine is finished.

ROUTINE $B487 – SET UP STRING

This sets various registers so that the PRINT routine knows where to print from and how long the string is.

ROUTINE $AB21 – PRINT

This takes the data from the bottom of the stack and prints it to the screen. We jumped to this routine, so when it is ended, the processor will be directed back to the position calling this whole subroutine.

This routine, being a separate routine, is therefore capable of being used by other commands as in the MEM command.

TWO – *Decimal to binary conversion*
COMMAND SYNTAX
TWO decimal number [,decimal number,....]

The maximum decimal number which can be converted is 65535 and must be positive. Multiple conversions can be done if they are separated by commas. The result will be two eight digit binary numbers separated by a space, unless the number is 255 or less, when only one binary result will be shown.

All we need to do is to test each bit and print a zero or a one according to its state.

ASSEMBLY LISTING

```
 9 *=$84EC
10 START       BEQ SYNTAX
20             BCS SYNTAX
30             JSR $81F5    ! GET PARAMETER
40             LDA #$20
50             JSR $FFD2    ! PRINT SPACE
60             LDA $15
70             BEQ LSB
80             LDX #$08     ! COUNTER
90 NEXT        ASL $15
```

```
100          BCS SET+1     ! TO PRINT A 1
110          LDA #$30      ! TO PRINT A 0
120 SET      BIT $31A9     ! SET+1 IS LDA #$31
130          JSR $FFD2
140          DEX
150          BNE NEXT
160          LDA #$20
170          JSR $FFD2     ! PRINT A SPACE
180 LSB      LDX #$08
190 NEXT1    ASL $14
200          BCS SETA+1
210          LDA #$30
220 SETA     BIT $31A9     ! SETA+1 IS LDA #$31
230          JSR $FFD2
240          DEX
250          BNE NEXT1
260          JSR $0079     ! GET LAST BYTE OFF
                             BUFFER AGAIN
270          CMP #$2C      ! IS IT A COMMA?
280          BNE EXIT      ! NO COMMA
290          JSR $FFD2     ! PRINT COMMA AS SPACER
300          JSR $0073     ! GET NEXT BYTE
310          JMP START
320 EXIT     RTS
330 SYNTAX   JMP $AF08
```

```
8533 EXIT           8512 LSB
84FE NEXT           8514 NEXT1
8504 SET            851A SETA
84EC START          8534 SYNTAX
```

LINES 10–20: On entering the first byte after the keyword is in the A register and by testing the carry and zero flags, we can check if a numeral is first.

LINES 30–150: We gather in the decimal number to convert and also print a space for presentation to move it away from the last item printed or from the border. Taking the high byte first, we check it for zero, if it is a zero we branch and just do the low byte. The x register is set to eight as there are eight bits to a byte, and we shall use it as a counter. We shift the bits of the high byte one place to the left. The leftmost bit comes off and goes to the carry flag. If the carry flag is zero, we therefore print ASCII $30 which is zero and $31 when the carry is set. This is repeated a further seven times for all the remaining bits.

LINES 160–250: This is a repeat of the above except for the low byte.

LINES 270–320: By calling CHRGOT, we test for a comma or if the end of the command has been reached. When the former is found, it is printed to the screen and we gather in the next byte before going to carry out the next conversion.

BIN – *Binary to decimal conversion*
COMMAND SYNTAX
BIN 8 bit binary number [,8 bit binary number,....]

Here we will convert an eight bit binary number to decimal. We supply a value that would be a high byte and one that is the low byte. For example, if you demanded 11111111 was converted, the answer would come out as 255. Only eight bit numbers are accepted but more conversions can be done by separating the items with commas.

This is essentially the reverse of the previous command. The 1s and 0s you type in will be picked up in their ASCII form. These have their rightmost bits corresponding to their numeric value, so by taking those we can build up a single byte number.

ASSEMBLY LISTING

```
  9 *=$85BF
 10              LDA $7A
 20              BNE LOW
 30              DEC $7B     ! DECREASE CHARGET
                               POINTER BY ONE
 40 LOW          DEC $7A
 50 ANOTHER      LDX #$08    ! COUNTER
 60 NEXT         JSR $0073   ! GET BYTE
 70              BCC NUMBER  ! IF NUMBER BRANCH
 80 SYNTAX       JMP $AF08
 90 NUMBER       CMP #$32    ! IS > ASCII FOR 2
100              BCS SYNTAX  ! YES
110              ROR A       ! GET BIT 0
120              ROL $14     ! PUT IN $14 MOVING
                               1 LEFT EVERY TIME
130              DEX
140              BNE NEXT    ! DONE IT 8 TIMES?
150              LDY $14     ! PUT ANS IN Y REG
160              LDA #$00    ! NO HIGH ANS
170              JSR $85AD   ! PRINT ANS-HEX ROUTINE
180              LDA #$2F
190              JSR $FFD2   ! PRINT SLASH TO DIVIDE
                               LOW ANS & HIGH ANS
200              LDA $14     ! NOW PUT ANS IN
                               HIGH ANS REG
210              LDY #$00    ! SET LOW ANS REG TO 0
```

```
220              JSR $85AD    ! PRINT ANS-HEX ROUTINE
230              JSR $0073    ! GET NEXT BYTE OF
                                INPUT BUFFER
240              CMP #$2C     ! IS IT A COMMA?
250              BNE EXIT     ! NO
260              JSR $FFD2    ! PRINT COMMA AS SPACER
270              JMP ANOTHER
280 EXIT         RTS
```

```
85C7 ANOTHER        85FB EXIT
85C5 LOW            85C9 NEXT
85D1 NUMBER         85CE SYNTAX
```

LINES 10–40: This decreases CHRGET address by one.

LINES 50–140: We want to pick up eight binary digits so the X register is used as a counter. After we pick up a digit, via CHRGET, we check for a number and also if it is two or greater, in ASCII. By rotating the accumulator right, we take off bit 0 and it ends up in the carry flag. Then if we rotate location $14 left, we move all its bits one to the left and put the carry flag state into the lowest bit. If we do this eight times, address $14 will have a number equivalent to the 1s and 0s you typed in.

LINES 150–220: The PRINT routine in the HEX command is used twice. For this the low byte needs to be in Y and the high in A. The value we want to print is in $14 and by changing the register we load it into we can print out the states we want. A slash is printed as a separator by a visit to the KERNAL routine at $FFD2.

LINES 230–280: Having done one conversion, we take a look to see if more are required. A comma is printed if so and then we go back to do it all again.

8 Enhancing the resident BASIC

Introduction

In the previous chapter the commands were of a toolkit nature. In this chapter they are mainly improvements to the standard 64 commands, GOTO, GOSUB, RESTORE, PRINT, INPUT, GET, PEEK, POKE and LOAD. To that end, we are supplying CGOTO, CGOSUB, PROC, DPROC, EPROC, RESET, WRITE, ENTER, INKEY$, DEEK, DOKE and CHAIN. There are five commands which have no 64 BASIC equivalent, but which we hope will enhance your BASIC programming. They are POP, PLOT, COLOUR, LOMEM and HIMEM. The final command given is that of QUIT and exists the UTILITY.

In comparison with the toolkit commands these are shorter, but no less useful to you. No doubt you can think of existing commands which could be enhanced and even more to add. This chapter should help you on your way.

CGOTO and CGOSUB

COMMAND SYNTAX

CGOTO variable or line number
CGOSUB variable or line number

A limitation of Commodore BASIC is that it does not permit the use of calculated destinations with GOTO and GOSUB. We thought it would be nice to be able to use variables and mathematical expressions, for example A*20. To allow this, we have come up with two commands CGOTO and CGOSUB, the C standing for calculated or computed – whichever you prefer.

CGOTO is the easiest of the two, not that either is complicated. The routine requires only two instructions. In the BASIC ROM routine of GOTO the first instruction gets the line number and is therefore the only thing we have missed out. So after getting the variable we only have to jump to that part of GOTO.

CGOSUB is a bit longer in that we have to copy the ROM routine for GOSUB and change the address for calling the GOTO routine as we want to use our 'computed' routine.

ASSEMBLY LISTING

```
  9 *=$8FAF
 10 CGOTO        JSR $81F5     ! GET PARAMETER
 20              JMP $A8A3     ! GO TO ROM GOTO
 30 ! CGOSUB ROUTINE
 40              LDA #$03
 50              JSR $A3FB     ! CHECK FOR ROOM ON
                                      STACK
 60              LDA $7B       ! SAVE CHRGET
                                 ADDRESS ON STACK
 70              PHA
 80              LDA $7A
 90              PHA
100              LDA $3A       ! SAVE CURRENT LINE
                                 NUMBER ON STACK
110              PHA
120              LDA $3B
130              PHA
140              LDA #$8D      ! MARKER FOR GOSUB
                                 ON STACK
150              PHA
160              JSR $0079     ! GET LAST BYTE AGAIN
170              JSR CGOTO
180              JMP $A7AE     ! BASIC TO EXECUTE
                                 PROGRAM FURTHER


8FAF CGOTO
```

LINES 10–20: CGOTO: We use GET PARAMETERS simply to find the destination line number. It will evaluate any expression, and jump to the normal GOTO routine, one instruction in.

LINES 40–60: CGOSUB: These lines check if there is enough room on the stack to store the routine's information and a buffer amount for other routines. To do this the value in the accumulator is doubled and added to $3E (62 dec). This is then compared with the stack pointer. If the stack pointer is the lesser value, then an 'OUT OF MEMORY' error is generated. In our case, the stack pointer would have to be less than $44 (it starts at $FF).

LINES 70–130: There are two markers we will require when the subroutine is finished. These are the present byte's address from the CHRGET routine and the line number we must return to later. The stack is used to store this information.

LINES 140–150: Another value is put on the stack. This is used by the RETURN routine to check a GOSUB has been implemented.

LINES 160–170: Now we can go to our destination. To do this, we get the last byte collected by CHRGET again and go to our new computed GOTO routine.

LINE 180: Once the GOTO routine has been completed, in which the CHRGET has been given new values, we return to the normal flow of BASIC and the program is continued at its new address.

Procedures

COMMAND SYNTAX

PROC title – call a procedure
DPROC title – define a procedure
EPROC – the end of a procedure

The title is not required within quotes. If it is then the quotes will be considered as part of the name. Spaces also cannot be used as CHRGET ignores them (a space in the DPROC title will not be matched in the PROC title). On the other hand, a space in the PROC title will have no bearing on the matching. A colon is the only other character which cannot be used in a title.

You can have as many PROCs on a line as you want, but the DPROC must be on a line of its own. Everything following the DPROC to the end of the line is included as the title. EPROC follows exactly as RETURN.

64 BASIC cannot be described as a structured language. GOTOs and GOSUBs do not form the basis of a structured language.

To start you on the road to 'structured programming', we are introducing PROCEDURES. We have nothing profound to offer but by giving you an introduction we hope you will be able to take it further (IF . . THEN . . ELSE WHILE . . . WEND DO . . . UNTIL etc . .)

The form of procedures we have written are really no more than GOSUBs with names or variables (CGOSUB). In fact, they will be slower, but not that you would notice, than GOSUBs because of the extra code required. So what advantage will they have? Well, they can be relocated anywhere in the program without changing any directive line numbers; adding procedures from one program to another, especially if they include procedures within them, is a simple matter. If GOTO was also given the same treatment, all directive line numbers could vanish. Renumbering a program would be a simple matter of changing the line numbers rather than going through the whole program and correcting destinations. A further function they perform, and one that should not be overlooked, is that they make a program easier to follow. For instance, to see PROC PERFORM-WAIT is clearer than GOSUB 2000.

Quite simply, all we do when finding a PROC is to search through for the token DPROC and then compare the named titles. On finding it, we perform a GOSUB. The UTILITY interpreter will action the command DPROC as a REM if it encounters one. The third command of the trio is EPROC and is just a RETURN by a different name. We actually go to the RETURN routine. After the listing and description we suggest some improvements.

ASSEMBLY LISTING

```
  9 *=$8FD2
 10                LDX #$00
 20                DEC $7A       ! DECREASE CHRGET ADD
 30                BCS COLLECT
 40                DEC $7B
 50 COLLECT        JSR $0073     ! GET PROC NAME
 60                BEQ NAMEEND   ! FOUND 0 OR :
 70                STA $0200,X   ! STORE IN INPUT BUFFER
 80                INX
 90                BNE COLLECT   ! ENFORCED
100 NAMEEND        LDA #$00      ! 0 AT END
110                STA $0200,X
120                LDA #$03
130                JSR $A3FB     ! CHECK STACK DEPTH
140                LDA $7B       ! SAVE CHRGET ADDRESS
150                PHA
160                LDA $7A
170                PHA
180                LDA $3A       ! SAVE CURRENT LINE NO
190                PHA
200                LDA $39
210                PHA
220                LDA #$8D      ! STACKMARKER FOR GOSUB
230                PHA
240                LDA $2B       ! GET AND STORE
250                STA $FB       ! ADDRESS OF 1ST
260                LDA $2C       ! PROGRAM LINE
270                STA $FC
280 NEXT           LDY #$00
290                LDA ($FB),Y   ! GET AND STORE LINKS
300                STA $FD
310                INY
320                LDA ($FB),Y
330                STA $FE
340                BNE CONT      ! NOT END OF PROGRAM
350                LDX #$11      ! DPROC NOT FOUND
```

```
360              JMP $A437     ! UNDEF'D STATEMENT
                                 ERROR
370 CONT         LDY #$04      ! SKIP LINKS AND
                                 LINE NUMBERS
380              LDA ($FB),Y
390              CMP #$E1      ! TOKEN OF DPROC
400              BEQ PROC      ! FOUND A DPROC
410 LINE         LDA $FD       ! PUT LINKS TO
420              STA $FB       ! LINE REGISTERS
430              LDA $FE
440              STA $FC
450              BNE NEXT      ! ENFORCED $FE
                                 CHECKED EARLIER
460 PROC         LDX #$FF
470 CHECK        INX
480              INY
490              LDA ($FB),Y   ! GET PROC TITLE
500              BEQ ZERO      ! END OF LINE
510              CMP $0200,X   ! COMPARE FOR MATCH
520              BEQ CHECK     ! MATCH FOUND
530              BNE LINE      ! NO MATCH FIND NEXT
                                 DPROC
540 ZERO         CMP $0200     ! COMPARE LAST BYTE
550              BNE LINE      ! NO MATCH
560              SEC           ! PREPARE FOR SUBTRACT
570              LDA $FB       ! ADDRESS OF PROCEDURE
580              SBC #$01      ! DECREASE TO END OF
                                 LAST LINE
590              STA $7A       ! PUT AS CHRGET ADD
600              LDA $FC
610              SBC #$00      ! IN CASE PAGE CROSSED
620              STA $7B
630              JMP $A7AE     ! BASIC TO CONT PROG


902C CHECK          8FDA COLLECT
9018 CONT           9020 LINE
8FE5 NAMEEND        9006 NEXT
902A PROC           9039 ZERO
```

LINES 10–110: Using CHRGET, after decreasing it by one, we take the PROC title and store it at the start of the input buffer and tag a zero byte on the end for checking purposes.

LINES 120–230: Same as in CGOSUB, saving relevant details for RETURN, or in this case EPROC.

LINES 240–360: After collecting the start address of the program, we search through the program. This part gets the links and checks for the end of the program. The UNDEF'D STATEMENT error is given if the latter occurs without finding the procedure.

LINES 370–450: When inputting a BASIC line, any spaces between a line number and the first character are removed during tokenizing, (LIST inserts a space for clarity). This means that the first token in a line is the fourth byte (starting at zero, remember), so we check only this byte for the DPROC token of $E1. If not found, the link address is placed into the line registers and the hunt continues.

LINES 460–550: Having found a DPROC token, we have to compare each character separately and as long as they match we continue checking. When we reach the end of the DPROC program line, we check the input buffer for a matching zero. When all checks succeed, we have found the required procedure.

LINES 560–630: Knowing our destination, we take the start address of the DPROC line and reduce it by one, the end of the preceding line, and store it as the CHRGET address. Finally, we jump to BASIC to evaluate.

IMPROVEMENTS?

One of the first questions that came to mind was: how could we speed up the search for the procedures? One solution to the problem is to form a table in RAM holding the start address where you first check to see if the PROC name is in it. This would involve setting aside an area of RAM: under ROM would be an ideal place, for such a table. Two characters would then have to be chosen: one to mark the end of an entry and the other, the end of the table. The make-up of the table could consist of the PROC name and its start address. How could the table be filled? When the interpreter finds the keyword PROC, the table would be searched for a match. If no match is found, then a program search, like our routine, could be instigated. On finding the DPROC with the correct name, it would be added to the table in case of another call.

There are, however, problems. Let us assume that a program containing PROCs has been RUN. This would mean the table has names and addresses within it. Before running it again, you add an extra line before the procedure. The line with the DPROC now has a different address from that in the table. Another action giving rise to the same problem is when you load another program. It may have a PROC with the same name as the previous program. Again the table may have another address. A further problem may arise in that more PROCs will be added making the table longer and longer.

Two solutions to this problem spring to mind; there are probably others. The first is to write a new RUN command, for example, PRUN, where one of its actions would be to place the end of table marker back to the beginning – thereby effectively clearing the table. The other is to have a command that can be actioned to do just this and only this. It could be initiated in direct mode or from within a program.

A further improvement would be to allow parameters to be passed using variables which are local to the procedure. These variables could be used elsewhere in the program without losing their original values. We would envisage the PROC command to include, in say, brackets, the values or other variables to be used, for example, PROC INPUT(2,4,6) or PROC INPUT(2,A,6). The DPROC, on the other hand, will define the variables to be used. For example, DPROC INPUT(X,Y,Z). These variables may be used elsewhere but in the procedure they will start with values given in the PROC command.

What would have to be done is that when arriving at the procedure a search is carried out for the variables X,Y,Z in the normal variable area. If they are found, their current values would have to be transferred to a keeping area, and the new values set up. If the variable is not present, then it will have to be created. The default value of a numeric variable is zero and this will also have to be stored in the keeping area. For strings, the addresses will have to be stored. The EPROC would have to reverse the situation and restore the original values.

The process would be the same if you wanted to incorporate GLOBAL and LOCAL commands to a BASIC extension.

Our last improvement, although we are sure you could think of more, is to allow the names of procedures to include keywords. This would be relatively simple in that all you have to do is to slightly alter the CRUNCH token routine (see Chapter 3). In that routine when it comes across a REM, for instance, it skips further crunching. All you have to do is to insert further coding to check for PROC and DPROC and follow the same path as REM.

POP – RETURN without returning

COMMAND SYNTAX

POP

There are no parameters to this command. If it is activated without a GOSUB, CGOSUB or a PROC being used, a 'RETURN WITHOUT GOSUB' error will be generated.

There are many occasions when one requires to leave a subroutine but not go back to the calling position. This is, of course, possible but leaves values on the stack; do it too often and the stack will become full and an 'OUT OF MEMORY' error will occur

POP will remove from the stack the data placed there by the last

GOSUB, or equivalent. This will mean, for example, if you called a subroutine which in turn called another and whilst in the second you called POP, then you will go back to main program when the RETURN is met, not the first subroutine. A GOTO after a POP will mean you can go anywhere from a subroutine without any worries about the stack. POP will also discharge any FOR/NEXT loops. If you happen to be in one at the time, watch out.

ASSEMBLY LISTING

```
  9 *=$8631
 10          LDA #$FF     ! RESET FOR/NEXT PTR
 20          STA $4A
 30          JSR $A38A    ! SEARCH STACK FOR
                            GOSUB & FOR ACTIVITY
 40          TXS          ! X REGISTER TO
                            STACK POINTER
 50          CMP #$8D     ! GOSUB MARKER ON STACK
 60          BEQ CONT
 70          LDX #$16     ! ERROR-RETURN
                            WITHOUT GOSUB
 80          JMP $A437
 90 CONT     INX          ! REMOVE GOSUB ACTIVITY
100          INX          ! INCREASE X AS IT WILL
                            BE STACK POINTER
110          INX
120          INX
130          INX
140          TXS          ! REPLACE STACK POINTER
150          RTS


8642 CONT
```

LINES 10–20: By loading $4A with $FF, we effectively cancel any FOR/NEXT loop.

LINES 30–80: This is the ROM routine used by RETURN to look for the GOSUB marker on the stack. On return the stack pointer is in the X register and the accumulator has a value from the stack. If this is $8D, the RETURN marker was present. An error will be produced if anything else is found.

LINES 90–150: To remove the GOSUB activity, we take the stack pointer, which is still in the X register, and increase it by five and then use it as the new stack pointer.

RESET – Selective data restorer

COMMAND SYNTAX

RESET [line number]

When no line number is present it behaves as the standard command RESTORE. With the parameter it will set the DATA pointer to the specified line.

DATA statements are extremely useful commands, and with sprites on the 64 you no doubt use them frequently. The snag comes when you want to use the same DATA statements again. RESTORE only allows you to set the pointer back to the first DATA statement, actually the start of the program, which has the same effect. To use statements again that are not at the beginning, dummy READs have to be employed to get to the desired position. To allow you greater flexibility, RESET will allow you to specify the line the next READ will start at, whether before or ahead of the present position. The RESTORE command takes the start of program address, subtracts one from it, and places it in the DATA pointer registers. RESET will take the line number, find its address, decrease it and set the pointers. Although the routine will give an error if the prescribed line number is not present, we do not check to see if it is a DATA line. This does not matter to BASIC as it will find the next DATA line when READ is sanctioned.

ASSEMBLY LISTING

```
  9 *=$8611
 10             BNE RESET     ! CHECK FOR PARAMETER
 20             JMP $A81D     ! NO - RESTORE IN
                                BASIC ROM
 30 RESET       JSR $81F5     ! GET LINE NUMBER
 40             JSR $A613     ! FIND BASIC LINE
 50             BCS CONT      ! FOUND LINE
 60             LDX #$15      ! ILLEGAL DIRECT ERROR
 70             JMP $A437     ! ROM ERROR ROUTINE
 80 CONT        SEC           ! PREPARE FOR SUBTRACT
 90             LDA $5F       ! LOW ADD OF LINE
100             SBC #$01      ! DECREASE BY ONE
110             STA $41       ! DATA REG IN PAGE 0
120             LDA $60       ! HIGH ADD OF LINE
130             SBC #$00      ! IF PAGE IS CROSSED
140             STA $42       ! DATA REGISTER
150             RTS

8623 CONT           8616 RESET
```

LINES 10–20: If no line number, we go straight to RESTORE in the ROM.

LINES 30–70: When the line number is picked up it will be in the right location for a line search, which is immediately carried out. The carry flag set will indicate the line was found. The error given for not finding it is 'ILLEGAL DIRECT'.

LINES 80–150: Locations $5F and $60 will have the address of the line, and from these we subtract one and store them in the DATA pointers.

DEEK and DOKE – BASIC Addressing

It should be clear by now that addresses are stored in two locations as a low and a high byte. In the resident BASIC the only way to find the address, held in locations, is to do two PEEKS, one for each location, then multiply the high byte by 256 and add in the low byte, giving the address in decimal. To set up an address, the reverse process is used, but using POKE in place of PEEK.

The UTILITY commands are therefore obvious. We wish to read or set an address, or pair of locations, with one command. These are DEEK and DOKE.

DEEK – seeing double

COMMAND SYNTAX

DEEK (low byte location)

This returns the 16 bit value held in the given address and the following one. The rules for PEEK apply in that it must be an argument to a command (that is, a function).

ASSEMBLY LISTING

```
  9 *=$83D7
 10             LDA $15
 20             PHA
 30             LDA $14
 40             PHA
 50             JSR $AEFA     ! CHECK FOR (
 60             JSR $81F5     ! GET PARAMETER
 70             JSR $AEF7     ! CHECK FOR )
 80             LDY #$01
 90             LDA ($14),Y
100             TAX
110             DEY
120             LDA ($14),Y
```

```
130              TAY
140              PLA
150              STA $14
160              PLA
170              STA $15
180              TXA
190              JSR $B391     ! A & Y IN FAC#1
200              JSR CONVERT
210              PLA
220              PLA
230              JMP $AD8D     ! EXIT
240 CONVERT      LDA $66       ! CHECK FOR SIGN
250              BPL EXIT
260              LDY #>DATA
270              LDA #<DATA
280              JSR $BA8C     ! CONSTANT TO FAC#2
290              JSR $B86A     ! ADD FAC#2 TO FAC#1
300 EXIT         RTS
310 DATA         BYT $91,$00,$00,$00,$00


8401 CONVERT        8410 DATA
840F EXIT
```

LINES 10–40: DEEK, being a BASIC function rather than a command, is used in conjunction with other keywords. You have no doubt gathered that keywords use a fair number of zero page locations, notably $14, $15 and the FACs. We cannot take DEEK in isolation and also have to get its parameters. The latter means that we will use $14 and $15. We do not require to use these on exit, so we take the precaution of saving the current contents on the stack for the time being.

LINES 50–70: These not only get the parameters, but also check for the convention of them being in brackets. The ROM routines used will give the error if they are not present.

LINES 80–180: Using the address, now in $14 and $15, we read the contents and store them into the registers A and Y. We can restore the original values to $14 and $15, and do so.

LINES 190–230: The calling routine will expect the result in the FAC#1 and this is all the routine at $B391 does. Unfortunately it also stores it as a signed integer. To correct this, CONVERT is called. Having done so, we pull off the return address, but we do not require to go back to the evaluation routine, and jump back to ROM. In ROM it will check that it is numeric data and will return to the calling routine, say PRINT or DOKE.

LINES 240–300: CONVERT: If the number requires converting it will have a negative sign. With no sign we exit the routine. Failing that we load a constant into FAC#2 which when added to the contents of FAC#1, will change it to an unsigned number. For a more detailed explanation, see MEM in Chapter 7.

DOKE – complete addressing

COMMAND SYNTAX

DOKE low byte address, value

This turns the value to a two byte number and stores it in the given address and the following one. The value has a maximum of 65535 ($FFFF).

ASSEMBLY LISTING

```
  9 *=$83B3
 10            JSR $81F5     ! GET PARAMETER
 20            JSR $AEFD     ! CHECK FOR COMMA
 30            JSR $AD8A     ! GET NEXT PARAMETER
 40            LDA $66
 50            BMI ERROR
 60            CMP #$91
 70            BCS ERROR
 80            JSR $BC9B     ! PUT PARAMS IN A & Y
 90            LDA $65
100            LDX $64
110            LDY #$00
120            STA ($14),Y
130            INY
140            TXA
150            STA ($14),Y
160            RTS
170 ERROR      JMP $B248     ! ILLEGAL QUANTITY
                               ERROR
```

```
83D4 ERROR
```

LINES 10–30: The first routine called is the familiar one. The address will be in $14 and $15 after this call. The next routine checks for a comma. The last one collects the data for storing and puts it in the FAC#1.

LINES 40–80: These check for the legality of the data and set up the FAC#1 so we can take our values off.

LINES 80–160: After getting the data from FAC#1, we store them in the addresses specified.

OUTPUT – Setting the cursor

In the standard BASIC, the normal way to set the print position is to use the cursor control codes. Although they do the job, they are not ideal. You have to remember where the current position is, they take up bytes in the program, and TAB and SPC are not much better.

A far better way would be to specify the X and Y coordinates directly. To do this, three commands are included here, PLOT, WRITE and ENTER. The first will only set the cursor, the second will set the cursor and print what you want, whilst the last is INPUT with cursor positioning. The major command as far as routines go is PLOT. It really is a subroutine for the other two.

PLOT – cursor setting

COMMAND SYNTAX

PLOT (X,Y)

The maximum value of X is 39 and of Y is 24. The top left hand corner of the screen, cursor home position, has the coordinates of 0,0.

ASSEMBLY LISTING

```
  9 *=$8381
 10                JSR $AEFA    ! CHECK FOR (
 20                JSR $81F5    ! GET PARAMETER
 30                LDA $14
 40                CMP #$28     ! IS X > 40
 50                BCC COMMA
 60 ILLEGAL        JSR $B248    ! ILLEGAL QUANTITY
                                  ERROR
 70 COMMA          PHA
 80                JSR $AEFD    ! CHECK FOR COMMA
 90                JSR $81F5    ! GET PARAMETER
100                LDX $14
110                CPX #$19     ! IS Y > 25
120                BCS ILLEGAL
130                PLA          ! RETRIEVE 1ST PARAM
140                TAY
150                CLC          ! SET NOT READ CO-OR
160                JSR $FFF0
```

```
170             JSR $0073     ! GET NEXT BYTE
180             RTS


8390 COMMA          838D ILLEGAL
```

LINES 10–70: The left hand bracket is checked for and the X coordinate of the command is picked up. It is then checked that it does not exceed the limit. On an occasion that it does, we go to a ROM routine whose sole purpose is to generate the 'ILLEGAL QUANTITY ERROR'. We require to use location $14 again so the X coordinate is put on the stack for a while.

LINES 80–120: After checking for the separating comma, we get the Y coordinate. This, too, is checked for legality.

LINES 130–180: the Y coordinate was picked up in the X register and now we retrieve the X coordinate and place it in the Y register. This is the opposite to what is logical but the KERNAL routine calls for them in that order. Before calling the routine, we clear the carry flag. (If we set the carry we would read the cursor position.)

After setting the cursor we get the next byte. This is for WRITE and ENTER so that they are set up for their respective ROM routines.

WRITE and ENTER

COMMAND SYNTAX

WRITE (X,Y)[string or variable]
ENTER (X,Y)[string],variable

The coordinates take the syntax of PLOT. The remainder of the commands have the same syntax as their respective standard commands, WRITE as PRINT and ENTER as INPUT.

ASSEMBLY LISTING

```
 9 *=$83A7
10 ! WRITE COMMAND -PRINT
20              JSR $8381     ! PLOT ROUTINE
30              JMP $AAA0     ! PRINT ROUTINE IN ROM
40 ! ENTER COMMAND -INPUT
50              JSR $8381     ! PLOT ROUTINE
60              JMP $ABBF     ! INPUT ROUTINE IN ROM
```

These simply call the previous PLOT routine and then go to their normal ROM routines.

Colour

COMMAND SYNTAX

COLOUR background[,border][,text]

The latter two parameters are optional. If they are omitted it will not affect their present values. There is no error checking on values in the command. However, only the low byte of a number is used, that is, numbers up to 255, and of that only the lower four bits have effect (15 uses four bits whilst 16 uses five). The values to be used are the same as in the *Programmer's Reference Guide* or if you prefer the key number less one, with the logo key the number plus seven. Variables can be used as parameters. If no parameters are used, the background only will be changed, and that will be to black.

ASSEMBLY LISTING

```
  9 *=$8352
 10          JSR $81F5  ! GET PARAMETER
 20          LDA $14
 30          AND #$0F
 40          STA $D021  ! BACKGROUND
 50          JSR $0079  ! GET LAST BYTE AGAIN
 60          BEQ EXIT
 70          JSR $AEFD  ! CHECK COMMA
 80          JSR $81F5  ! GET PARAMETER
 90          LDA $14
100          AND #$0F
110          STA $D020  ! BORDER
120          JSR $0079  ! GET LAST BYTE AGAIN
130          BEQ EXIT
140          JSR $AEFD  ! CHECK COMMA
150          JSR $81F5  ! GET PARAMETER
160          LDA $14
170          AND #$0F
180          STA $0286  ! TEXT
190 EXIT     RTS


8380 EXIT
```

LINES 10–60: This handles the background colour. We get the first parameter of the command, and load in the low byte only. This is ANDed with $0F which will set the top four bits to zero no matter what state they were in. The result is used to set the colour. Finally, we check, by getting the last byte again, if the end of the command has been reached. If it has not, we continue.

LINES 70–130: This first checks for the comma. Then we can get the parameter and proceed as for the background, except to store the value in the border register.

LINES 140–190: This is the same as above except, of course, we set the text colour.

CHAIN – Passing variables

COMMAND SYNTAX

CHAIN ["filename"],[device]

The syntax for CHAIN follows that for LOAD except for the secondary address. No errors will be given for the inclusion of the secondary address, as the routine will overwrite it.

One of the problems of LOAD is that if you load a larger progrram, after running a smaller program, you overwrite any variables. Also LOAD, if initiated by a direct command, will perform a CLR so you will lose the variables anyway. Sometimes we wish to transfer as many of the variables as possible from one program to another, hence CHAIN.

CHAIN differs from the normal LOAD in two respects. First, it saves the data held in the variable and string areas before the load and restores it afterwards. Secondly, it automatically RUNs the program – obviously it has to be in BASIC.

CHAIN transfers the area of memory holding the variables and arrays to below the string storage. The desired program is loaded and then the data moved back down to the end of the new program. Finally all we have to do is to RUN the program.

Although a fuller, and better, explanation of the way variables are stored is given in Chapter 1, here is a reminder of areas that CHAIN cannot deal with. Defined functions are held in the program, only the pointer is in the variable area, and therefore cannot be transferred. The same applies to strings unless they are concatenated or held in arrays.

There are two listings for this command. The first is entered on the command and it will call the main CHAIN routine. Although the CHAIN routine works as designed we found that, due to the memory move routines, if there were no variables to move you ended up with a page which could contain anything. The first routine will rectify that after the main routine. This also means that CHAIN could be used as a direct command to load and run disk or tape programs.

ASSEMBLY LISTING 1

```
  9 *=$92B3
 10               LDA $32     ! END OF ARRAYS
 20               CMP $2E     ! CHECK WITH START
                                OF VARIABLES
 30               BNE ZERO+1
 40               LDA $31
 50               CMP $2D
 60               BNE ZERO+1  ! NOT THE SAME ADDRESSES
 70               LDA #$80
 80 ZERO          BIT $00A9
 90               STA $0C     ! $00 FOR VARIABLES
                                $80 FOR NONE
100               JSR $0079   ! GET LAST CHRGET BYTE
110               JMP $9080   ! PERFORM CHAIN
120               LDA $0C     ! GET FLAG
130               BPL RUN     ! VARIABLES
140               DEC $30     ! DEC ARRAY ADDS BY PAGE
150               DEC $32
160 RUN           JMP $A7AE


92D3 RUN          92C1 ZERO
```

ASSEMBLY LISTING 2

```
  9 *=$9080
 10               JSR $E1D4   ! GET LOAD
                                    PARAMETERS
 20               LDA #$00    ! ENSURE RELOCATING LOAD
 30               STA $B9
 40               JSR $B526   ! GARBAGE COLLECTION
 50               LDA $2D     ! START OF BLOCK TO MOVE
 60               STA $5F
 70               LDA $2E     ! END OF RESIDENT PROG
 80               STA $60
 90               SEC
100               LDA $31     ! END OF BLOCK
110               STA $5A
120               SBC $2F     ! CALC AREA OF ARRAYS
130               STA $FD
140               LDA $32     ! ALSO END OF ARRAYS
150               STA $5B
160               SBC $30
170               STA $FE
```

```
180            LDA $33
190            SEC         ! NEW END OF BLOCK
200            SBC #$01
210            STA $58
220            LDA $34
230            SBC #$00
240            STA $59
250            JSR $A3BF   ! PERFORM MOVE
260            LDA $37     ! SAVE END OF BASIC AREA
270            STA $41
280            LDA $38
290            STA $42
300            LDA $58     ! SAVE BEGINNING OF
                             NEW BLOCK
310            STA $FB
320            STA $37     ! SET TOP OF BASIC AREA
330            INC $59     ! RECTIFY PAGE
340            LDA $59
350            STA $FC
360            STA $38
370            LDX $2B     ! SET LOAD ADDRESS
380            LDY $2C
390            LDA #$00    ! SET FOR LOAD
400            JSR $FFD5   ! KERNAL LOAD
410            BCC STATUS  ! MAYBE GOOD LOAD
420            JMP $E0F9   ! LOAD ERROR
                             DEPENDING ON A
430 STATUS     JSR $FFB7   ! READ I/O STATUS WORD
440            AND #$BF
450            BEQ CONT    ! LOAD OK
460            LDX #$1D
470            JMP $A437   ! LOAD ERROR
480 CONT       STX $2D     ! SET END OF PROGRAM
490            STY $2E
500            STX $5F     ! SET FOR VARIABLE MOVE
510            STY $60
520            LDA $FB     ! START OF BLOCK TO MOVE
530            STA $5A
540            LDA $FC
550            STA $5B
560            SEC
570            LDA $33     ! END OF BLOCK
580            SBC #$01
590            TAY
600            LDA $34
610            SBC #$00
```

```
620          TAX
630          TYA
640          SEC          ! CALC AMOUNT TO MOVE
650          SBC $5A
660          STA $58
670          TAY          ! NO OF BYTES OF
                            INCOMPLETE PAGE
680          TXA
690          SBC $5B
700          TAX          ! NO OF PAGES TO MOVE
710          INX          ! FOR EASIER CHECKING
720          TYA
730          BEQ PAGE     ! NO SEPARATE BYTES
740          LDA $5A      ! MOVE SEPARATE
                            BYTES FIRST
750          CLC
760          ADC $58
770          STA $5A
780          BCC NOINC
790          INC $58
800          CLC
810 NOINC    LDA $5F
820          ADC $58
830          STA $5F
840          BCC NOINCA
850          INC $60
860 NOINCA   TYA
870          EOR #$FF     ! 1'S COMPLEMENT
880          TAY
890          INY          ! 2'S COMPLEMENT
900          DEC $5B
910          DEC $60
920 PAGE     LDA ($5A),Y
930          STA ($5F),Y
940          INY
950          BNE PAGE
960          INC $5B
970          INC $60
980          DEX          ! POINTER FOR COMPLETION
990          BNE PAGE
1000         SEC
1010         LDA $5F
1020         STA $31      ! NEW ARRAY END
1030         SBC $FD      ! CALC ARRAY START
1040         STA $2F
1050         LDA $60
```

```
1060          STA $32
1070          SBC $FE
1080          STA $30
1090          LDA $41     ! RESET END OF BASIC
1100          STA $37
1110          LDA $42
1120          STA $38
1130          PLA
1140          PLA         ! REMOVE RETURN ADDRESS
1150          JSR $A533   ! RECHAIN LINES
1160          LDA #$00
1170          JSR $FF90   ! TURN OF KERNAL MESSAGES
1180          JSR $FFE7   ! CLALL
1190          JSR $A677   ! END OF CLR
1200          JSR $A68E   ! BACK UP TEXT POINTER
1210          JMP $92CC   ! BACK TO FIRST ROUTINE


90E3 CONT           9119 NOINC
9123 NOINCA         912C PAGE
90D7 STATUS
```

As CHAIN is just moving memory and loading, it is an amalgamation of routines previously described. Where we come across lines used elsewhere, the description will direct you there. By copying lines rather than using subroutines, we make the routine more transportable.

LISTING 1

LINES 10–110: Here we find out if there are variables to move by taking the address of the end of program away from the end of arrays address. On the result we set a flag in location $0C to $80 or $00 denoting variables. We then jump to the main CHAIN routine, LISTING 2.

LINES 120–160: Having returned from the routine, we check our flag by loading and testing the sign flag. A positive result tells us that variables were transferred and no further adjustments are required. If the result was minus, then the addresses denoting the start and end of arrays are reduced by one page, the high byte of the address less one. The final action of CHAIN is to go to ROM, where the next BASIC line is executed. The main routine does the setting up for this just before it comes back to here.

LISTING 2

LINES 10–40: We use the ROM routine to get and set up the loading parameters. To ensure that the load has no secondary address, we unset that location. The garbage collection routine at $B526 will tidy up the variable area so that it uses the least space possible.

LINES 50–250: Although the locations from which the addresses are gathered are different, these lines are discussed in Chapter 6, Memory. Moving, lines 1190 –1340 (see pages 131–136). There is, however, one extra item involved. To be able to set the start of array address, after loading, we calculate its number of bytes and store it in locations $FD and $FE.

LINES 260–360: The data has been moved and now we protect it by changing the pointer to the limit of BASIC. This value will be obtained from the move routine, locations $58 and $59, after increasing the latter by one. This is because in the move routine the high byte is decreased before checking for completion. Increasing rectifies this. The original end of BASIC pointer is stored for later use.

LINES 370–470: The loading sequence is covered in the MERGE and APPEND routine in Chapter 7, lines 550–640 (see pages 158–163).

LINES 480–990: Moving the block down is virtually identical to lines 90–550 of Memory Moving in Chapter 6.

LINES 1000–1210: With the major work done, just the clearing up remains. First we calculate the new start of arrays and set its registers. Then we restore the pointer to the end of BASIC. Six ROM routines are visited to finish off the routine. The first two rechain the lines of the BASIC program, so that the interpreter can follow them, and turn off the KERNAL messages. The call to $FFE7 closes all open files and sets the input/output channels to their default values. The following subroutine is made halfway into CLR. This will do a RESTORE, reset CONT locations, and amend the stack point. The last two routines will do the auto-run. The former sets the CHRGET address to one byte before the program starts. The last one returns us to the calling routine to finish off.

INKEY$ – A waiting GET

COMMAND SYNTAX

i INKEY$
ii INKEY$ ""
iii INKEY$ A$ – where A$ is predefined
iv INKEY$ "characters"

All commands will stop and wait for a key press. The first two will wait until any key is pressed. The latter two will wait for a key press corresponding to a character within the defined string. The ASCII value of the key press will be placed in the variable ST, and will remain there until an input-output is performed on cassette, serial or RS232.

In 64 BASIC there are two commands for receiving a user input from the keyboard: INPUT and GET. The last accepts a key press without a RETURN but will not wait for one. This entails checking the input and GOTOS until the key press you want is received (see Chapter 4 on checking for function keys in BASIC). INPUT waits for a key but you also have to press RETURN, and the cursor is also in operation.

INKEY$ will sit and wait for a key press, after emptying the keyboard buffer, and, if required will check for a particular key or keys. To allow for further checks we use the reserved variable 'ST' to store the input. Using ST is easy in that it has a predefined location in zero page.

ASSEMBLY LISTING

```
  9 *=$904E
 10                BNE STRING ! PARAMETERS PRESENT
 20 ANYKEY         LDA #$00   ! CLEAR KEY BUFFER
 30                STA $CB
 40 BYTE           JSR $FFE4  ! GET CHARACTER
 50                BEQ BYTE   ! NO KEY
 60                STA $90    ! ST LOCATION
 70                RTS
 80 STRING         JSR $AD9E  ! GET STRING
 90                JSR $B6A3  ! DISCARD UNWANTED STRING
100                CMP #$00   ! NULL STRING?
110                BEQ ANYKEY ! NULL STRING
120                STA $FB    ! NO OF CHARS IN STRING
130                LDA #$00   ! EMPTY KEY BUFFER
140                STA $C6
150 BYTE1          JSR $FFE4  ! GET CHARACTER
160                BEQ BYTE1  ! NO KEY
170                LDY $FB    ! GET NO. OF CHARS
180                DEY
190 NEXT           CMP ($22),Y! CHECK STRING
200                BEQ MATCH  ! FOUND SAME CHAR
210                DEY
220                BPL NEXT   ! CONTINUE SEARCH
230                BMI BYTE1  ! ANOTHER KEY PRESS
240 MATCH          STA $90    ! ST LOCATION
250                RTS


9050 ANYKEY         9054 BYTE
906C BYTE1          907D MATCH
9074 NEXT           905C STRING
```

LINES 10–70: If there are parameters (cases ii, iii and iv of the command syntax), the zero flag will not be set and these lines are skipped over, at least for the time being. Proceeding on we set the flag for the number of characters in the keyboard buffer to zero. The KERNAL routine at $FFE4 will return the ASCII value of key presses in the order they were placed in the buffer. If none, then the accumulator will hold zero, so we continue to call the routine until a value is returned. That value is placed in the location which the reserved variable ST uses, and we return to continue the BASIC program.

LINES 80–240: The call to the ROM routine does our string work. It finds the string, especially if it is a variable, determining its length and giving syntax errors if a non-string parameter was supplied. On returning from the routine, the number of characters will be in A and the start address in locations $22 and $23. If there were no characters in the string, we branch back to the previous section and wait for any key.

After clearing the buffer and getting a key press value we can check it against the string. The Y register will be loaded with the number of characters and decreased as we check the whole string. If the complete string is checked and no match is found, then the next key press is evaluated. Once a match is found, it is stored in ST and we return to carry on with your program.

LOMEM and HIMEM – Setting the area of work

COMMAND SYNTAX

LOMEM address
HIMEM address

The address range that is permissible with these commands is between 1024 and 32767. 'ILLEGAL QUANTITY' errors are given outside this range. The actual start of a program will be one greater than the address given in LOMEM. Commands can be used in direct or program mode.

Changing the memory configuration is a useful, and indeed necessary, task. By raising the base of a program, you can store items such as sprite data, hires screens or even two normal screens and it will not be affected by a program.

At the other end you may wish to put a machine code routine and so to protect it at the top of memory from being overwritten by the variables, so you can set the limit of BASIC to below your routine.

LOMEM will set the lower and HIMEM the upper limit of BASIC. So that they could be used in a loader program the routine does not clear that program. Subsequent programs will be loaded to the new LOMEM address. The ideal place for these commands is at the beginning of a program before any variables are defined. Variables defined after these commands will be placed in the new area. You can use CHAIN to load the next program if there are variables you wish to transfer.

ASSEMBLY LISTING

```
  9 *=$9169
 10 INPUT        BNE GATHER ! PARAMETERS
 20              JMP $AF08  ! SYNTAX ERROR
 30 GATHER       JSR $81F5  ! GET PARAMETERS
 40              LDA $15
 50              CMP #$04   ! CHECK LOW LIMIT
 60              BCS TOP    ! O.K.
 70 ERROR        LDX #$0E   ! ILLEGAL QUANTITY
 80              JMP $A437  ! ERROR ROUTINE
 90 TOP          CMP #$80   ! UPPER LIMIT
100              BCS ERROR  ! FAILED
110              RTS
120 ! START OF HIMEM
130              JSR INPUT
140              STA $38    ! SET TOP POINTER
150              LDA $14
160              STA $37
170              JMP $A65E  ! CLR AND RETURN
180 ! START OF LOMEM
190              JSR INPUT
200              LDY #$00
210              TYA
220              STA ($14),Y! CLEAR FIRST 3 BYTES
230              INY
240              STA ($14),Y
250              INY
260              STA ($14),Y
270              LDA $14
280              CLC
290              ADC #$01
300              STA $2B    ! SET START OF BASIC
310              TAX
320              LDA $15
330              ADC #$00
340              STA $2C
350              TAY
360              TXA
370              ADC #$02
380              STA $2D    ! SET START OF VARIABLES
390              TYA
400              ADC #$00
```

```
410           STA $2E
420           JMP $A663  ! CLR AND RETURN


9177 ERROR         916E GATHER
9169 INPUT         917C TOP
```

LINES 10–110: INPUT: This subroutine is used by both commands. It deals with the gathering and checking of addresses. First we check that there is an address. No address, then no command, and a SYNTAX ERROR is given. When the address is picked up, it is first checked for the lower limit and then for the higher.

LINES 130–170: HIMEM: After visiting the input routine, we place the address in the pointers to the limit of BASIC. We then jump to the CLR routine to finish off: this will set all the remaining relevant pointers (such as the string pointer).

LINES 190–420: LOMEM: BASIC requires that the first byte of the BASIC program area is zero (normally 2048, $0800) and that two zeros signify the end of the program. In the new area these will be together, as there is no program, so we set those first from the address given. To set the start of the program we increase it by one, and from that we add a further two for the address to the start of the variables, or end of program if you prefer. Calling the CLR routine will set the end of variables and array pointers.

QUIT

COMMAND SYNTAX

QUIT

There are no arguments with this particular command.

QUIT disables the UTILITY and its commands, leaving you with the standard BASIC. It does not, however, reset the top of memory back to its original ($A000). This will leave the UTILITY intact which can be reinitiated by SYS 32768.

QUIT simply restores all the vectors and pointers we changed on start up to their standard values.

ASSEMBLY LISTING

```
 9 *=$91B7
10           LDA #$76
20           STA $0304  ! TOKENISE BASIC TEXT
30           LDA #$A5
40           STA $0305
```

```
 50      LDA #$1A
 60      STA $0306  ! BASIC TEXT LIST
 70      LDA #$E4
 80      STA $0308  ! BASIC CHAR DISPATCH
 90      LDA #$A7
100      STA $0307
110      STA $0309
120      LDA #$86
130      STA $030A  ! BASIC TOKEN EVALUATION
140      LDA #$AE
150      STA $030B
160      LDA #$FE
170      STA $0317  ! BRK INTERRUPT
180      STA $0319  ! NMI INTERRUPT
190      LDA #$66
200      STA $0316
210      LDA #$47
220      STA $0318
230      SEI
240      LDA #$48
250      STA $028F  ! KEYBOARD TABLE SETUP
260      LDA #$EB
270      STA $0290
280      CLI
290      PLA
300      PLA
310      JMP $A474  ! READY FOR BASIC
```

9 The complete utility

Introduction

We are going to supply the complete UTILITY in the form of a Supermon listing. If you do not possess a monitor, you can find Supermon in the appendices. For the area $80DE to $81F4, keywords and vectors, use the M function of the monitor. You may also find it easier to use the memory dump in Chapter 6 for that area. Save to tape or disk regularly as you go.

We had thought of also giving the UTILITY in DATA statement form. This would have come to about 690 lines, of seven items of data on each, which would have been a mammoth task of programming for anyone and very prone to error.

```
8000 20 0F 80    JSR $800F
8003 20 54 80    JSR $8054
8006 20 41 80    JSR $8041
8009 20 34 80    JSR $8034
800C 4C 00 92    JMP $9200
800F A9 09       LDA #$09
8011 8D 04 03    STA $0304
8014 A9 BC       LDA #$BC
8016 8D 06 03    STA $0306
8019 A9 02       LDA #$02
801B 8D 08 03    STA $0308
801E A9 29       LDA #$29
8020 8D 0A 03    STA $030A
8023 A9 82       LDA #$82
8025 8D 05 03    STA $0305
8028 8D 07 03    STA $0307
802B A9 83       LDA #$83
802D 8D 09 03    STA $0309
8030 8D 0B 03    STA $030B
8033 60          RTS
8034 A9 FF       LDA #$FF
8036 85 37       STA $37
8038 85 33       STA $33
803A A9 7F       LDA #$7F
803C 85 38       STA $38
803E 85 34       STA $34
8040 60          RTS
8041 A9 7E       LDA #$7E
8043 8D 16 03    STA $0316
8046 A9 61       LDA #$61
8048 8D 18 03    STA $0318
804B A9 80       LDA #$80
804D 8D 17 03    STA $0317
8050 8D 19 03    STA $0319
8053 60          RTS
8054 78          SEI
8055 A9 22       LDA #$22
8057 8D 8F 02    STA $028F
805A A9 87       LDA #$87
805C 8D 90 02    STA $0290
805F 58          CLI
8060 60          RTS
8061 48          PHA
8062 8A          TXA
8063 48          PHA
8064 98          TYA
8065 48          PHA
8066 A9 7F       LDA #$7F
```

```
8068 8D 0D DD    STA $DD0D
806B AC 0D DD    LDY $DD0D
806E 10 03       BPL $8073
8070 4C 72 FE    JMP $FE72
8073 20 BC F6    JSR $F6BC
8076 20 E1 FF    JSR $FFE1
8079 F0 03       BEQ $807E
807B 4C 72 FE    JMP $FE72
807E 20 15 FD    JSR $FD15
8081 20 A3 FD    JSR $FDA3
8084 20 18 E5    JSR $E518
8087 20 54 80    JSR $8054
808A 20 41 80    JSR $8041
808D 6C 02 A0    JMP ($A002)
8090 98          TYA
8091 87          ???
8092 4C 86 B2    JMP $B286
8095 83          ???
8096 9F          ???
8097 84 EB       STY $EB
8099 84 36       STY $36
809B 85 BE       STA $BE
809D 85 14       STA $14
809F 84 51       STY $51
80A1 83          ???
80A2 A6 83       LDX $83
80A4 AE 8F B4    LDX $B48F
80A7 8F          ???
80A8 80          ???
80A9 83          ???
80AA AC 83 51    LDY $5183
80AD 8E C4 89    STX $89C4
80B0 43          ???
80B1 8F          ???
80B2 A6 87       LDX $87
80B4 92          ???
80B5 8B          ???
80B6 2D 84 D1    AND $D184
80B9 8F          ???
80BA 3A          ???
80BB A9 D1       LDA #$D1
80BD A8          TAY
80BE 30 86       BMI $8046
80C0 B6 91       LDX $91,Y
80C2 39 8D 10    AND $108D,Y
80C5 86 B5       STX $B5
80C7 92          ???
80C8 8C 91 80    STY $8091
80CB 91 4D       STA ($4D),Y
80CD 90 FB       BCC $80CA
80CF 85 6E       STA $6E
80D1 88          DEY
80D2 60          RTS
80D3 8D FF FF    STA $FFFF
80D6 FF          ???
80D7 FF          ???
80D8 00          BRK
80D9 FF          ???
80DA FF          ???
80DB FF          ???
80DC F6 FF       INC $FF,X
80DE B6 F7       LDX $F7,Y
80E0 00          BRK
80E1 60          RTS
80E2 00          BRK
80E3 00          BRK
80E4 D6 83       DEC $83,X
80E6 D6 83       DEC $83,X
80E8 00          BRK
80E9 00          BRK
80EA 00          BRK
80EB 68          PLA
80EC 00          BRK
80ED 00          BRK
80EE 00          BRK
80EF 40          RTI
80F0 00          BRK
80F1 00          BRK
80F2 00          BRK
80F3 40          RTI
80F4 00          BRK
80F5 40          RTI
80F6 4F          ???
80F7 46 C6       LSR $C6
80F9 4B          ???
80FA 45 D9       EOR $D9
80FC 44          ???
80FD 4F          ???
80FE 4B          ???
80FF C5 54       CMP $54
8101 45 CE       EOR $CE
8103 54          ???
```

```
8104 57          ???
8105 CF          ???
8106 48          PHA
8107 45 D8       EOR $D8
8109 42          ???
810A 49 CE       EOR #$CE
810C 4F          ???
810D 4C C4 43    JMP $43C4
8110 4F          ???
8111 4C 4F 55    JMP $554F
8114 D2          ???
8115 57          ???
8116 52          ???
8117 49 54       EOR #$54
8119 C5 43       CMP $43
811B 47          ???
811C 4F          ???
811D 54          ???
811E CF          ???
811F 43          ???
8120 47          ???
8121 4F          ???
8122 53          ???
8123 55 C2       EOR $C2,X
8125 50 4C       BVC $8173
8127 4F          ???
8128 D4          ???
8129 45 4E       EOR $4E
812B 54          ???
812C 45 D2       EOR $D2
812E 44          ???
812F 55 4D       EOR $4D,X
8131 D0 52       BNE $8185
8133 45 4E       EOR $4E
8135 55 CD       EOR $CD,X
8137 44          ???
8138 45 4C       EOR $4C
813A 45 54       EOR $54
813C C5 4D       CMP $4D
813E 45 52       EOR $52
8140 47          ???
8141 C5 43       CMP $43
8143 4F          ???
8144 44          ???
8145 45 D2       EOR $D2
8147 41 55       EOR ($55,X)
8149 54          ???
814A CF          ???
814B 50 52       BVC $819F
814D 4F          ???
814E C3          ???
814F 44          ???
8150 50 52       BVC $81A4
8152 4F          ???
8153 C3          ???
8154 45 50       EOR $50
8156 52          ???
8157 4F          ???
8158 C3          ???
8159 50 4F       BVC $81AA
815B D0 51       BNE $81AE
815D 55 49       EOR $49,X
815F D4          ???
8160 54          ???
8161 52          ???
8162 41 43       EOR ($43,X)
8164 C5 52       CMP $52
8166 45 53       EOR $53
8168 45 D4       EOR $D4
816A 43          ???
816B 48          PHA
816C 41 49       EOR ($49,X)
816E CE 4C 4F    DEC $4F4C
8171 4D 45 CD    EOR $CD45
8174 48          PHA
8175 49 4D       EOR #$4D
8177 45 CD       EOR $CD
8179 49 4E       EOR #$4E
817B 4B          ???
817C 45 59       EOR $59
817E A4 4D       LDY $4D
8180 45 CD       EOR $CD
8182 41 50       EOR ($50,X)
8184 50 45       BVC $81CB
8186 4E C4 54    LSR $54C4
8189 52          ???
818A 4F          ???
818B 46 C6       LSR $C6
818D 5A          ???
818E 5A          ???
818F 5A          ???
8190 5A          ???
```

```
8191 5A          ???
8192 EA          NOP
8193 5A          ???
8194 5A          ???
8195 5A          ???
8196 5A          ???
8197 5A          ???
8198 5A          ???
8199 EA          NOP
819A 5A          ???
819B 5A          ???
819C 5A          ???
819D 5A          ???
819E 5A          ???
819F 5A          ???
81A0 5A          ???
81A1 EA          NOP
81A2 5A          ???
81A3 5A          ???
81A4 5A          ???
81A5 5A          ???
81A6 EA          NOP
81A7 5A          ???
81A8 5A          ???
81A9 5A          ???
81AA 5A          ???
81AB EA          NOP
81AC 5A          ???
81AD 5A          ???
81AE 5A          ???
81AF 5A          ???
81B0 EA          NOP
81B1 5A          ???
81B2 5A          ???
81B3 5A          ???
81B4 5A          ???
81B5 5A          ???
81B6 5A          ???
81B7 EA          NOP
81B8 5A          ???
81B9 5A          ???
81BA 5A          ???
81BB 5A          ???
81BC 5A          ???
81BD EA          NOP
81BE 5A          ???
81BF 5A          ???
81C0 5A          ???
81C1 5A          ???
81C2 5A          ???
81C3 5A          ???
81C4 5A          ???
81C5 5A          ???
81C6 EA          NOP
81C7 44          ???
81C8 45 45       EOR $45
81CA CB          ???
81CB 00          BRK
81CC FF          ???
81CD FF          ???
81CE FF          ???
81CF FF          ???
81D0 FF          ???
81D1 FF          ???
81D2 FF          ???
81D3 FF          ???
81D4 FF          ???
81D5 FF          ???
81D6 FD FF FF    SBC $FFFF,X
81D9 FF          ???
81DA FF          ???
81DB FF          ???
81DC FF          ???
81DD 7F          ???
81DE FF          ???
81DF FF          ???
81E0 00          BRK
81E1 00          BRK
81E2 00          BRK
81E3 00          BRK
81E4 00          BRK
81E5 00          BRK
81E6 00          BRK
81E7 00          BRK
81E8 00          BRK
81E9 00          BRK
81EA 00          BRK
81EB 08          PHP
81EC 01 00       ORA ($00,X)
81EE 00          BRK
81EF 00          BRK
81F0 00          BRK
```

```
81F1 00          BRK
81F2 00          BRK
81F3 00          BRK
81F4 00          BRK
81F5 20 8A AD    JSR $AD8A
81F8 4C F7 B7    JMP $B7F7
81FB A5 01       LDA $01
81FD 29 FE       AND #$FE
81FF 85 01       STA $01
8201 60          RTS
8202 A5 01       LDA $01
8204 09 01       ORA #$01
8206 85 01       STA $01
8208 60          RTS
8209 A6 7A       LDX $7A
820B A0 04       LDY #$04
820D 84 0F       STY $0F
820F BD 00 02    LDA $0200,X
8212 10 07       BPL $821B
8214 C9 FF       CMP #$FF
8216 F0 3E       BEQ $8256
8218 E8          INX
8219 D0 F4       BNE $820F
821B C9 20       CMP #$20
821D F0 37       BEQ $8256
821F 85 08       STA $08
8221 C9 22       CMP #$22
8223 F0 56       BEQ $827B
8225 24 0F       BIT $0F
8227 70 2D       BVS $8256
8229 C9 3F       CMP #$3F
822B D0 04       BNE $8231
822D A9 99       LDA #$99
822F D0 25       BNE $8256
8231 C9 30       CMP #$30
8233 90 04       BCC $8239
8235 C9 3C       CMP #$3C
8237 90 1D       BCC $8256
8239 84 71       STY $71
823B A0 00       LDY #$00
823D 84 0B       STY $0B
823F 88          DEY
8240 86 7A       STX $7A
8242 CA          DEX
8243 C8          INY
8244 E8          INX
8245 BD 00 02    LDA $0200,X
8248 38          SEC
8249 F9 9E A0    SBC $A09E,Y
824C F0 F5       BEQ $8243
824E C9 80       CMP #$80
8250 D0 30       BNE $8282
8252 05 0B       ORA $0B
8254 A4 71       LDY $71
8256 E8          INX
8257 C8          INY
8258 99 FB 01    STA $01FB,Y
825B B9 FB 01    LDA $01FB,Y
825E F0 59       BEQ $82B9
8260 38          SEC
8261 E9 3A       SBC #$3A
8263 F0 04       BEQ $8269
8265 C9 49       CMP #$49
8267 D0 02       BNE $826B
8269 85 0F       STA $0F
826B 38          SEC
826C E9 55       SBC #$55
826E D0 9F       BNE $820F
8270 85 08       STA $08
8272 BD 00 02    LDA $0200,X
8275 F0 DF       BEQ $8256
8277 C5 08       CMP $08
8279 F0 DB       BEQ $8256
827B C8          INY
827C 99 FB 01    STA $01FB,Y
827F E8          INX
8280 D0 F0       BNE $8272
8282 A6 7A       LDX $7A
8284 E6 0B       INC $0B
8286 C8          INY
8287 B9 9D A0    LDA $A09D,Y
828A 10 FA       BPL $8286
828C B9 9E A0    LDA $A09E,Y
828F D0 B4       BNE $8245
8291 A0 FF       LDY #$FF
8293 CA          DEX
8294 C8          INY
8295 E8          INX
8296 BD 00 02    LDA $0200,X
8299 38          SEC
829A F9 F6 80    SBC $80F6,Y
829D F0 F5       BEQ $8294
```

```
829F C9 80       CMP #$80
82A1 D0 02       BNE $82A5
82A3 F0 AD       BEQ $8252
82A5 A6 7A       LDX $7A
82A7 E6 0B       INC $0B
82A9 C8          INY
82AA B9 F5 80    LDA $80F5,Y
82AD 10 FA       BPL $82A9
82AF B9 F6 80    LDA $80F6,Y
82B2 D0 E2       BNE $8296
82B4 BD 00 02    LDA $0200,X
82B7 10 9B       BPL $8254
82B9 4C 09 A6    JMP $A609
82BC 10 3E       BPL $82FC
82BE C9 FF       CMP #$FF
82C0 F0 3A       BEQ $82FC
82C2 24 0F       BIT $0F
82C4 30 36       BMI $82FC
82C6 C9 CC       CMP #$CC
82C8 90 0E       BCC $82D8
82CA 38          SEC
82CB E9 CB       SBC #$CB
82CD AA          TAX
82CE A9 F6       LDA #$F6
82D0 85 22       STA $22
82D2 A9 80       LDA #$80
82D4 85 23       STA $23
82D6 D0 0C       BNE $82E4
82D8 38          SEC
82D9 E9 7F       SBC #$7F
82DB AA          TAX
82DC A9 9E       LDA #$9E
82DE 85 22       STA $22
82E0 A9 A0       LDA #$A0
82E2 85 23       STA $23
82E4 84 49       STY $49
82E6 A0 FF       LDY #$FF
82E8 CA          DEX
82E9 F0 07       BEQ $82F2
82EB C8          INY
82EC B1 22       LDA ($22),Y
82EE 10 FB       BPL $82EB
82F0 30 F6       BMI $82E8
82F2 C8          INY
82F3 B1 22       LDA ($22),Y
82F5 30 08       BMI $82FF
82F7 20 47 AB    JSR $AB47
82FA D0 F6       BNE $82F2
82FC 4C F3 A6    JMP $A6F3
82FF 4C EF A6    JMP $A6EF
8302 20 73 00    JSR $0073
8305 C9 CC       CMP #$CC
8307 90 1A       BCC $8323
8309 C9 EE       CMP #$EE
830B B0 16       BCS $8323
830D 20 13 83    JSR $8313
8310 4C EA A7    JMP $A7EA
8313 38          SEC
8314 E9 CC       SBC #$CC
8316 0A          ASL
8317 A8          TAY
8318 B9 91 80    LDA $8091,Y
831B 48          PHA
831C B9 90 80    LDA $8090,Y
831F 48          PHA
8320 4C 73 00    JMP $0073
8323 20 79 00    JSR $0079
8326 4C E7 A7    JMP $A7E7
8329 A9 00       LDA #$00
832B 85 0D       STA $0D
832D 20 73 00    JSR $0073
8330 C9 F7       CMP #$F7
8332 90 18       BCC $834C
8334 C9 F8       CMP #$F8
8336 B0 14       BCS $834C
8338 20 3C 83    JSR $833C
833B 60          RTS
833C 38          SEC
833D E9 F6       SBC #$F6
833F 0A          ASL
8340 A8          TAY
8341 B9 E5 80    LDA $80E5,Y
8344 48          PHA
8345 B9 E4 80    LDA $80E4,Y
8348 48          PHA
8349 4C 73 00    JMP $0073
834C 20 79 00    JSR $0079
834F 4C 8D AE    JMP $AE8D
8352 20 F5 81    JSR $81F5
8355 A5 14       LDA $14
8357 29 0F       AND #$0F
8359 8D 21 D0    STA $D021
```

```
835C 20 79 00    JSR $0079
835F F0 1F       BEQ $8380
8361 20 FD AE    JSR $AEFD
8364 20 F5 81    JSR $81F5
8367 A5 14       LDA $14
8369 29 0F       AND #$0F
836B 8D 20 D0    STA $D020
836E 20 79 00    JSR $0079
8371 F0 0D       BEQ $8380
8373 20 FD AE    JSR $AEFD
8376 20 F5 81    JSR $81F5
8379 A5 14       LDA $14
837B 29 0F       AND #$0F
837D 8D 86 02    STA $0286
8380 60          RTS
8381 20 FA AE    JSR $AEFA
8384 20 F5 81    JSR $81F5
8387 A5 14       LDA $14
8389 C9 28       CMP #$28
838B 90 03       BCC $8390
838D 20 48 B2    JSR $B248
8390 48          PHA
8391 20 FD AE    JSR $AEFD
8394 20 F5 81    JSR $81F5
8397 A6 14       LDX $14
8399 E0 19       CPX #$19
839B B0 F0       BCS $838D
839D 68          PLA
839E A8          TAY
839F 18          CLC
83A0 20 F0 FF    JSR $FFF0
83A3 20 73 00    JSR $0073
83A6 60          RTS
83A7 20 81 83    JSR $8381
83AA 4C A0 AA    JMP $AAA0
83AD 20 81 83    JSR $8381
83B0 4C BF AB    JMP $ABBF
83B3 20 F5 81    JSR $81F5
83B6 20 FD AE    JSR $AEFD
83B9 20 8A AD    JSR $AD8A
83BC A5 66       LDA $66
83BE 30 14       BMI $83D4
83C0 C9 91       CMP #$91
83C2 B0 10       BCS $83D4
83C4 20 9B BC    JSR $BC9B
83C7 A5 65       LDA $65
83C9 A6 64       LDX $64
83CB A0 00       LDY #$00
83CD 91 14       STA ($14),Y
83CF C8          INY
83D0 8A          TXA
83D1 91 14       STA ($14),Y
83D3 60          RTS
83D4 4C 48 B2    JMP $B248
83D7 A5 15       LDA $15
83D9 48          PHA
83DA A5 14       LDA $14
83DC 48          PHA
83DD 20 FA AE    JSR $AEFA
83E0 20 F5 81    JSR $81F5
83E3 20 F7 AE    JSR $AEF7
83E6 A0 01       LDY #$01
83E8 B1 14       LDA ($14),Y
83EA AA          TAX
83EB 88          DEY
83EC B1 14       LDA ($14),Y
83EE A8          TAY
83EF 68          PLA
83F0 85 14       STA $14
83F2 68          PLA
83F3 85 15       STA $15
83F5 8A          TXA
83F6 20 91 B3    JSR $B391
83F9 20 01 84    JSR $8401
83FC 68          PLA
83FD 68          PLA
83FE 4C 8D AD    JMP $AD8D
8401 A5 66       LDA $66
8403 10 0A       BPL $840F
8405 A0 84       LDY #$84
8407 A9 10       LDA #$10
8409 20 8C BA    JSR $BA8C
840C 20 6A B8    JSR $B86A
840F 60          RTS
8410 91 00       STA ($00),Y
8412 00          BRK
8413 00          BRK
8414 00          BRK
8415 A9 FF       LDA #$FF
8417 A0 01       LDY #$01
8419 91 2B       STA ($2B),Y
841B 20 33 A5    JSR $A533
```

```
841E A5 22        LDA $22         8485 38           SEC
8420 18           CLC             8486 20 49 BC     JSR $BC49
8421 69 02        ADC #$02        8489 20 DF BD     JSR $BDDF
8423 85 2D        STA $2D         848C 20 87 B4     JSR $B487
8425 A5 23        LDA $23         848F 20 A6 B6     JSR $B6A6
8427 69 00        ADC #$00        8492 A2 00        LDX #$00
8429 85 2E        STA $2E         8494 BD 00 01     LDA $0100,X
842B 4C 60 A6     JMP $A660       8497 9D 00 02     STA $0200,X
842E 20 F5 81     JSR $81F5       849A F0 03        BEQ $849F
8431 20 FD AE     JSR $AEFD       849C E8           INX
8434 A5 14        LDA $14         849D D0 F5        BNE $8494
8436 85 FB        STA $FB         849F 60           RTS
8438 A5 15        LDA $15         84A0 F0 47        BEQ $84E9
843A 85 FC        STA $FC         84A2 B0 45        BCS $84E9
843C 20 F5 81     JSR $81F5       84A4 20 F5 81     JSR $81F5
843F A5 14        LDA $14         84A7 A9 20        LDA #$20
8441 85 FD        STA $FD         84A9 20 D2 FF     JSR $FFD2
8443 A9 4D        LDA #$4D        84AC A0 02        LDY #$02
8445 8D 02 03     STA $0302       84AE 20 D1 84     JSR $84D1
8448 A9 84        LDA #$84        84B1 88           DEY
844A 8D 03 03     STA $0303       84B2 D0 FA        BNE $84AE
844D AD 00 02     LDA $0200       84B4 A4 14        LDY $14
8450 F0 20        BEQ $8472       84B6 84 15        STY $15
8452 A6 FB        LDX $FB         84B8 A0 02        LDY #$02
8454 A5 FC        LDA $FC         84BA 20 D1 84     JSR $84D1
8456 20 7F 84     JSR $847F       84BD 88           DEY
8459 86 C6        STX $C6         84BE D0 FA        BNE $84BA
845B BD 00 02     LDA $0200,X     84C0 20 79 00     JSR $0079
845E 9D 77 02     STA $0277,X     84C3 C9 2C        CMP #$2C
8461 CA           DEX             84C5 D0 09        BNE $84D0
8462 10 F7        BPL $845B       84C7 20 D2 FF     JSR $FFD2
8464 18           CLC             84CA 20 73 00     JSR $0073
8465 A5 FB        LDA $FB         84CD 4C A0 84     JMP $84A0
8467 65 FD        ADC $FD         84D0 60           RTS
8469 85 FB        STA $FB         84D1 A2 04        LDX #$04
846B 90 02        BCC $846F       84D3 A9 00        LDA #$00
846D E6 FC        INC $FC         84D5 06 15        ASL $15
846F 4C 83 A4     JMP $A483       84D7 2A           ROL
8472 A9 83        LDA #$83        84D8 CA           DEX
8474 8D 02 03     STA $0302       84D9 D0 FA        BNE $84D5
8477 A9 A4        LDA #$A4        84DB C9 0A        CMP #$0A
8479 8D 03 03     STA $0303       84DD 90 04        BCC $84E3
847C 6C 02 03     JMP ($0302)     84DF 18           CLC
847F 86 63        STX $63         84E0 69 37        ADC #$37
8481 85 62        STA $62         84E2 2C 69 30     BIT $3069
8483 A2 90        LDX #$90        84E5 20 D2 FF     JSR $FFD2
```

```
84E8 60        RTS
84E9 4C 08 AF  JMP $AF08
84EC F0 46     BEQ $8534
84EE B0 44     BCS $8534
84F0 20 F5 81  JSR $81F5
84F3 A9 20     LDA #$20
84F5 20 D2 FF  JSR $FFD2
84F8 A5 15     LDA $15
84FA F0 16     BEQ $8512
84FC A2 08     LDX #$08
84FE 06 15     ASL $15
8500 B0 03     BCS $8505
8502 A9 30     LDA #$30
8504 2C A9 31  BIT $31A9
8507 20 D2 FF  JSR $FFD2
850A CA        DEX
850B D0 F1     BNE $84FE
850D A9 20     LDA #$20
850F 20 D2 FF  JSR $FFD2
8512 A2 08     LDX #$08
8514 06 14     ASL $14
8516 B0 03     BCS $851B
8518 A9 30     LDA #$30
851A 2C A9 31  BIT $31A9
851D 20 D2 FF  JSR $FFD2
8520 CA        DEX
8521 D0 F1     BNE $8514
8523 20 79 00  JSR $0079
8526 C9 2C     CMP #$2C
8528 D0 09     BNE $8533
852A 20 D2 FF  JSR $FFD2
852D 20 73 00  JSR $0073
8530 4C EC 84  JMP $84EC
8533 60        RTS
8534 4C 08 AF  JMP $AF08
8537 85 63     STA $63
8539 20 73 00  JSR $0073
853C 85 62     STA $62
853E 20 7C 85  JSR $857C
8541 48        PHA
8542 20 73 00  JSR $0073
8545 F0 1B     BEQ $8562
8547 C9 2C     CMP #$2C
8549 F0 1F     BEQ $856A
854B 85 63     STA $63
854D 20 73 00  JSR $0073
8550 85 62     STA $62
8552 20 7C 85  JSR $857C
8555 A8        TAY
8556 68        PLA
8557 20 AD 85  JSR $85AD
855A 20 73 00  JSR $0073
855D C9 2C     CMP #$2C
855F F0 10     BEQ $8571
8561 60        RTS
8562 68        PLA
8563 A8        TAY
8564 A9 00     LDA #$00
8566 20 AD 85  JSR $85AD
8569 60        RTS
856A 68        PLA
856B A8        TAY
856C A9 00     LDA #$00
856E 20 AD 85  JSR $85AD
8571 A9 2C     LDA #$2C
8573 20 D2 FF  JSR $FFD2
8576 20 73 00  JSR $0073
8579 4C 37 85  JMP $8537
857C A0 01     LDY #$01
857E B9 62 00  LDA $0062,Y
8581 C9 30     CMP #$30
8583 90 25     BCC $85AA
8585 C9 47     CMP #$47
8587 B0 21     BCS $85AA
8589 C9 3A     CMP #$3A
858B 90 08     BCC $8595
858D C9 41     CMP #$41
858F 90 19     BCC $85AA
8591 E9 37     SBC #$37
8593 D0 03     BNE $8598
8595 38        SEC
8596 E9 30     SBC #$30
8598 99 14 00  STA $0014,Y
859B 88        DEY
859C 10 E0     BPL $857E
859E A0 04     LDY #$04
85A0 06 15     ASL $15
85A2 88        DEY
85A3 D0 FB     BNE $85A0
85A5 A5 14     LDA $14
85A7 05 15     ORA $15
85A9 60        RTS
```

```
85AA 4C 08 AF    JMP $AF08
85AD 20 91 B3    JSR $B391
85B0 20 01 84    JSR $8401
85B3 20 8D AD    JSR $AD8D
85B6 20 DD BD    JSR $BDDD
85B9 20 87 B4    JSR $B487
85BC 4C 21 AB    JMP $AB21
85BF A5 7A       LDA $7A
85C1 D0 02       BNE $85C5
85C3 C6 7B       DEC $7B
85C5 C6 7A       DEC $7A
85C7 A2 08       LDX #$08
85C9 20 73 00    JSR $0073
85CC 90 03       BCC $85D1
85CE 4C 08 AF    JMP $AF08
85D1 C9 32       CMP #$32
85D3 B0 F9       BCS $85CE
85D5 6A          ROR
85D6 26 14       ROL $14
85D8 CA          DEX
85D9 D0 EE       BNE $85C9
85DB A4 14       LDY $14
85DD A9 00       LDA #$00
85DF 20 AD 85    JSR $85AD
85E2 A9 2F       LDA #$2F
85E4 20 D2 FF    JSR $FFD2
85E7 A5 14       LDA $14
85E9 A0 00       LDY #$00
85EB 20 AD 85    JSR $85AD
85EE 20 73 00    JSR $0073
85F1 C9 2C       CMP #$2C
85F3 D0 06       BNE $85FB
85F5 20 D2 FF    JSR $FFD2
85F8 4C C7 85    JMP $85C7
85FB 60          RTS
85FC A5 9D       LDA $9D
85FE D0 01       BNE $8601
8600 60          RTS
8601 20 26 B5    JSR $B526
8604 38          SEC
8605 A5 33       LDA $33
8607 E5 31       SBC $31
8609 A8          TAY
860A A5 34       LDA $34
860C E5 32       SBC $32
860E 4C AD 85    JMP $85AD
8611 D0 03       BNE $8616
8613 4C 1D A8    JMP $A81D
8616 20 F5 81    JSR $81F5
8619 20 13 A6    JSR $A613
861C B0 05       BCS $8623
861E A2 15       LDX #$15
8620 4C 37 A4    JMP $A437
8623 38          SEC
8624 A5 5F       LDA $5F
8626 E9 01       SBC #$01
8628 85 41       STA $41
862A A5 60       LDA $60
862C E9 00       SBC #$00
862E 85 42       STA $42
8630 60          RTS
8631 A9 FF       LDA #$FF
8633 85 4A       STA $4A
8635 20 8A A3    JSR $A38A
8638 9A          TXS
8639 C9 8D       CMP #$8D
863B F0 05       BEQ $8642
863D A2 16       LDX #$16
863F 4C 37 A4    JMP $A437
8642 E8          INX
8643 E8          INX
8644 E8          INX
8645 E8          INX
8646 E8          INX
8647 9A          TXS
8648 60          RTS
8649 60          RTS
864A 4C 08 AF    JMP $AF08
864D AD 5B 80    LDA $805B
8650 C9 87       CMP #$87
8652 F0 0D       BEQ $8661
8654 A9 87       LDA #$87
8656 8D 5B 80    STA $805B
8659 A9 22       LDA #$22
865B 8D 56 80    STA $8056
865E 20 54 80    JSR $8054
8661 20 79 00    JSR $0079
8664 F0 42       BEQ $86A8
8666 20 F5 81    JSR $81F5
8669 20 FD AE    JSR $AEFD
866C A5 14       LDA $14
866E F0 1E       BEQ $868E
```

```
8670 C9 11       CMP #$11
8672 B0 1A       BCS $868E
8674 C6 14       DEC $14
8676 A5 14       LDA $14
8678 0A          ASL
8679 0A          ASL
867A 0A          ASL
867B 0A          ASL
867C A8          TAY
867D A9 A1       LDA #$A1
867F 85 15       STA $15
8681 A9 00       LDA #$00
8683 85 14       STA $14
8685 A6 0A       LDX $0A
8687 20 79 00    JSR $0079
868A C9 22       CMP #$22
868C F0 03       BEQ $8691
868E 4C 08 AF    JMP $AF08
8691 20 73 00    JSR $0073
8694 F0 0A       BEQ $86A0
8696 C9 22       CMP #$22
8698 F0 06       BEQ $86A0
869A 91 14       STA ($14),Y
869C C8          INY
869D CA          DEX
869E D0 F1       BNE $8691
86A0 A9 00       LDA #$00
86A2 91 14       STA ($14),Y
86A4 20 73 00    JSR $0073
86A7 60          RTS
86A8 A2 00       LDX #$00
86AA 86 5F       STX $5F
86AC E8          INX
86AD A9 20       LDA #$20
86AF 85 22       STA $22
86B1 A9 31       LDA #$31
86B3 85 23       STA $23
86B5 A9 00       LDA #$00
86B7 85 14       STA $14
86B9 A9 A1       LDA #$A1
86BB 85 15       STA $15
86BD 20 DE 86    JSR $86DE
86C0 E6 23       INC $23
86C2 E6 5F       INC $5F
86C4 E8          INX
86C5 E0 0A       CPX #$0A
86C7 90 F4       BCC $86BD
86C9 A9 31       LDA #$31
86CB 85 22       STA $22
86CD A9 30       LDA #$30
86CF 85 23       STA $23
86D1 20 DE 86    JSR $86DE
86D4 E6 23       INC $23
86D6 E6 5F       INC $5F
86D8 E8          INX
86D9 E0 11       CPX #$11
86DB 90 F4       BCC $86D1
86DD 60          RTS
86DE A0 05       LDY #$05
86E0 B9 1C 87    LDA $871C,Y
86E3 20 D2 FF    JSR $FFD2
86E6 88          DEY
86E7 D0 F7       BNE $86E0
86E9 A5 22       LDA $22
86EB 20 D2 FF    JSR $FFD2
86EE A5 23       LDA $23
86F0 20 D2 FF    JSR $FFD2
86F3 A9 2C       LDA #$2C
86F5 20 D2 FF    JSR $FFD2
86F8 A9 22       LDA #$22
86FA 20 D2 FF    JSR $FFD2
86FD A5 5F       LDA $5F
86FF 0A          ASL
8700 0A          ASL
8701 0A          ASL
8702 0A          ASL
8703 A8          TAY
8704 20 FB 81    JSR $81FB
8707 B1 14       LDA ($14),Y
8709 48          PHA
870A 20 02 82    JSR $8202
870D 68          PLA
870E F0 06       BEQ $8716
8710 20 D2 FF    JSR $FFD2
8713 C8          INY
8714 D0 EE       BNE $8704
8716 A9 22       LDA #$22
8718 20 D2 FF    JSR $FFD2
871B 60          RTS
871C 20 59 45    JSR $4559
871F 4B          ???
8720 20 0D A4    JSR $A40D
```

```
8723 CB          ???
8724 C0 03       CPY #$03
8726 90 04       BCC $872C
8728 C0 07       CPY #$07
872A 90 03       BCC $872F
872C 4C 48 EB    JMP $EB48
872F AD 8D 02    LDA $028D
8732 C4 C5       CPY $C5
8734 D0 05       BNE $873B
8736 CD 8E 02    CMP $028E
8739 F0 F1       BEQ $872C
873B 84 C5       STY $C5
873D 8D 8E 02    STA $028E
8740 C0 04       CPY #$04
8742 F0 0B       BEQ $874F
8744 C0 05       CPY #$05
8746 F0 0A       BEQ $8752
8748 C0 06       CPY #$06
874A F0 09       BEQ $8755
874C A0 07       LDY #$07
874E 2C A0 01    BIT $01A0
8751 2C A0 03    BIT $03A0
8754 2C A0 05    BIT $05A0
8757 C9 02       CMP #$02
8759 90 07       BCC $8762
875B F0 03       BEQ $8760
875D A9 09       LDA #$09
875F 2C A9 08    BIT $08A9
8762 84 BB       STY $BB
8764 C6 BB       DEC $BB
8766 18          CLC
8767 65 BB       ADC $BB
8769 0A          ASL
876A 0A          ASL
876B 0A          ASL
876C 0A          ASL
876D A0 A1       LDY #$A1
876F 84 15       STY $15
8771 A0 00       LDY #$00
8773 84 14       STY $14
8775 A8          TAY
8776 A2 00       LDX #$00
8778 20 FB 81    JSR $81FB
877B B1 14       LDA ($14),Y
877D 48          PHA
877E 20 02 82    JSR $8202
8781 68          PLA
8782 F0 0D       BEQ $8791
8784 C9 5F       CMP #$5F
8786 D0 02       BNE $878A
8788 A9 0D       LDA #$0D
878A 9D 77 02    STA $0277,X
878D E8          INX
878E C8          INY
878F D0 E7       BNE $8778
8791 86 C6       STX $C6
8793 A9 7F       LDA #$7F
8795 8D 00 DC    STA $DC00
8798 60          RTS
8799 A9 48       LDA #$48
879B 8D 56 80    STA $8056
879E A9 EB       LDA #$EB
87A0 8D 5B 80    STA $805B
87A3 20 54 80    JSR $8054
87A6 60          RTS
87A7 20 5E 88    JSR $885E
87AA 86 2B       STX $2B
87AC 84 2C       STY $2C
87AE A0 00       LDY #$00
87B0 98          TYA
87B1 91 2B       STA ($2B),Y
87B3 20 10 88    JSR $8810
87B6 86 2D       STX $2D
87B8 84 2E       STY $2E
87BA 20 33 A5    JSR $A533
87BD 20 4D 88    JSR $884D
87C0 A9 F9       LDA #$F9
87C2 A2 87       LDX #$87
87C4 8D 02 03    STA $0302
87C7 8E 03 03    STX $0303
87CA A9 01       LDA #$01
87CC 85 7B       STA $7B
87CE A9 FF       LDA #$FF
87D0 85 7A       STA $7A
87D2 A0 00       LDY #$00
87D4 B1 FB       LDA ($FB),Y
87D6 85 FD       STA $FD
87D8 C8          INY
87D9 B1 FB       LDA ($FB),Y
87DB 85 FE       STA $FE
87DD F0 24       BEQ $8803
87DF C8          INY
```

```
87E0 B1 FB       LDA ($FB),Y
87E2 85 14       STA $14
87E4 C8          INY
87E5 B1 FB       LDA ($FB),Y
87E7 85 15       STA $15
87E9 A2 04       LDX #$04
87EB E8          INX
87EC C8          INY
87ED B1 FB       LDA ($FB),Y
87EF 9D FB 01    STA $01FB,X
87F2 D0 F7       BNE $87EB
87F4 8A          TXA
87F5 A8          TAY
87F6 20 A2 A4    JSR $A4A2
87F9 A5 FD       LDA $FD
87FB A6 FE       LDX $FE
87FD 85 FB       STA $FB
87FF 86 FC       STX $FC
8801 D0 C7       BNE $87CA
8803 A9 83       LDA #$83
8805 A2 A4       LDX #$A4
8807 8D 02 03    STA $0302
880A 8E 03 03    STX $0303
880D 20 74 A4    JSR $A474
8810 20 D4 E1    JSR $E1D4
8813 A9 00       LDA #$00
8815 85 B9       STA $B9
8817 A6 2B       LDX $2B
8819 A4 2C       LDY $2C
881B 20 D5 FF    JSR $FFD5
881E B0 10       BCS $8830
8820 20 B7 FF    JSR $FFB7
8823 29 BF       AND #$BF
8825 F0 08       BEQ $882F
8827 20 38 88    JSR $8838
882A A2 1D       LDX #$1D
882C 4C 37 A4    JMP $A437
882F 60          RTS
8830 48          PHA
8831 20 38 88    JSR $8838
8834 68          PLA
8835 4C F9 E0    JMP $E0F9
8838 A5 FB       LDA $FB
883A 38          SEC
883B E9 02       SBC #$02
883D 85 14       STA $14
883F A5 FC       LDA $FC
8841 E9 00       SBC #$00
8843 85 FC       STA $FC
8845 A9 00       LDA #$00
8847 A8          TAY
8848 91 14       STA ($14),Y
884A C8          INY
884B 91 14       STA ($14),Y
884D A5 FD       LDA $FD
884F A6 FE       LDX $FE
8851 85 2B       STA $2B
8853 86 2C       STX $2C
8855 A5 FB       LDA $FB
8857 A6 FC       LDX $FC
8859 85 2D       STA $2D
885B 86 2E       STX $2E
885D 60          RTS
885E A5 2B       LDA $2B
8860 85 FD       STA $FD
8862 A5 2C       LDA $2C
8864 85 FE       STA $FE
8866 A6 2D       LDX $2D
8868 A4 2E       LDY $2E
886A 86 FB       STX $FB
886C 84 FC       STY $FC
886E 60          RTS
886F 20 5E 88    JSR $885E
8872 8A          TXA
8873 38          SEC
8874 E9 02       SBC #$02
8876 85 2B       STA $2B
8878 98          TYA
8879 E9 00       SBC #$00
887B 85 2C       STA $2C
887D 20 10 88    JSR $8810
8880 86 FB       STX $FB
8882 84 FC       STY $FC
8884 20 4D 88    JSR $884D
8887 20 33 A5    JSR $A533
888A 60          RTS
888B 86 C2       STX $C2
888D A5 3E       LDA $3E
888F 38          SEC
8890 E5 C2       SBC $C2
8892 85 BB       STA $BB
8894 18          CLC
```

```
8895 A5 FB     LDA $FB
8897 65 49     ADC $49
8899 85 5F     STA $5F
889B A5 FC     LDA $FC
889D 69 00     ADC #$00
889F 85 60     STA $60
88A1 A5 5F     LDA $5F
88A3 65 BB     ADC $BB
88A5 85 5A     STA $5A
88A7 A5 60     LDA $60
88A9 69 00     ADC #$00
88AB 85 5B     STA $5B
88AD A5 2D     LDA $2D
88AF 38        SEC
88B0 E5 5A     SBC $5A
88B2 85 58     STA $58
88B4 A8        TAY
88B5 A5 2E     LDA $2E
88B7 E5 5B     SBC $5B
88B9 AA        TAX
88BA E8        INX
88BB 98        TYA
88BC F0 1F     BEQ $88DD
88BE A5 5A     LDA $5A
88C0 18        CLC
88C1 65 58     ADC $58
88C3 85 5A     STA $5A
88C5 90 03     BCC $88CA
88C7 E6 5B     INC $5B
88C9 18        CLC
88CA A5 5F     LDA $5F
88CC 65 58     ADC $58
88CE 85 5F     STA $5F
88D0 90 02     BCC $88D4
88D2 E6 60     INC $60
88D4 98        TYA
88D5 49 FF     EOR #$FF
88D7 A8        TAY
88D8 C8        INY
88D9 C6 5B     DEC $5B
88DB C6 60     DEC $60
88DD B1 5A     LDA ($5A),Y
88DF 91 5F     STA ($5F),Y
88E1 C8        INY
88E2 D0 F9     BNE $88DD
88E4 E6 5B     INC $5B
88E6 E6 60     INC $60
88E8 CA        DEX
88E9 D0 F2     BNE $88DD
88EB 38        SEC
88EC A5 2D     LDA $2D
88EE E5 BB     SBC $BB
88F0 85 2D     STA $2D
88F2 B0 03     BCS $88F7
88F4 C6 2E     DEC $2E
88F6 38        SEC
88F7 A0 00     LDY #$00
88F9 A5 FD     LDA $FD
88FB E5 BB     SBC $BB
88FD 85 FD     STA $FD
88FF 91 FB     STA ($FB),Y
8901 85 57     STA $57
8903 A5 FE     LDA $FE
8905 E9 00     SBC #$00
8907 C8        INY
8908 85 FE     STA $FE
890A 85 58     STA $58
890C 91 FB     STA ($FB),Y
890E 88        DEY
890F B1 57     LDA ($57),Y
8911 85 B9     STA $B9
8913 C8        INY
8914 B1 57     LDA ($57),Y
8916 85 BA     STA $BA
8918 F0 18     BEQ $8932
891A 88        DEY
891B 38        SEC
891C A5 B9     LDA $B9
891E E5 BB     SBC $BB
8920 AA        TAX
8921 91 57     STA ($57),Y
8923 A5 BA     LDA $BA
8925 E9 00     SBC #$00
8927 C8        INY
8928 91 57     STA ($57),Y
892A 85 58     STA $58
892C 8A        TXA
892D 85 57     STA $57
892F 4C 0E 89  JMP $890E
8932 60        RTS
8933 8A        TXA
8934 38        SEC
```

```
8935 E5 3E      SBC $3E
8937 85 BB      STA $BB
8939 18         CLC
893A A5 49      LDA $49
893C 65 BB      ADC $BB
893E B0 04      BCS $8944
8940 C9 FE      CMP #$FE
8942 90 05      BCC $8949
8944 A2 17      LDX #$17
8946 4C 37 A4   JMP $A437
8949 A5 2D      LDA $2D
894B 65 BB      ADC $BB
894D AA         TAX
894E A5 2E      LDA $2E
8950 69 00      ADC #$00
8952 C5 38      CMP $38
8954 D0 07      BNE $895D
8956 E4 37      CPX $37
8958 90 03      BCC $895D
895A 4C 35 A4   JMP $A435
895D 18         CLC
895E A5 2D      LDA $2D
8960 85 5A      STA $5A
8962 65 BB      ADC $BB
8964 85 58      STA $58
8966 A5 2E      LDA $2E
8968 85 5B      STA $5B
896A 69 00      ADC #$00
896C 85 59      STA $59
896E A5 FB      LDA $FB
8970 65 49      ADC $49
8972 85 5F      STA $5F
8974 A5 FC      LDA $FC
8976 69 00      ADC #$00
8978 85 60      STA $60
897A 20 BF A3   JSR $A3BF
897D 18         CLC
897E A0 00      LDY #$00
8980 A5 2D      LDA $2D
8982 65 BB      ADC $BB
8984 85 2D      STA $2D
8986 90 03      BCC $898B
8988 E6 2E      INC $2E
898A 18         CLC
898B A5 FD      LDA $FD
898D 65 BB      ADC $BB
898F 85 FD      STA $FD
8991 85 57      STA $57
8993 91 FB      STA ($FB),Y
8995 A5 FE      LDA $FE
8997 69 00      ADC #$00
8999 C8         INY
899A 85 FE      STA $FE
899C 85 58      STA $58
899E 91 FB      STA ($FB),Y
89A0 88         DEY
89A1 B1 57      LDA ($57),Y
89A3 85 B9      STA $B9
89A5 C8         INY
89A6 B1 57      LDA ($57),Y
89A8 85 BA      STA $BA
89AA F0 18      BEQ $89C4
89AC 88         DEY
89AD 18         CLC
89AE A5 B9      LDA $B9
89B0 65 BB      ADC $BB
89B2 AA         TAX
89B3 91 57      STA ($57),Y
89B5 A5 BA      LDA $BA
89B7 69 00      ADC #$00
89B9 C8         INY
89BA 91 57      STA ($57),Y
89BC 85 58      STA $58
89BE 8A         TXA
89BF 85 57      STA $57
89C1 4C A0 89   JMP $89A0
89C4 60         RTS
89C5 20 F5 81   JSR $81F5
89C8 20 FD AE   JSR $AEFD
89CB A5 14      LDA $14
89CD 85 C9      STA $C9
89CF A5 15      LDA $15
89D1 85 CA      STA $CA
89D3 20 F5 81   JSR $81F5
89D6 20 FD AE   JSR $AEFD
89D9 A5 14      LDA $14
89DB 85 BC      STA $BC
89DD 20 F5 81   JSR $81F5
89E0 A5 14      LDA $14
89E2 85 BD      STA $BD
89E4 A5 15      LDA $15
89E6 85 BE      STA $BE
```

```
89E8 A5 2B      LDA $2B
89EA 85 FB      STA $FB
89EC A5 2C      LDA $2C
89EE 85 FC      STA $FC
89F0 A5 CA      LDA $CA
89F2 D0 0F      BNE $8A03
89F4 A5 C9      LDA $C9
89F6 D0 0B      BNE $8A03
89F8 A0 02      LDY #$02
89FA B1 FB      LDA ($FB),Y
89FC 85 C9      STA $C9
89FE C8         INY
89FF B1 FB      LDA ($FB),Y
8A01 85 CA      STA $CA
8A03 A5 C9      LDA $C9
8A05 85 14      STA $14
8A07 A5 CA      LDA $CA
8A09 85 15      STA $15
8A0B 20 13 A6   JSR $A613
8A0E B0 05      BCS $8A15
8A10 A2 15      LDX #$15
8A12 4C 37 A4   JMP $A437
8A15 A5 5F      LDA $5F
8A17 85 41      STA $41
8A19 A5 60      LDA $60
8A1B 85 42      STA $42
8A1D A0 00      LDY #$00
8A1F B1 FB      LDA ($FB),Y
8A21 85 FD      STA $FD
8A23 C8         INY
8A24 B1 FB      LDA ($FB),Y
8A26 85 FE      STA $FE
8A28 D0 03      BNE $8A2D
8A2A 4C 62 8B   JMP $8B62
8A2D C8         INY
8A2E C8         INY
8A2F C8         INY
8A30 B1 FB      LDA ($FB),Y
8A32 D0 0A      BNE $8A3E
8A34 A5 FD      LDA $FD
8A36 85 FB      STA $FB
8A38 A5 FE      LDA $FE
8A3A 85 FC      STA $FC
8A3C D0 DF      BNE $8A1D
8A3E C9 22      CMP #$22
8A40 D0 0B      BNE $8A4D
```

```
8A42 C8         INY
8A43 B1 FB      LDA ($FB),Y
8A45 F0 ED      BEQ $8A34
8A47 C9 22      CMP #$22
8A49 D0 F7      BNE $8A42
8A4B F0 E2      BEQ $8A2F
8A4D C9 8F      CMP #$8F
8A4F F0 E3      BEQ $8A34
8A51 C9 83      CMP #$83
8A53 F0 DF      BEQ $8A34
8A55 C9 A7      CMP #$A7
8A57 F0 21      BEQ $8A7A
8A59 C9 8A      CMP #$8A
8A5B F0 1D      BEQ $8A7A
8A5D C9 89      CMP #$89
8A5F F0 2C      BEQ $8A8D
8A61 C9 CB      CMP #$CB
8A63 D0 0B      BNE $8A70
8A65 C8         INY
8A66 B1 FB      LDA ($FB),Y
8A68 C9 20      CMP #$20
8A6A F0 F9      BEQ $8A65
8A6C C9 A4      CMP #$A4
8A6E F0 1D      BEQ $8A8D
8A70 C9 8D      CMP #$8D
8A72 F0 19      BEQ $8A8D
8A74 C9 E6      CMP #$E6
8A76 F0 15      BEQ $8A8D
8A78 D0 B5      BNE $8A2F
8A7A C8         INY
8A7B B1 FB      LDA ($FB),Y
8A7D C9 20      CMP #$20
8A7F F0 F9      BEQ $8A7A
8A81 C9 30      CMP #$30
8A83 B0 03      BCS $8A88
8A85 88         DEY
8A86 D0 A7      BNE $8A2F
8A88 88         DEY
8A89 C9 3A      CMP #$3A
8A8B B0 A2      BCS $8A2F
8A8D C8         INY
8A8E B1 FB      LDA ($FB),Y
8A90 C9 20      CMP #$20
8A92 F0 F9      BEQ $8A8D
8A94 84 49      STY $49
8A96 88         DEY
```

```
8A97 A2 00       LDX #$00
8A99 C8          INY
8A9A B1 FB       LDA ($FB),Y
8A9C C9 30       CMP #$30
8A9E 90 0A       BCC $8AAA
8AA0 C9 3A       CMP #$3A
8AA2 B0 06       BCS $8AAA
8AA4 9D 00 02    STA $0200,X
8AA7 E8          INX
8AA8 D0 EF       BNE $8A99
8AAA A9 3A       LDA #$3A
8AAC 9D 00 02    STA $0200,X
8AAF 86 BF       STX $BF
8AB1 A9 02       LDA #$02
8AB3 85 7B       STA $7B
8AB5 A9 00       LDA #$00
8AB7 85 7A       STA $7A
8AB9 20 F5 81    JSR $81F5
8ABC A5 14       LDA $14
8ABE 85 C3       STA $C3
8AC0 A5 15       LDA $15
8AC2 85 C4       STA $C4
8AC4 C5 CA       CMP $CA
8AC6 F0 0A       BEQ $8AD2
8AC8 B0 0E       BCS $8AD8
8ACA A5 49       LDA $49
8ACC 65 BF       ADC $BF
8ACE A8          TAY
8ACF 4C 55 8B    JMP $8B55
8AD2 A5 C3       LDA $C3
8AD4 C5 C9       CMP $C9
8AD6 90 F2       BCC $8ACA
8AD8 A5 BD       LDA $BD
8ADA 85 B9       STA $B9
8ADC A5 41       LDA $41
8ADE 85 58       STA $58
8AE0 A5 BE       LDA $BE
8AE2 85 BA       STA $BA
8AE4 A5 42       LDA $42
8AE6 85 59       STA $59
8AE8 A0 00       LDY #$00
8AEA B1 58       LDA ($58),Y
8AEC 85 5A       STA $5A
8AEE C8          INY
8AEF B1 58       LDA ($58),Y
8AF1 85 5B       STA $5B
8AF3 D0 10       BNE $8B05
8AF5 A0 02       LDY #$02
8AF7 B1 FB       LDA ($FB),Y
8AF9 85 39       STA $39
8AFB C8          INY
8AFC B1 FB       LDA ($FB),Y
8AFE 85 3A       STA $3A
8B00 A2 11       LDX #$11
8B02 4C 37 A4    JMP $A437
8B05 C8          INY
8B06 B1 58       LDA ($58),Y
8B08 85 B7       STA $B7
8B0A C8          INY
8B0B B1 58       LDA ($58),Y
8B0D C5 C4       CMP $C4
8B0F D0 06       BNE $8B17
8B11 A5 B7       LDA $B7
8B13 C5 C3       CMP $C3
8B15 F0 15       BEQ $8B2C
8B17 A5 B9       LDA $B9
8B19 18          CLC
8B1A 65 BC       ADC $BC
8B1C 85 B9       STA $B9
8B1E 90 02       BCC $8B22
8B20 E6 BA       INC $BA
8B22 A5 5A       LDA $5A
8B24 85 58       STA $58
8B26 A5 5B       LDA $5B
8B28 85 59       STA $59
8B2A D0 BC       BNE $8AE8
8B2C A6 B9       LDX $B9
8B2E A5 BA       LDA $BA
8B30 20 7F 84    JSR $847F
8B33 A5 BF       LDA $BF
8B35 85 3E       STA $3E
8B37 E4 3E       CPX $3E
8B39 F0 0B       BEQ $8B46
8B3B B0 06       BCS $8B43
8B3D 20 8B 88    JSR $888B
8B40 4C 46 8B    JMP $8B46
8B43 20 33 89    JSR $8933
8B46 A4 49       LDY $49
8B48 A2 00       LDX #$00
8B4A BD 00 02    LDA $0200,X
8B4D F0 06       BEQ $8B55
8B4F 91 FB       STA ($FB),Y
```

```
8B51 C8        INY
8B52 E8        INX
8B53 D0 F5     BNE $8B4A
8B55 B1 FB     LDA ($FB),Y
8B57 C9 2C     CMP #$2C
8B59 F0 04     BEQ $8B5F
8B5B 88        DEY
8B5C 4C 2F 8A  JMP $8A2F
8B5F 4C 8D 8A  JMP $8A8D
8B62 A0 00     LDY #$00
8B64 B1 41     LDA ($41),Y
8B66 85 5A     STA $5A
8B68 C8        INY
8B69 B1 41     LDA ($41),Y
8B6B 85 5B     STA $5B
8B6D D0 05     BNE $8B74
8B6F 68        PLA
8B70 68        PLA
8B71 4C 74 A4  JMP $A474
8B74 C8        INY
8B75 A5 BD     LDA $BD
8B77 91 41     STA ($41),Y
8B79 C8        INY
8B7A A5 BE     LDA $BE
8B7C 91 41     STA ($41),Y
8B7E 18        CLC
8B7F A5 BD     LDA $BD
8B81 65 BC     ADC $BC
8B83 85 BD     STA $BD
8B85 90 02     BCC $8B89
8B87 E6 BE     INC $BE
8B89 A5 5A     LDA $5A
8B8B 85 41     STA $41
8B8D A5 5B     LDA $5B
8B8F 85 42     STA $42
8B91 D0 CF     BNE $8B62
8B93 A2 00     LDX #$00
8B95 A5 2B     LDA $2B
8B97 85 FB     STA $FB
8B99 A5 2C     LDA $2C
8B9B 85 FC     STA $FC
8B9D A0 00     LDY #$00
8B9F B1 FB     LDA ($FB),Y
8BA1 85 FD     STA $FD
8BA3 C8        INY
8BA4 B1 FB     LDA ($FB),Y
8BA6 85 FE     STA $FE
8BA8 D0 05     BNE $8BAF
8BAA 68        PLA
8BAB 68        PLA
8BAC 4C 74 A4  JMP $A474
8BAF C8        INY
8BB0 C8        INY
8BB1 C8        INY
8BB2 B1 FB     LDA ($FB),Y
8BB4 D0 0C     BNE $8BC2
8BB6 A5 FD     LDA $FD
8BB8 85 FB     STA $FB
8BBA A5 FE     LDA $FE
8BBC 85 FC     STA $FC
8BBE A2 00     LDX #$00
8BC0 F0 DB     BEQ $8B9D
8BC2 C9 FF     CMP #$FF
8BC4 D0 04     BNE $8BCA
8BC6 85 3E     STA $3E
8BC8 F0 21     BEQ $8BEB
8BCA C9 22     CMP #$22
8BCC D0 09     BNE $8BD7
8BCE E8        INX
8BCF E0 02     CPX #$02
8BD1 D0 08     BNE $8BDB
8BD3 A2 00     LDX #$00
8BD5 F0 DA     BEQ $8BB1
8BD7 E0 01     CPX #$01
8BD9 D0 D6     BNE $8BB1
8BDB 85 3E     STA $3E
8BDD C9 C0     CMP #$C0
8BDF 90 02     BCC $8BE3
8BE1 E9 60     SBC #$60
8BE3 C9 60     CMP #$60
8BE5 B0 04     BCS $8BEB
8BE7 C9 21     CMP #$21
8BE9 B0 C6     BCS $8BB1
8BEB 85 3D     STA $3D
8BED 84 49     STY $49
8BEF 86 3C     STX $3C
8BF1 A2 01     LDX #$01
8BF3 C8        INY
8BF4 B1 FB     LDA ($FB),Y
8BF6 C5 3E     CMP $3E
8BF8 D0 03     BNE $8BFD
8BFA E8        INX
```

```
8BFB D0 F6      BNE $8BF3
8BFD 86 3E      STX $3E
8BFF E0 02      CPX #$02
8C01 B0 12      BCS $8C15
8C03 CA         DEX
8C04 A5 3D      LDA $3D
8C06 C9 20      CMP #$20
8C08 F0 08      BEQ $8C12
8C0A A9 00      LDA #$00
8C0C 85 40      STA $40
8C0E 85 3F      STA $3F
8C10 F0 15      BEQ $8C27
8C12 4C 33 8D   JMP $8D33
8C15 A9 00      LDA #$00
8C17 20 7F 84   JSR $847F
8C1A A2 00      LDX #$00
8C1C BD 00 02   LDA $0200,X
8C1F 85 3F      STA $3F
8C21 E8         INX
8C22 BD 00 02   LDA $0200,X
8C25 85 40      STA $40
8C27 A5 3D      LDA $3D
8C29 C9 61      CMP #$61
8C2B 90 51      BCC $8C7E
8C2D C9 7B      CMP #$7B
8C2F B0 4D      BCS $8C7E
8C31 38         SEC
8C32 E9 20      SBC #$20
8C34 85 3D      STA $3D
8C36 A2 07      LDX #$07
8C38 A5 40      LDA $40
8C3A D0 06      BNE $8C42
8C3C CA         DEX
8C3D A5 3F      LDA $3F
8C3F D0 01      BNE $8C42
8C41 CA         DEX
8C42 E4 3E      CPX $3E
8C44 F0 0B      BEQ $8C51
8C46 B0 06      BCS $8C4E
8C48 20 8B 88   JSR $888B
8C4B 4C 51 8C   JMP $8C51
8C4E 20 33 89   JSR $8933
8C51 A4 49      LDY $49
8C53 A9 5B      LDA #$5B
8C55 91 FB      STA ($FB),Y
8C57 A5 40      LDA $40
8C59 F0 03      BEQ $8C5E
8C5B C8         INY
8C5C 91 FB      STA ($FB),Y
8C5E A5 3F      LDA $3F
8C60 F0 03      BEQ $8C65
8C62 C8         INY
8C63 91 FB      STA ($FB),Y
8C65 A9 47      LDA #$47
8C67 C8         INY
8C68 91 FB      STA ($FB),Y
8C6A A9 3E      LDA #$3E
8C6C C8         INY
8C6D 91 FB      STA ($FB),Y
8C6F A5 3D      LDA $3D
8C71 C8         INY
8C72 91 FB      STA ($FB),Y
8C74 A9 5D      LDA #$5D
8C76 C8         INY
8C77 91 FB      STA ($FB),Y
8C79 A6 3C      LDX $3C
8C7B 4C B1 8B   JMP $8BB1
8C7E 85 3D      STA $3D
8C80 A9 50      LDA #$50
8C82 85 62      STA $62
8C84 A9 A3      LDA #$A3
8C86 85 63      STA $63
8C88 A2 51      LDX #$51
8C8A A0 00      LDY #$00
8C8C 20 FB 81   JSR $81FB
8C8F B1 62      LDA ($62),Y
8C91 48         PHA
8C92 20 02 82   JSR $8202
8C95 68         PLA
8C96 C5 3D      CMP $3D
8C98 F0 20      BEQ $8CBA
8C9A C8         INY
8C9B C8         INY
8C9C C8         INY
8C9D CA         DEX
8C9E 10 EC      BPL $8C8C
8CA0 A5 3D      LDA $3D
8CA2 C9 1B      CMP #$1B
8CA4 90 05      BCC $8CAB
8CA6 A2 0D      LDX #$0D
8CA8 4C 37 A4   JMP $A437
8CAB 69 40      ADC #$40
```

```
8CAD 8D 48 A4    STA $A448
8CB0 A9 43       LDA #$43
8CB2 85 62       STA $62
8CB4 A9 A4       LDA #$A4
8CB6 85 63       STA $63
8CB8 D0 12       BNE $8CCC
8CBA C8          INY
8CBB 20 FB 81    JSR $81FB
8CBE B1 62       LDA ($62),Y
8CC0 48          PHA
8CC1 C8          INY
8CC2 B1 62       LDA ($62),Y
8CC4 85 63       STA $63
8CC6 20 02 82    JSR $8202
8CC9 68          PLA
8CCA 85 62       STA $62
8CCC A0 00       LDY #$00
8CCE 20 FB 81    JSR $81FB
8CD1 B1 62       LDA ($62),Y
8CD3 85 C1       STA $C1
8CD5 20 02 82    JSR $8202
8CD8 A5 C1       LDA $C1
8CDA 18          CLC
8CDB 69 04       ADC #$04
8CDD AA          TAX
8CDE A5 40       LDA $40
8CE0 D0 06       BNE $8CE8
8CE2 CA          DEX
8CE3 A5 3F       LDA $3F
8CE5 D0 01       BNE $8CE8
8CE7 CA          DEX
8CE8 E4 3E       CPX $3E
8CEA F0 0B       BEQ $8CF7
8CEC B0 06       BCS $8CF4
8CEE 20 8B 88    JSR $888B
8CF1 4C F7 8C    JMP $8CF7
8CF4 20 33 89    JSR $8933
8CF7 A4 49       LDY $49
8CF9 A9 5B       LDA #$5B
8CFB 91 FB       STA ($FB),Y
8CFD A5 3F       LDA $3F
8CFF F0 03       BEQ $8D04z
8D01 C8          INY
8D02 91 FB       STA ($FB),Y
8D04 A5 40       LDA $40
8D06 F0 03       BEQ $8D0B
8D08 C8          INY
8D09 91 FB       STA ($FB),Y
8D0B 84 49       STY $49
8D0D A0 00       LDY #$00
8D0F 20 FB 81    JSR $81FB
8D12 C8          INY
8D13 B1 62       LDA ($62),Y
8D15 84 C2       STY $C2
8D17 A4 49       LDY $49
8D19 C8          INY
8D1A 91 FB       STA ($FB),Y
8D1C 84 49       STY $49
8D1E A4 C2       LDY $C2
8D20 C4 C1       CPY $C1
8D22 D0 EE       BNE $8D12
8D24 20 02 82    JSR $8202
8D27 A4 49       LDY $49
8D29 A9 5D       LDA #$5D
8D2B C8          INY
8D2C 91 FB       STA ($FB),Y
8D2E A6 3C       LDX $3C
8D30 4C B1 8B    JMP $8BB1
8D33 A4 49       LDY $49
8D35 A6 3C       LDX $3C
8D37 4C B1 8B    JMP $8BB1
8D3A A5 9D       LDA $9D
8D3C F0 21       BEQ $8D5F
8D3E 78          SEI
8D3F A9 FF       LDA #$FF
8D41 8D 47 8E    STA $8E47
8D44 A9 FF       LDA #$FF
8D46 8D 46 8E    STA $8E46
8D49 AE 08 03    LDX $0308
8D4C 8E 50 8E    STX $8E50
8D4F A9 79       LDA #$79
8D51 8D 08 03    STA $0308
8D54 AE 09 03    LDX $0309
8D57 8E 51 8E    STX $8E51
8D5A A9 8D       LDA #$8D
8D5C 8D 09 03    STA $0309
8D5F 58          CLI
8D60 60          RTS
8D61 78          SEI
8D62 A5 9D       LDA $9D
8D64 F0 F9       BEQ $8D5F
8D66 A9 00       LDA #$00
```

```
8D68 8D 46 8E     STA $8E46
8D6B AD 50 8E     LDA $8E50
8D6E 8D 08 03     STA $0308
8D71 AD 51 8E     LDA $8E51
8D74 8D 09 03     STA $0309
8D77 58           CLI
8D78 60           RTS
8D79 8D 49 8E     STA $8E49
8D7C 08           PHP
8D7D 8E 4A 8E     STX $8E4A
8D80 8C 4B 8E     STY $8E4B
8D83 A5 9D        LDA $9D
8D85 F0 0D        BEQ $8D94
8D87 AD 49 8E     LDA $8E49
8D8A AC 4B 8E     LDY $8E4B
8D8D AE 4A 8E     LDX $8E4A
8D90 28           PLP
8D91 6C 50 8E     JMP ($8E50)
8D94 A5 39        LDA $39
8D96 AD 46 8E     LDA $8E46
8D99 F0 EC        BEQ $8D87
8D9B 38           SEC
8D9C 20 F0 FF     JSR $FFF0
8D9F 8E 4D 8E     STX $8E4D
8DA2 8C 4C 8E     STY $8E4C
8DA5 18           CLC
8DA6 A2 00        LDX #$00
8DA8 A0 18        LDY #$18
8DAA 20 F0 FF     JSR $FFF0
8DAD A2 0F        LDX #$0F
8DAF A9 20        LDA #$20
8DB1 20 D2 FF     JSR $FFD2
8DB4 CA           DEX
8DB5 D0 F8        BNE $8DAF
8DB7 18           CLC
8DB8 A2 00        LDX #$00
8DBA A0 18        LDY #$18
8DBC 20 F0 FF     JSR $FFF0
8DBF A9 12        LDA #$12
8DC1 20 D2 FF     JSR $FFD2
8DC4 AD 4F 8E     LDA $8E4F
8DC7 AE 4E 8E     LDX $8E4E
8DCA 20 CD BD     JSR $BDCD
8DCD A9 92        LDA #$92
8DCF 20 D2 FF     JSR $FFD2
8DD2 A9 20        LDA #$20
8DD4 20 D2 FF     JSR $FFD2
8DD7 A9 12        LDA #$12
8DD9 20 D2 FF     JSR $FFD2
8DDC A5 3A        LDA $3A
8DDE 8D 4F 8E     STA $8E4F
8DE1 A6 39        LDX $39
8DE3 8E 4E 8E     STX $8E4E
8DE6 20 CD BD     JSR $BDCD
8DE9 A9 92        LDA #$92
8DEB 20 D2 FF     JSR $FFD2
8DEE 18           CLC
8DEF B0 96        BCS $8D87
8DF1 AE 4D 8E     LDX $8E4D
8DF4 AC 4C 8E     LDY $8E4C
8DF7 20 F0 FF     JSR $FFF0
8DFA 20 E4 FF     JSR $FFE4
8DFD F0 25        BEQ $8E24
8DFF C9 2F        CMP #$2F
8E01 90 21        BCC $8E24
8E03 C9 3A        CMP #$3A
8E05 B0 1D        BCS $8E24
8E07 E9 30        SBC #$30
8E09 D0 07        BNE $8E12
8E0B A9 FF        LDA #$FF
8E0D 8D 47 8E     STA $8E47
8E10 D0 12        BNE $8E24
8E12 AA           TAX
8E13 38           SEC
8E14 A9 00        LDA #$00
8E16 2A           ROL
8E17 CA           DEX
8E18 D0 FC        BNE $8E16
8E1A 8D 48 8E     STA $8E48
8E1D A9 00        LDA #$00
8E1F 8D 47 8E     STA $8E47
8E22 F0 14        BEQ $8E38
8E24 AE 47 8E     LDX $8E47
8E27 F0 0F        BEQ $8E38
8E29 C9 20        CMP #$20
8E2B F0 16        BEQ $8E43
8E2D 20 E4 FF     JSR $FFE4
8E30 F0 FB        BEQ $8E2D
8E32 C9 20        CMP #$20
8E34 F0 0D        BEQ $8E43
8E36 D0 C5        BNE $8DFD
8E38 AE 48 8E     LDX $8E48
```

```
8E3B A0  FF       LDY #$FF
8E3D 88           DEY
8E3E D0  FD       BNE $8E3D
8E40 CA           DEX
8E41 D0  F8       BNE $8E3B
8E43 38           SEC
8E44 F0  A9       BEQ $8DEF
8E46 FF           ???
8E47 FF           ???
8E48 00           BRK
8E49 FF           ???
8E4A 09  0B       ORA #$0B
8E4C 00           BRK
8E4D 00           BRK
8E4E 00           BRK
8E4F 00           BRK
8E50 02           ???
8E51 83           ???
8E52 A5  9D       LDA $9D
8E54 C9  80       CMP #$80
8E56 F0  01       BEQ $8E59
8E58 60           RTS
8E59 A5  2D       LDA $2D
8E5B 85  BB       STA $BB
8E5D A5  2E       LDA $2E
8E5F 85  BC       STA $BC
8E61 A5  BC       LDA $BC
8E63 C5  30       CMP $30
8E65 D0  07       BNE $8E6E
8E67 A5  BB       LDA $BB
8E69 C5  2F       CMP $2F
8E6B D0  01       BNE $8E6E
8E6D 60           RTS
8E6E A0  00       LDY #$00
8E70 B1  BB       LDA ($BB),Y
8E72 C9  80       CMP #$80
8E74 B0  5A       BCS $8ED0
8E76 20  D2 FF    JSR $FFD2
8E79 C8           INY
8E7A B1  BB       LDA ($BB),Y
8E7C C9  7F       CMP #$7F
8E7E B0  17       BCS $8E97
8E80 20  D2 FF    JSR $FFD2
8E83 20  33 8F    JSR $8F33
8E86 A5  BB       LDA $BB
8E88 A4  BC       LDY $BC
```

```
8E8A 20  A2 BB    JSR $BBA2
8E8D 20  DD BD    JSR $BDDD
8E90 20  1E AB    JSR $AB1E
8E93 A9  FF       LDA #$FF
8E95 D0  7A       BNE $8F11
8E97 29  7F       AND #$7F
8E99 20  D2 FF    JSR $FFD2
8E9C A9  24       LDA #$24
8E9E 20  D2 FF    JSR $FFD2
8EA1 20  33 8F    JSR $8F33
8EA4 A9  22       LDA #$22
8EA6 20  D2 FF    JSR $FFD2
8EA9 A0  00       LDY #$00
8EAB B1  BB       LDA ($BB),Y
8EAD AA           TAX
8EAE F0  15       BEQ $8EC5
8EB0 C8           INY
8EB1 B1  BB       LDA ($BB),Y
8EB3 85  22       STA $22
8EB5 C8           INY
8EB6 B1  BB       LDA ($BB),Y
8EB8 85  23       STA $23
8EBA A0  00       LDY #$00
8EBC B1  22       LDA ($22),Y
8EBE 20  D2 FF    JSR $FFD2
8EC1 C8           INY
8EC2 CA           DEX
8EC3 D0  F7       BNE $8EBC
8EC5 A9  22       LDA #$22
8EC7 20  D2 FF    JSR $FFD2
8ECA F0  45       BEQ $8F11
8ECC D0  43       BNE $8F11
8ECE B0  91       BCS $8E61
8ED0 29  7F       AND #$7F
8ED2 20  D2 FF    JSR $FFD2
8ED5 C8           INY
8ED6 B1  BB       LDA ($BB),Y
8ED8 C9  7F       CMP #$7F
8EDA B0  12       BCS $8EEE
8EDC 20  D2 FF    JSR $FFD2
8EDF 20  33 8F    JSR $8F33
8EE2 A9  46       LDA #$46
8EE4 20  D2 FF    JSR $FFD2
8EE7 A9  4E       LDA #$4E
8EE9 20  D2 FF    JSR $FFD2
8EEC D0  23       BNE $8F11
```

```
8EEE 29 7F      AND #$7F
8EF0 20 D2 FF   JSR $FFD2
8EF3 A9 25      LDA #$25
8EF5 20 D2 FF   JSR $FFD2
8EF8 20 33 8F   JSR $8F33
8EFB A0 00      LDY #$00
8EFD B1 BB      LDA ($BB),Y
8EFF 85 62      STA $62
8F01 C8         INY
8F02 B1 BB      LDA ($BB),Y
8F04 85 63      STA $63
8F06 A2 90      LDX #$90
8F08 20 44 BC   JSR $BC44
8F0B 20 DD BD   JSR $BDDD
8F0E 20 1E AB   JSR $AB1E
8F11 A9 0D      LDA #$0D
8F13 20 D2 FF   JSR $FFD2
8F16 18         CLC
8F17 A5 BB      LDA $BB
8F19 69 05      ADC #$05
8F1B 85 BB      STA $BB
8F1D 90 02      BCC $8F21
8F1F E6 BC      INC $BC
8F21 20 E4 FF   JSR $FFE4
8F24 20 E1 FF   JSR $FFE1
8F27 D0 01      BNE $8F2A
8F29 60         RTS
8F2A A5 CB      LDA $CB
8F2C C9 40      CMP #$40
8F2E D0 F1      BNE $8F21
8F30 38         SEC
8F31 B0 9B      BCS $8ECE
8F33 A9 3D      LDA #$3D
8F35 20 D2 FF   JSR $FFD2
8F38 18         CLC
8F39 A5 BB      LDA $BB
8F3B 69 02      ADC #$02
8F3D 85 BB      STA $BB
8F3F 90 02      BCC $8F43
8F41 E6 BC      INC $BC
8F43 60         RTS
8F44 90 03      BCC $8F49
8F46 4C 08 AF   JMP $AF08
8F49 20 F5 81   JSR $81F5
8F4C 20 13 A6   JSR $A613
8F4F B0 05      BCS $8F56
8F51 A2 15      LDX #$15
8F53 4C 37 A4   JMP $A437
8F56 A5 5F      LDA $5F
8F58 85 FB      STA $FB
8F5A A5 60      LDA $60
8F5C 85 FC      STA $FC
8F5E 20 79 00   JSR $0079
8F61 C9 2C      CMP #$2C
8F63 D0 E1      BNE $8F46
8F65 20 73 00   JSR $0073
8F68 D0 0F      BNE $8F79
8F6A 38         SEC
8F6B A5 2D      LDA $2D
8F6D E9 02      SBC #$02
8F6F 85 5F      STA $5F
8F71 A5 2E      LDA $2E
8F73 E9 00      SBC #$00
8F75 85 60      STA $60
8F77 D0 18      BNE $8F91
8F79 B0 CB      BCS $8F46
8F7B 20 F5 81   JSR $81F5
8F7E E6 14      INC $14
8F80 20 13 A6   JSR $A613
8F83 A5 FC      LDA $FC
8F85 C5 60      CMP $60
8F87 90 08      BCC $8F91
8F89 D0 BB      BNE $8F46
8F8B A5 FB      LDA $FB
8F8D C5 5F      CMP $5F
8F8F B0 B5      BCS $8F46
8F91 A0 00      LDY #$00
8F93 A5 5F      LDA $5F
8F95 91 FB      STA ($FB),Y
8F97 C8         INY
8F98 A5 60      LDA $60
8F9A 91 FB      STA ($FB),Y
8F9C C8         INY
8F9D B1 FB      LDA ($FB),Y
8F9F AA         TAX
8FA0 C8         INY
8FA1 B1 FB      LDA ($FB),Y
8FA3 20 7F 84   JSR $847F
8FA6 68         PLA
8FA7 68         PLA
8FA8 A2 FF      LDX #$FF
8FAA A9 01      LDA #$01
```

```
8FAC 4C 7D 92   JMP $927D
8FAF 20 F5 81   JSR $81F5
8FB2 4C A3 A8   JMP $A8A3
8FB5 A9 03      LDA #$03
8FB7 20 FB A3   JSR $A3FB
8FBA A5 7B      LDA $7B
8FBC 48         PHA
8FBD A5 7A      LDA $7A
8FBF 48         PHA
8FC0 A5 3A      LDA $3A
8FC2 48         PHA
8FC3 A5 3B      LDA $3B
8FC5 48         PHA
8FC6 A9 8D      LDA #$8D
8FC8 48         PHA
8FC9 20 79 00   JSR $0079
8FCC 20 AF 8F   JSR $8FAF
8FCF 4C AE A7   JMP $A7AE
8FD2 A2 00      LDX #$00
8FD4 C6 7A      DEC $7A
8FD6 B0 02      BCS $8FDA
8FD8 C6 7B      DEC $7B
8FDA 20 73 00   JSR $0073
8FDD F0 06      BEQ $8FE5
8FDF 9D 00 02   STA $0200,X
8FE2 E8         INX
8FE3 D0 F5      BNE $8FDA
8FE5 A9 00      LDA #$00
8FE7 9D 00 02   STA $0200,X
8FEA A9 03      LDA #$03
8FEC 20 FB A3   JSR $A3FB
8FEF A5 7B      LDA $7B
8FF1 48         PHA
8FF2 A5 7A      LDA $7A
8FF4 48         PHA
8FF5 A5 3A      LDA $3A
8FF7 48         PHA
8FF8 A5 39      LDA $39
8FFA 48         PHA
8FFB A9 8D      LDA #$8D
8FFD 48         PHA
8FFE A5 2B      LDA $2B
9000 85 FB      STA $FB
9002 A5 2C      LDA $2C
9004 85 FC      STA $FC
9006 A0 00      LDY #$00
9008 B1 FB      LDA ($FB),Y
900A 85 FD      STA $FD
900C C8         INY
900D B1 FB      LDA ($FB),Y
900F 85 FE      STA $FE
9011 D0 05      BNE $9018
9013 A2 11      LDX #$11
9015 4C 37 A4   JMP $A437
9018 A0 04      LDY #$04
901A B1 FB      LDA ($FB),Y
901C C9 E1      CMP #$E1
901E F0 0A      BEQ $902A
9020 A5 FD      LDA $FD
9022 85 FB      STA $FB
9024 A5 FE      LDA $FE
9026 85 FC      STA $FC
9028 D0 DC      BNE $9006
902A A2 FF      LDX #$FF
902C E8         INX
902D C8         INY
902E B1 FB      LDA ($FB),Y
9030 F0 07      BEQ $9039
9032 DD 00 02   CMP $0200,X
9035 F0 F5      BEQ $902C
9037 D0 E7      BNE $9020
9039 DD 00 02   CMP $0200,X
903C D0 E2      BNE $9020
903E 38         SEC
903F A5 FB      LDA $FB
9041 E9 01      SBC #$01
9043 85 7A      STA $7A
9045 A5 FC      LDA $FC
9047 E9 00      SBC #$00
9049 85 7B      STA $7B
904B 4C AE A7   JMP $A7AE
904E D0 0C      BNE $905C
9050 A9 00      LDA #$00
9052 85 CB      STA $CB
9054 20 E4 FF   JSR $FFE4
9057 F0 FB      BEQ $9054
9059 85 90      STA $90
905B 60         RTS
905C 20 9E AD   JSR $AD9E
905F 20 A3 B6   JSR $B6A3
9062 C9 00      CMP #$00
9064 F0 EA      BEQ $9050
```

```
9066 85 FB       STA $FB
9068 A9 00       LDA #$00
906A 85 C6       STA $C6
906C 20 E4 FF    JSR $FFE4
906F F0 FB       BEQ $906C
9071 A4 FB       LDY $FB
9073 88          DEY
9074 D1 22       CMP ($22),Y
9076 F0 05       BEQ $907D
9078 88          DEY
9079 10 F9       BPL $9074
907B 30 EF       BMI $906C
907D 85 90       STA $90
907F 60          RTS
9080 20 D4 E1    JSR $E1D4
9083 A9 00       LDA #$00
9085 85 B9       STA $B9
9087 20 26 B5    JSR $B526
908A A5 2D       LDA $2D
908C 85 5F       STA $5F
908E A5 2E       LDA $2E
9090 85 60       STA $60
9092 38          SEC
9093 A5 31       LDA $31
9095 85 5A       STA $5A
9097 E5 2F       SBC $2F
9099 85 FD       STA $FD
909B A5 32       LDA $32
909D 85 5B       STA $5B
909F E5 30       SBC $30
90A1 85 FE       STA $FE
90A3 A5 33       LDA $33
90A5 38          SEC
90A6 E9 01       SBC #$01
90A8 85 58       STA $58
90AA A5 34       LDA $34
90AC E9 00       SBC #$00
90AE 85 59       STA $59
90B0 20 BF A3    JSR $A3BF
90B3 A5 37       LDA $37
90B5 85 41       STA $41
90B7 A5 38       LDA $38
90B9 85 42       STA $42
90BB A5 58       LDA $58
90BD 85 FB       STA $FB
90BF 85 37       STA $37
90C1 E6 59       INC $59
90C3 A5 59       LDA $59
90C5 85 FC       STA $FC
90C7 85 38       STA $38
90C9 A6 2B       LDX $2B
90CB A4 2C       LDY $2C
90CD A9 00       LDA #$00
90CF 20 D5 FF    JSR $FFD5
90D2 90 03       BCC $90D7
90D4 4C F9 E0    JMP $E0F9
90D7 20 B7 FF    JSR $FFB7
90DA 29 BF       AND #$BF
90DC F0 05       BEQ $90E3
90DE A2 1D       LDX #$1D
90E0 4C 37 A4    JMP $A437
90E3 86 2D       STX $2D
90E5 84 2E       STY $2E
90E7 86 5F       STX $5F
90E9 84 60       STY $60
90EB A5 FB       LDA $FB
90ED 85 5A       STA $5A
90EF A5 FC       LDA $FC
90F1 85 5B       STA $5B
90F3 38          SEC
90F4 A5 33       LDA $33
90F6 E9 01       SBC #$01
90F8 A8          TAY
90F9 A5 34       LDA $34
90FB E9 00       SBC #$00
90FD AA          TAX
90FE 98          TYA
90FF 38          SEC
9100 E5 5A       SBC $5A
9102 85 58       STA $58
9104 A8          TAY
9105 8A          TXA
9106 E5 5B       SBC $5B
9108 AA          TAX
9109 E8          INX
910A 98          TYA
910B F0 1F       BEQ $912C
910D A5 5A       LDA $5A
910F 18          CLC
9110 65 58       ADC $58
9112 85 5A       STA $5A
9114 90 03       BCC $9119
```

```
9116 E6 58       INC $58
9118 18          CLC
9119 A5 5F       LDA $5F
911B 65 58       ADC $58
911D 85 5F       STA $5F
911F 90 02       BCC $9123
9121 E6 60       INC $60
9123 98          TYA
9124 49 FF       EOR #$FF
9126 A8          TAY
9127 C8          INY
9128 C6 5B       DEC $5B
912A C6 60       DEC $60
912C B1 5A       LDA ($5A),Y
912E 91 5F       STA ($5F),Y
9130 C8          INY
9131 D0 F9       BNE $912C
9133 E6 5B       INC $5B
9135 E6 60       INC $60
9137 CA          DEX
9138 D0 F2       BNE $912C
913A 38          SEC
913B A5 5F       LDA $5F
913D 85 31       STA $31
913F E5 FD       SBC $FD
9141 85 2F       STA $2F
9143 A5 60       LDA $60
9145 85 32       STA $32
9147 E5 FE       SBC $FE
9149 85 30       STA $30
914B A5 41       LDA $41
914D 85 37       STA $37
914F A5 42       LDA $42
9151 85 38       STA $38
9153 68          PLA
9154 68          PLA
9155 20 33 A5    JSR $A533
9158 A9 00       LDA #$00
915A 20 90 FF    JSR $FF90
915D 20 E7 FF    JSR $FFE7
9160 20 77 A6    JSR $A677
9163 20 8E A6    JSR $A68E
9166 4C CF 92    JMP $92CF
9169 D0 03       BNE $916E
916B 4C 08 AF    JMP $AF08
916E 20 F5 81    JSR $81F5
```

```
9171 A5 15       LDA $15
9173 C9 04       CMP #$04
9175 B0 05       BCS $917C
9177 A2 0E       LDX #$0E
9179 4C 37 A4    JMP $A437
917C C9 80       CMP #$80
917E B0 F7       BCS $9177
9180 60          RTS
9181 20 69 91    JSR $9169
9184 85 38       STA $38
9186 A5 14       LDA $14
9188 85 37       STA $37
918A 4C 63 A6    JMP $A663
918D 20 69 91    JSR $9169
9190 A0 00       LDY #$00
9192 98          TYA
9193 91 14       STA ($14),Y
9195 C8          INY
9196 91 14       STA ($14),Y
9198 C8          INY
9199 91 14       STA ($14),Y
919B A5 14       LDA $14
919D 18          CLC
919E 69 01       ADC #$01
91A0 85 2B       STA $2B
91A2 AA          TAX
91A3 A5 15       LDA $15
91A5 69 00       ADC #$00
91A7 85 2C       STA $2C
91A9 A8          TAY
91AA 8A          TXA
91AB 69 02       ADC #$02
91AD 85 2D       STA $2D
91AF 98          TYA
91B0 69 00       ADC #$00
91B2 85 2E       STA $2E
91B4 4C 63 A6    JMP $A663
91B7 A9 7C       LDA #$7C
91B9 8D 04 03    STA $0304
91BC A9 A5       LDA #$A5
91BE 8D 05 03    STA $0305
91C1 A9 1A       LDA #$1A
91C3 8D 06 03    STA $0306
91C6 A9 E4       LDA #$E4
91C8 8D 08 03    STA $0308
91CB A9 A7       LDA #$A7
```

```
91CD 8D 07 03    STA $0307
91D0 8D 09 03    STA $0309
91D3 A9 86       LDA #$86
91D5 8D 0A 03    STA $030A
91D8 A9 AE       LDA #$AE
91DA 8D 0B 03    STA $030B
91DD A9 FE       LDA #$FE
91DF 8D 17 03    STA $0317
91E2 8D 19 03    STA $0319
91E5 A9 66       LDA #$66
91E7 8D 16 03    STA $0316
91EA A9 47       LDA #$47
91EC 8D 18 03    STA $0318
91EF 78          SEI
91F0 A9 48       LDA #$48
91F2 8D 8F 02    STA $028F
91F5 A9 EB       LDA #$EB
91F7 8D 90 02    STA $0290
91FA 58          CLI
91FB 68          PLA
91FC 68          PLA
91FD 4C 74 A4    JMP $A474
9200 20 63 A6    JSR $A663
9203 A9 93       LDA #$93
9205 20 D2 FF    JSR $FFD2
9208 A9 00       LDA #$00
920A 8D 20 D0    STA $D020
920D 8D 21 D0    STA $D021
9210 A9 05       LDA #$05
9212 8D 86 02    STA $0286
9215 A2 0A       LDX #$0A
9217 A0 09       LDY #$09
9219 20 58 92    JSR $9258
921C A2 0C       LDX #$0C
921E A0 09       LDY #$09
9220 18          CLC
9221 20 F0 FF    JSR $FFF0
9224 A2 15       LDX #$15
9226 BD 67 92    LDA $9267,X
9229 20 D2 FF    JSR $FFD2
922C CA          DEX
922D 10 F7       BPL $9226
922F A2 0E       LDX #$0E
9231 A0 09       LDY #$09
9233 20 58 92    JSR $9258
9236 8D C1 05    STA $05C1
9239 8D 11 06    STA $0611
923C 8D D6 05    STA $05D6
923F 8D 26 06    STA $0626
9242 A9 05       LDA #$05
9244 8D C1 D9    STA $D9C1
9247 8D 11 DA    STA $DA11
924A 8D D6 D9    STA $D9D6
924D 8D 26 DA    STA $DA26
9250 A9 0D       LDA #$0D
9252 20 D2 FF    JSR $FFD2
9255 4C 74 A4    JMP $A474
9258 18          CLC
9259 20 F0 FF    JSR $FFF0
925C A9 2A       LDA #$2A
925E A2 16       LDX #$16
9260 20 D2 FF    JSR $FFD2
9263 CA          DEX
9264 D0 FA       BNE $9260
9266 60          RTS
9267 2A          ROL
9268 20 59 54    JSR $5459
926B 49 4C       EOR #$4C
926D 49 54       EOR #$54
926F 55 20       EOR $20,X
9271 43          ???
9272 49 53       EOR #$53
9274 41 42       EOR ($42,X)
9276 20 20 4E    JSR $4E20
9279 43          ???
927A 50 20       BVC $929C
927C 2A          ROL
927D AD 02 03    LDA $0302
9280 8D 97 92    STA $9297
9283 AD 03 03    LDA $0303
9286 8D 9C 92    STA $929C
9289 A9 96       LDA #$96
928B 8D 02 03    STA $0302
928E A9 92       LDA #$92
9290 8D 03 03    STA $0303
9293 4C 86 A4    JMP $A486
9296 A9 83       LDA #$83
9298 8D 02 03    STA $0302
929B A9 A4       LDA #$A4
929D 8D 03 03    STA $0303
92A0 20 33 A5    JSR $A533
92A3 18          CLC
```

```
92A4 A5 22        LDA $22
92A6 69 02        ADC #$02
92A8 85 2D        STA $2D
92AA A5 23        LDA $23
92AC 69 00        ADC #$00
92AE 85 2E        STA $2E
92B0 20 60 A6     JSR $A660
92B3 4C 74 A4     JMP $A474
92B6 A5 32        LDA $32
92B8 C5 2E        CMP $2E
92BA D0 09        BNE $92C5
92BC A5 31        LDA $31
92BE C5 2D        CMP $2D
92C0 D0 03        BNE $92C5
92C2 A9 80        LDA #$80
92C4 2C A9 00     BIT $00A9
92C7 85 0C        STA $0C
92C9 20 79 00     JSR $0079
92CC 4C 80 90     JMP $9080
92CF A5 0C        LDA $0C
92D1 10 04        BPL $92D7
92D3 C6 30        DEC $30
92D5 C6 32        DEC $32
92D7 4C AE A7     JMP $A7AE
```

Loading the utility

Once the UTILITY and the data for CODER have been set up, a loader program something like the following should be used.

```
10 A=A+1:IF A=1 THEN LOAD"UTILITY DATA",8,1
20 IF A=2 THEN LOAD"UTILITY",8,1
30 SYS32768
```

10 Bits 'n' pieces

General

This chapter is a collection of snippets of information we have found out since acquiring our 64s about 18 months ago. No detailed code here, just the bare facts and a few ideas.

AUTO-REPEATS and INTERRUPTS

We have seen two articles on the subject of providing a repeat on all keys. Both articles were based upon the same idea used on the pre-8000 series PETS. In essence the normal IRQ service routine is patched to include additional code by changing the vector at CINV from its default of $EA31. The additional routine simple scans SFDX – if a key is being pressed then it is reset to no key ($40) – and ends with a JUMP to $EA31 to process the normal interrupt. This will then detect a key as being pressed and enter the appropriate character in the keyboard buffer. Alternatively, for a repeat on all keys, simply POKE 650,128. To disable the repeat, POKE 650,1. (For a full description of the IRQ service routine, see Chapter 4.)

The second method is obviously far easier, but the first does allow a selective auto-repeat to be implemented.

Whilst on the subject of the hardware interrupt (see Chapter 4 for its implementation in the KEY commands), here is a short example to demonstrate what can be done. The following program patches IRQ to scan for function keys 1 and 3. These keys are used to increment the border and background colours respectively. The routine only takes the appropriate action once every 60 interrupts (about a second). If you remove the interrupt counter from $C01F to $C02D, the effects produced are quite unusual, but the routine becomes of little practical use as it is too fast to exercise selective control.

To enable: SYS 49152
To disable: SYS 49170

```
C000  78       SEI          ENABLE ENTRY
C001  A91F     LDA  #$1F    SET CINV TO POINT TO $C01F
C003  8D1403   STA  $0314
C006  A9C0     LDA  #$C0
```

```
C008  8D1503   STA  $0315
C00B  A900     LDA  #$00   SET IRQ COUNTER TO ZERO
C00D  8D00C1   STA  $C100
C010  58       CLI
C011  60       RTS
C012  78       SEI         DISABLE ENTRY
C013  A931     LDA  #$31   RESTORE CINV TO $EA31
C015  8D1403   STA  $0314
C018  A9EA     LDA  #$EA
C01A  8D1503   STA  $0315
C01D  58       CLI
C01E  60       RTS
C01F  EE00C1   INC  $C100  NEW IRQ ENRTY
C022  AD00C1   LDA  $C100
C025  C93C     CMP  #$3C   60 INTERRUPTS ???
C027  D02A     BNE  $C053  NO - SKIP KEY SCAN
C029  A900     LDA  #$00   YES - SO RESET COUNTER
C02B  8D00C1   STA  $C100
C02E  A5CB     LDA  $CB    SFDX - CURRENT KEY PRESS
C030  18       CLC
C031  C904     CMP  #$04   F1 ???
C033  D00F     BNE  $C044
C035  18       CLC
C036  AD20D0   LDA  $D020  BDR COLOUR
C039  290F     AND  #$0F   BITS 0-3 ONLY (0-15 DEC)
C03B  6901     ADC  #$01   INCREMENT IT
C03D  8D20D0   STA  $D020
C040  A900     LDA  #$00   ENSURE SKIP TAKEN
C042  F00F     BEQ  $C053  SKIP BKD COLOUR
C044  C905     CMP  #$05   BKD COLOUR
C046  D00B     BNE  $C053
C048  18       CLC
C049  AD21D0   LDA  $D021
C04C  290F     AND  #$0F
C04E  6901     ADC  #$01
C050  8D21D0   STA  $D021
C053  4C31EA   JMP  $EA31  CONTINUE NORMAL IRQ
```

The BASIC loader:

```
1 DATA 120, 169, 31, 141, 20, 3, 169, 19
2, 141, 21, 3
2 DATA 169, 0, 141, 0, 193, 88, 96, 120,
 169, 49
3 DATA 141, 20, 3, 169, 234, 141, 21, 3,
 88, 96
```

```
4 DATA 238, 0, 193, 173, 0, 193, 201, 60
, 208, 42
5 DATA 169, 0, 141, 0, 193, 165, 203, 24
, 201, 4
6 DATA 208, 15, 24, 173, 32, 208, 41, 15
, 105, 1
7 DATA 141, 32, 208, 169, 0, 240, 15, 20
1, 5, 208
8 DATA 11, 24, 173, 33, 208, 41, 15, 105
, 1, 141
9 DATA 33, 208, 76, 49, 234, 237, 61, 3,
 170, 173
10 FOR I=49152 TO 49238:READ A:POKE I,A:
NEXT I
```

The IRQ vector can be used to great advantage. One common use is to provide interrupt driven music (see *The Companion to the Commodore 64* pub. by Pan/PCN) and as in the UTILITY to make the function keys programmable (Chapter 4).

Simple program protection

Some BASICS include commands to 'unlist' or generate protected files; the 64's, however, does not. In order to protect our software we have to resort to programming tricks.

There are many ways to afford a program some degree of protection from unauthorized change. Most of these are well known and do little to prevent the experienced user from gaining access. Chapter 1 showed how the link addresses could be modified to make program lines invisible and list out of sequence. Another way to hide areas of code is to end lines with a REM"[DEL][DEL] . . . . ." sequence. On listing, the 'deletes' will erase characters to their left. Most other techniques require a program to be RUN.

Once a program has been run we can destroy some of the vectors from $0300 to $0333. These include the PRINT TOKENS LINK, IQPLOP, which could be directed to, say, print 'SYNTAX ERROR' at $AF08, the SAVE vector at ISAVE to prevent saving and also disable the RUN/STOP at ISTOP. We could also put a specified value somewhere in memory which, if not found, will cause the program to crash, erase itself or even perform a cold boot of the system. Unfortunately (or perhaps fortunately) any BASIC program can be loaded without being run. To produce programs which auto-run on loading requires knowledge of both machine code and the operating system of the 64 (see CHAIN in Chapter 8). Nearly all commercial software uses a number of levels of protection, one of which is usually auto-running. We have covered a number of ways to accomplish this for your own software in Chapter 2, but purposely leave out many of the techniques used by commercial software houses. (Remember it is illegal to reproduce commercial software.)

Commodore Computing International Volume 2 No.II has an article on program protection. It contains the usual:

DISABLE RUN/STOP	POKE 808,251
DISABLE LIST	POKE 774,131:POKE 775,164
DISABLE SAVE	POKE 818,131:POKE 819,164

The first simply bypasses the test by jumping to the end of the routine (RTS). The latter two jump to 'ready for BASIC'. Similar changes may be made to RUN/RESTORE at NMINV (see Chapter 6). The article does give a program to generate auto-run programs from your own code. If you are interested then, as the program is copyrighted, we suggest you get hold of a copy of the magazine. A further tape protection idea was given in *Commodore User* Volume 1 No.10.

Specified input

One of the most difficult and time-consuming tasks in producing software for use by others is in making it 'crash-proof'. BASIC does not allow the programmer to specify which keys are valid during input. The results of incorrect entries in type, size or number can spoil a well-thought-out, pleasing display or even crash the program. The way round this problem is to write your own input routine.

Commodore Horizons magazine of February 1984 published a very good machine code 'Keyscan' input program written by Adrian Warman which does just about everything you could ask for. We see little point in re-inventing the wheel, so we suggest that you read that article. However, we have approached the problem from a different angle and produced a simple routine entirely in BASIC. This is intended to be called when input is required. The type of input expected is set using the variable 'F' which is set to 0 for a real number, 1 for an integer and 2 for a string. Strings may contain commas and quotes. Editing an input may only be carried by using the DELETE key. The returned value may, if required, be converted to a number by a simple VAL(). If the routine is to be used more than once, A$ must be emptied by: A$="" before each use.

60000– Generate a flashing cursor.
60030 If it is a delete check chars are there to be removed.
60040 'Return' marks end of input and the resulting string A$ is passed back to the main program.
60050 Real numbers.
60060 Integer only.
60070 String.
60090 Update the display and wait for next char.

60110– Real numbers may begin with + or – and may contain only numerals and a decimal point.

60170– As for real but may not have a decimal point and must lie in the range given.

60230– Allows any of the standard alphanumerics. To provide for lower case mode where uppercase characters have their high bit set some graphics are permitted (128+32 to 128+64).

```
60000 POKE 204,0:POKE 207,0
60010 GET Y$:IF Y$="" GOTO 60000
60020 A=LEN(A$)
60030 IF Y$=CHR$(20) AND A>0 THEN A$=LEF
T$(A$,A-1):GOTO 60090
60040 IF Y$=CHR$(13) GOTO 60100
60050 IF F=0 THEN GOSUB 60110:GOTO 60090
60060 IF F=1 THEN GOSUB 60170:GOTO 60090
60070 IF F=2 THEN GOSUB 60230:GOTO 60090
60080 GOTO 60000
60090 PRINT Y$;:GOTO 60000
60100 RETURN
60110 REM REAL
60120 IF Y$="+" OR  Y$="-" AND A=0 THEN
A$=Y$:GOTO 60160
60130 IF Y$>"/" AND Y$<":" THEN A$=A$+Y$
:GOTO 60160
60140 IF Y$="." THEN A$=A$+Y$:GOTO 60160
60150 Y$=""
60160 RETURN
60170 REM INTEGER
60180 IF Y$="+" OR  Y$="-" AND A=0 THEN
A$=Y$:GOTO 60220
60190 IF Y$>"/" AND Y$<":" THEN A$=A$+Y$
:GOTO 60220
60200 IF VAL(A$)>32767 OR VAL(A$)<-32768
 THEN A$=LEFT$(A$,A):GOTO 60220
60210 Y$=""
60220 RETURN
60230 REM STRING
60240 IF (ASC(Y$) AND 127)<32 OR (ASC(Y$
) AND 127)>95 THEN Y$="":GOTO 60260
60250 A$=A$+Y$
60260 RETURN
```

The above is intended only as a starting point. Obvious improvements would be to allow the use of the cursor keys by manipulating the string with LEFT$, MID$ and RIGHT$. The maximum length of the string field could also be set to prevent overwriting an existing display.

Invisible characters

Most readers will no doubt be aware that character data may be directly POKEd to the screen. They will also have discovered that on occasions no effect is apparent. A screen character (normal mode) takes on the colour set in the corresponding location of the colour map ($D800 on). If no character has been printed at this location nor a colour set since the last clear screen the adopted colour is that of the background. To see if the character is there, simply move the cursor to the location.

We can use this to good effect by making displays change quite quickly from BASIC with just pokes to the colour map. It is important to remember that on an INPUT even though the character cannot be seen it is still there and active.

PET–64/64–PET

Commodore has maintained almost complete compatibility in the storage of programs on tape and to a lesser degree on disk. A tape prepared on any machine may be read by another. The 1541 disk drive uses an identical format to the 4040 unit. It can also read 2040 and 3040 formatted disks, but will corrupt these disks if it writes to them. You may also find that you get write problems on 4040 formatted disks. This compatibility does not mean to say that a program written for one machine will work directly on another.

A BASIC program saved on a PET can be loaded and run directly on the 64, whereas the reverse is not true. A word of warning to cassette users that the secondary address of 3 available on the 64 is not recognised by the PET. This is due to the different start address of BASIC on the two machines ($0401 on PET). Loading a program with a secondary address of zero will not allow it to load below the current start of BASIC. This means that on the 64 the load will be forced to $0801. A 64 program will normally have a start greater than $0401 and will go in above the start of BASIC and is not directly usable.

There are two ways to overcome the problem. The first is raise the start of BASIC on the PET to $0801 by (BASIC 2 & 4)

```
POKE 41,08:POKE 2048,0:NEW
```

before loading. All pokes to the screen will have to be adjusted for the PET screen which starts at $8000 (32768) and all pokes to the colour map removed. Defining the start of the screen with a variable and using offsets from this simplifies the conversion. The easiest way is to avoid using anything other than PRINT for output.

The second technique involves configuring the 64 to look like a PET – BASIC at $0401 and the screen in bank 2 at $8000. The *Programmer's Reference Guide* (Chapter 3, 'Screen Memory') tells us how to relocate the screen by setting bits 7 to 4 of $D018 (53272), remembering to tell the screen editor where it has gone by setting HIBASE ($0288/648). The start of BASIC is lowered by setting TXTTAB and the top of memory set using MEMTOP and string storage with FRETOP. The following program if run will carry out the necessary changes. SYNTAX ERRORS may result until the screen is cleared due to invisible characters.

```
10 POKE 51,000:POKE 55,000
20 POKE 52,128:POKE 56,128
30 POKE 56578,PEEK(56578) OR 3:POKE56576
,(PEEK(56576)AND252) OR 1
40 POKE 53272,(PEEK(53272)AND15)OR0:POKE
 648,128:POKE 1024,0:POKE 44,4:NEW
```

There are some very sophisticated PET emulators on the market and even cross-assemblers for machine code and cross-compilers for BASIC. If a lot of your work is in an area where portability is important, it might well be worthwhile pursuing the matter.

Load and run

Pressing SHIFT and RUN/STOP will load and run the first program on tape providing it is in BASIC. This may also be performed by:

POKE 631,131:POKE 198,1

The advantage of the second is that it can be used from within a program to avoid the problems associated with chaining if variables are not to be retained. Less well known is its use with disk. The format is:

LOAD"PROG",8:[Press SHIFT & RUN/STOP]

Disk bugs

When using sequential data files problems may be encountered if the same logical file number is used for both read and write operations. Typically, error 63 FILE EXISTS is reported. The only way to be sure is to use different numbers for input and output.

A less annoying feature is that null strings written to a data file are ignored on reading back. One way to overcome the problem is to always default null strings to a set value which is recognized on reading back. Alternatively, GET# may be used to pick up returns and commas (a bit laborious).

Appendices

APPENDIX A: Storage of BASIC text

Standard CBM 64 tokens

hex	dec		hex	dec		hex	dec		hex	dec	
$20	32	sp	$40	64	@	$80	128	END	$A6	166	SPC(
$21	33	!	$41	65	A	$81	129	FOR	$A7	167	THEN
$22	34	"	$42	66	B	$82	130	NEXT	$A8	168	NOT
$23	35	#	$43	67	C	$83	131	DATA	$A9	169	STEP
$24	36	$	$44	68	D	$84	132	INPUT#	$AA	170	+ add
$25	37	%	$45	69	E	$85	133	INPUT	$AB	171	− minus
$26	38	&	$46	70	F	$86	134	DIM	$AC	172	* multi
$27	39	'	$47	71	G	$87	135	READ	$AD	173	/ div
$28	40	(	$48	72	H	$88	136	LET	$AE	174	↑ power
$29	41	)	$49	73	I	$89	137	GOTO	$AF	175	AND
$2A	42	*	$4A	74	J	$8A	138	RUN	$B0	176	OR
$2B	43	+	$4B	75	K	$8B	139	IF	$B1	177	> gt
$2C	44	,	$4C	76	L	$8C	140	RESTORE	$B2	178	= eq
$2D	45	−	$4D	77	M	$8D	141	GOSUB	$B3	179	< lt
$2E	46	.	$4E	78	N	$8E	142	RETURN	$B4	180	SGN
$2F	47	/	$4F	79	O	$8F	143	REM	$B5	181	INT
$30	48	0	$50	80	P	$90	144	STOP	$B6	182	ABS
$31	49	1	$51	81	Q	$91	145	ON	$B7	183	USR
$32	50	2	$52	82	R	$92	146	WAIT	$B8	184	FRE
$33	51	3	$53	83	S	$93	147	LOAD	$B9	185	POS
$34	52	4	$54	84	T	$94	148	SAVE	$BA	186	SQR
$35	53	5	$55	85	U	$95	149	VERIFY	$BB	187	RND
$36	54	6	$56	86	V	$96	150	DEF	$BC	188	LOG
$37	55	7	$57	87	W	$97	151	POKE	$BD	189	EXP
$38	56	8	$58	88	X	$98	152	PRINT#	$BE	190	COS
$39	57	9	$59	89	Y	$99	153	PRINT	$BF	191	SIN
$3A	58	:	$5A	90	Z	$9A	154	CONT	$C0	192	TAN
$3B	59	;	$5B	91	[	$9B	155	LIST	$C1	193	ATN
$3C	60	<	$5C	92	£	$9C	156	CLR	$C2	194	PEEK
$3D	61	=	$5D	93	]	$9D	157	CMD	$C3	195	LEN
$3E	62	>	$5E	94	↑	$9E	158	SYS	$C4	196	STR$
$3F	63	?	$5F	95	←	$9F	159	OPEN	$C5	197	VAL
						$A0	160	CLOSE	$C6	198	ASC
						$A1	161	GET	$C7	199	CHR$
						$A2	162	NEW	$C8	200	LEFT$
						$A3	163	TAB(	$C9	201	RIGHT$
						$A4	164	TO	$CA	202	MID$
						$A5	165	FN	$CB	203	GO

Extended basic tokens

hex	dec		hex	dec	
$CC	204	OFF	$E6	230	RESET
$CD	205	KEY	$E7	231	CHAIN
$CE	206	DOKE	$E8	232	LOMEM
$CF	207	TEN	$E9	233	HIMEM
$D0	208	TWO	$EA	234	INKEY$
$D1	209	HEX	$EB	235	MEM
$D2	210	BIN	$EC	236	APPEND
$D3	211	OLD	$ED	237	TROFF
$D4	212	COLOUR	$EE	238	unused
$D5	213	WRITE	$EF	239	unused
$D6	214	CGOTO	$F0	240	unused
$D7	215	CGOSUB	$F1	241	unused
$D8	216	PLOT	$F2	242	unused
$D9	217	ENTER	$F3	243	unused
$DA	218	DUMP	$F4	244	unused
$DB	219	RENUM	$F5	245	unused
$DC	220	DELETE	$F6	246	unused
$DD	221	MERGE	$F7	247	DEEK
$DE	222	CODER	$F8	248	unused
$DF	223	AUTO	$F9	249	unused
$E0	224	PROC	$FA	250	unused
$E1	225	DPROC	$FB	251	unused
$E2	256	EPROC	$FC	252	unused
$E3	227	POP	$FD	253	unused
$E4	228	QUIT	$FE	254	unused
$E5	229	TRACE			

APPENDIX B: Hex to decimal and decimal to hex converter

hex	decimal		hex	decimal		hex	decimal		hex	decimal	
	low	high		low	high		low	high		low	high
$00	0	0	$29	41	10496	$52	82	20992	$7B	123	31488
$01	1	256	$2A	42	10752	$53	83	21248	$7C	124	31744
$02	2	512	$2B	43	11008	$54	84	21504	$7D	125	32000
$03	3	768	$2C	44	11264	$55	85	21760	$7E	126	32256
$04	4	1024	$2D	45	11520	$56	86	22016	$7F	127	32512
$05	5	1280	$2E	46	11776	$57	87	22272	$80	128	32768
$06	6	1536	$2F	47	12032	$58	88	22528	$81	129	33024
$07	7	1792	$30	48	12288	$59	89	22784	$82	130	33280
$08	8	2048	$31	49	12544	$5A	90	23040	$83	131	33536
$09	9	2304	$32	50	12800	$5B	91	23296	$84	132	33792
$0A	10	2560	$33	51	13056	$5C	92	23552	$85	133	34048
$0B	11	2816	$34	52	13312	$5D	93	23808	$86	134	34304
$0C	12	3072	$35	53	13568	$5E	94	24064	$87	135	34560
$0D	13	3328	$36	54	13824	$5F	95	24320	$88	136	34816
$0E	14	3584	$37	55	14080	$60	96	24576	$89	137	35072
$0F	15	3840	$38	56	14336	$61	97	24832	$8A	138	35328
$10	16	4096	$39	57	14592	$62	98	25088	$8B	139	35584
$11	17	4352	$3A	58	14848	$63	99	25344	$8C	140	35840
$12	18	4608	$3B	59	15104	$64	100	25600	$8D	141	36096
$13	19	4864	$3C	60	15360	$65	101	25856	$8E	142	36352
$14	20	5120	$3D	61	15616	$66	102	26112	$8F	143	36608
$15	21	5376	$3E	62	15872	$67	103	26368	$90	144	36864
$16	22	5632	$3F	63	16128	$68	104	26624	$91	145	37120
$17	23	5888	$40	64	16384	$69	105	26880	$92	146	37376
$18	24	6144	$41	65	16640	$6A	106	27136	$93	147	37632
$19	25	6400	$42	66	16896	$6B	107	27392	$94	148	37888
$1A	26	6656	$43	67	17152	$6C	108	27648	$95	149	38144
$1B	27	6912	$44	68	17408	$6D	109	27904	$96	150	38400
$1C	28	7168	$45	69	17664	$6E	110	28160	$97	151	38656
$1D	29	7424	$46	70	17920	$6F	111	28416	$98	152	38912
$1E	30	7680	$47	71	18176	$70	112	28672	$99	153	39168
$1F	31	7936	$48	72	18432	$71	113	28928	$9A	154	39424
$20	32	8192	$49	73	18688	$72	114	29184	$9B	155	39680
$21	33	8448	$4A	74	18944	$73	115	29440	$9C	156	39936
$22	34	8704	$4B	75	19200	$74	116	29696	$9D	157	40192
$23	35	8960	$4C	76	19456	$75	117	29952	$9E	158	40448
$24	36	9216	$4D	77	19712	$76	118	30208	$9F	159	40704
$25	37	9472	$4E	78	19968	$77	119	30464	$A0	160	40960
$26	38	9728	$4F	79	20224	$78	120	30720	$A1	161	41216
$27	39	9984	$50	80	20480	$79	121	30976	$A2	162	41472
$28	40	10240	$51	81	20736	$7A	122	31232	$A3	163	41728

hex	decimal		hex	decimal		hex	decimal		hex	decimal	
	low	high		low	high		low	high		low	high
$A4	164	41984	$BB	187	47872	$D2	210	53760	$E9	233	59648
$A5	165	42240	$BC	188	48128	$D3	211	54016	$EA	234	59904
$A6	166	42496	$BD	189	48384	$D4	212	54272	$EB	235	60160
$A7	167	42752	$BE	190	48640	$D5	213	54528	$EC	236	60416
$A8	168	43008	$BF	191	48896	$D6	214	54784	$ED	237	60672
$A9	169	43264	$C0	192	49152	$D7	215	55040	$EE	238	60928
$AA	170	43520	$C1	193	49408	$D8	216	55296	$EF	239	61184
$AB	171	43776	$C2	194	49664	$D9	217	55552	$F0	240	61440
$AC	172	44032	$C3	195	49920	$DA	218	55808	$F1	241	61696
$AD	173	44288	$C4	196	50176	$DB	219	56064	$F2	242	61952
$AE	174	44544	$C5	197	50432	$DC	220	56320	$F3	243	62208
$AF	175	44800	$C6	198	50688	$DD	221	56576	$FA	244	62464
$B0	176	45056	$C7	199	50944	$DE	222	56832	$F5	245	62720
$B1	177	45312	$C8	200	51200	$DF	223	57088	$F6	246	62976
$B2	178	45568	$C9	201	51456	$E0	224	57344	$F7	247	63232
$B3	179	45824	$CA	202	51712	$E1	225	57600	$F8	248	63488
$B4	180	46080	$CB	203	51968	$E2	226	57856	$F9	249	63744
$B5	181	46336	$CC	204	52224	$E3	227	58112	$FA	250	64000
$B6	182	46592	$CD	205	52480	$E4	228	58368	$FB	251	64256
$B7	183	46848	$CE	206	52736	$E5	229	58624	$FC	252	64512
$B8	184	47104	$CF	207	52992	$E6	230	58880	$FD	253	64768
$B9	185	47360	$D0	208	53248	$E7	231	59136	$FE	254	65024
$BA	186	47616	$D1	209	53504	$E8	232	59392	$FF	255	65280

APPENDIX C: Machine code mnemonics and hex values

6510 OP-CODES

The tables below are a quick reference guide only and for more detailed information a 6502 assembler book should be consulted.

The tables should be read row then column. If in doubt, remember LDA immediate mode is $A9. The following abbreviations have been used:

– immediate mode
$ – absolute address
Z – zero page
I – indirect address

A=accumulator
X=X index register
Y=Y index register

0	1	2	4	5	6
0 BRK	ORA (I,X)			ORA Z	ASL Z
1 BPL	ORA (I),Y			ORA Z,X	ASL Z,X
2 JSR	AND (I,X)		BIT Z	AND Z	ROL Z
3 BMI	AND (I),Y			AND Z,X	ROL Z,X
4 RTI	EOR (I,X)			EOR Z	LSR Z
5 BVC	EOR (I),Y			EOR Z,X	LSR Z,X
6 RTS	ADC (I,X)			ADC Z	ROR Z
7 BVS	ADC (I),Y			ADC Z,X	ROR Z,X
8	STA (I,X)		STY Z	STA Z	STX Z
9 BCC	STA (I),Y		STY Z,X	STA Z,X	STX Z,Y
A LDY#	LDA (I,X)	LDX #	LDY Z	LDA Z	LDX Z
B BCS	LDA (I),Y		LDY Z,X	LDA Z,X	LDX Z,Y
C CPY#	CMP (I,X)		CPY Z	CMP Z	DEC Z
D BNE	CMP (I),Y			CMP Z,X	DEC Z,X
E CPX #	SBC (I,X)		CPX Z	SBC Z	INC Z
F BEQ	SBC (I),Y			SBC Z,X	INC Z,X

8	9	A	C	D	E
0 PHP	ORA #	ASL A		ORA $	ASL $
1 CLC	ORA $,Y			ORA $,X	ASL $,X
2 PLP	AND #	ROL A	BIT $	AND $	ROL $
3 SEC	AND $,Y			AND $,X	ROL $,X
4 PHA	EOR #	LSR A	JMP $	EOR $	LSR $
5 CLI	EOR $,Y			EOR $,X	LSR $,X
6 PLA	ADC #	ROR A	JMP I	ADC $	ROR $
7 SEI	ADC $,Y			ADC $,X	ROR $,X
8 DEY		TXA	STY $	STA $	STX $
9 TYA	STA $,Y	TXS		STA $,X	
A TAY	LDA #	TAX	LDY $	LDA $	LDX $
B CLV	LDA $,Y	TSX	LDY $,X	LDA $,X	LDX $,Y
C INY	CMP #	DEX	CPY $	CMP $	DEC $
D CLD	CMP $,Y			CMP $,X	DEC $,X
E INX	SBC #	NOP	CPX $	SBC $	INC $
F SED	SBC $,Y			SBC $,X	INC $,X

APPENDIX D: BASIC loader for Supermon

This is the BASIC program to produce Jim Butterfield's Supermon monitor. Type it in and SAVE it before running. There are lots of numbers so it is easy to make a mistake.

We have put in some checksums to help isolate errors. Once loaded correctly you will be able to save the machine code version from the monitor itself. Instructions for using Supermon are given in Appendix E.

Supermon normally loads to the top of BASIC memory. We have modified it to sit at $C000 on to allow you to enter the code for the UTILITY. Once you have it up and running, you can use Supermon to modify itself to sit anywhere. There are addresses which require changing so we have included a relocater program after the loader.

The Loader

```
10 A=49152:C=0
20 READB:IFB=-1THEN40
30 POKEA,B:A=A+1:C=C+B:GOTO20
40 IFC=27914THEN60
50 PRINT"DATA ERROR IN 1000 - 1300":END
60 C=0
70 READB:IFB=-1THEN90
80 POKEA,B:A=A+1:C=C+B:GOTO70
90 IFC=26078THEN110
100 PRINT"DATA ERROR IN 1310 - 1600":END
110 C=0
120 READB:IFB=-1THEN140
130 POKEA,B:A=A+1:C=C+B:GOTO120
140 IFC=26897THEN160
150 PRINT"DATA ERROR IN 1610 - 1900":END
160 C=0
170 READB:IFB=-1THEN190
180 POKEA,B:A=A+1:C=C+B:GOTO170
190 IFC=28055THEN210
200 PRINT"DATA ERROR IN 1910 - 2200":END
210 C=0
220 READB:IFB=-1THEN240
230 POKEA,B:A=A+1:C=C+B:GOTO220
240 IFC=25343THEN260
250 PRINT"DATA ERROR IN 2210 - 2500":END
260 C=0
270 READB:IFB=-1THEN290
280 POKEA,B:A=A+1:C=C+B:GOTO270
290 IFC=25432THEN310
300 PRINT"DATA ERROR IN 2510 - 2800":END
310 C=0
320 READB:IFB=-1THEN340
330 POKEA,B:A=A+1:C=C+B:GOTO320
340 IFC=27324THEN360
350 PRINT"DATA ERROR IN 2810 - 3100":END
```

```
360 C=0
370 READB:IFB=-1THEN390
380 POKEA,B:A=A+1:C=C+B:GOTO370
390 IFC=25335THEN410
400 PRINT"DATA ERROR IN 3110 - 3400":END
410 C=0
420 READB:IFB=-1THEN440
430 POKEA,B:A=A+1:C=C+B:GOTO420
440 IFC=28057THEN460
450 PRINT"DATA ERROR IN 3410 - 3700":END
460 C=0
470 READB:IFB=-1THEN490
480 POKEA,B:A=A+1:C=C+B:GOTO470
490 IFC=20514THEN510
500 PRINT"DATA ERROR IN 3710 - 4000":END
510 C=0
520 READB:IFB=-1THEN540
530 POKEA,B:A=A+1:C=C+B:GOTO520
540 IFC=22061THEN560
550 PRINT"DATA ERROR IN 4010 - 4290":END
560 PRINT"DATA CORRECT":PRINT
570 PRINT"SYS49152 TO USE"
580 END
1000 DATA76,233,192,255,0,0,255
1010 DATA255,0,0,255,255,0,0
1020 DATA255,255,0,0,255,255,0
1030 DATA0,255,255,0,0,255,255
1040 DATA0,0,255,255,0,0,255
1050 DATA255,0,0,255,255,0,0
1060 DATA255,255,0,0,255,255,0
1070 DATA0,255,255,0,0,255,255
1080 DATA0,0,255,255,0,0,255
1090 DATA255,0,0,255,255,0,0
1100 DATA255,255,0,0,255,255,0
1110 DATA0,255,255,0,0,255,255
1120 DATA0,0,255,255,0,0,255
1130 DATA255,0,0,255,255,0,0
1140 DATA255,255,0,0,255,255,0
1150 DATA0,255,255,0,0,255,255
1160 DATA0,0,255,255,0,0,255
1170 DATA255,0,0,255,255,0,0
1180 DATA255,255,128,0,255,255,0
1190 DATA0,255,255,0,0,255,255
1200 DATA0,0,255,255,0,0,255
1210 DATA255,0,0,255,255,0,0
1220 DATA255,255,0,0,255,255,0
```

```
1230 DATA0,255,255,0,0,255,255
1240 DATA0,0,255,255,0,0,255
1250 DATA255,0,0,255,255,0,0
1260 DATA255,255,0,0,255,255,0
1270 DATA0,255,255,0,0,255,255
1280 DATA0,0,255,255,0,0,255
1290 DATA255,0,0,255,255,0,0
1300 DATA255,255,0,0,255,255,0,-1
1310 DATA0,255,255,0,0,255,255
1320 DATA0,0,255,255,0,0,255
1330 DATA255,0,169,160,133,56,173
1340 DATA230,200,141,22,3,173,231
1350 DATA200,141,23,3,169,128,32
1360 DATA144,255,0,216,104,141,62
1370 DATA2,104,141,61,2,104,141
1380 DATA60,2,104,141,59,2,104
1390 DATA170,104,168,56,138,233,2
1400 DATA141,58,2,152,233,0,141
1410 DATA57,2,186,142,63,2,32
1420 DATA87,198,162,66,169,42,32
1430 DATA87,195,169,82,208,52,230
1440 DATA193,208,6,230,194,208,2
1450 DATA230,38,96,32,207,255,201
1460 DATA13,208,248,104,104,169,154
1470 DATA32,210,255,169,0,133,38
1480 DATA162,13,169,46,32,87,195
1490 DATA169,159,32,210,255,32,62
1500 DATA193,201,46,240,249,201,32
1510 DATA240,245,162,14,221,183,200
1520 DATA208,12,138,10,170,189,199
1530 DATA200,72,189,198,200,72,96
1540 DATA202,16,236,76,237,195,165
1550 DATA193,141,58,2,165,194,141
1560 DATA57,2,96,169,8,133,29
1570 DATA160,0,32,84,198,177,193
1580 DATA32,72,195,32,51,193,198
1590 DATA29,208,241,96,32,136,195
1600 DATA144,11,162,0,129,193,193,-1
1610 DATA193,240,3,76,237,195,32
1620 DATA51,193,198,29,96,169,59
1630 DATA133,193,169,2,133,194,169
1640 DATA5,96,152,72,32,87,198
1650 DATA104,162,46,76,87,195,169
1660 DATA154,32,210,255,162,0,189
1670 DATA234,200,32,210,255,232,224
1680 DATA22,208,245,160,59,32,194
```

```
1690 DATA193,173,57,2,32,72,195
1700 DATA173,58,2,32,72,195,32
1710 DATA183,193,32,141,193,240,92
1720 DATA32,62,193,32,121,195,144
1730 DATA51,32,105,195,32,62,193
1740 DATA32,121,195,144,40,32,105
1750 DATA195,169,154,32,210,255,32
1760 DATA225,255,240,60,166,38,208
1770 DATA56,165,195,197,193,165,196
1780 DATA229,194,144,46,160,58,32
1790 DATA194,193,32,65,195,32,139
1800 DATA193,240,224,76,237,195,32
1810 DATA121,195,144,3,32,128,193
1820 DATA32,183,193,208,7,32,121
1830 DATA195,144,235,169,8,133,29
1840 DATA32,62,193,32,161,193,208
1850 DATA248,76,71,193,32,207,255
1860 DATA201,13,240,12,201,32,208
1870 DATA209,32,121,195,144,3,32
1880 DATA128,193,169,154,32,210,255
1890 DATA174,63,2,154,120,173,57
1900 DATA2,72,173,58,2,72,173,-1
1910 DATA59,2,72,173,60,2,174
1920 DATA61,2,172,62,2,64,169
1930 DATA154,32,210,255,174,63,2
1940 DATA154,108,2,160,160,1,132
1950 DATA186,132,185,136,132,183,132
1960 DATA144,132,147,169,64,133,187
1970 DATA169,2,133,188,32,207,255
1980 DATA201,32,240,249,201,13,240
1990 DATA56,201,34,208,20,32,207
2000 DATA255,201,34,240,16,201,13
2010 DATA240,41,145,187,230,183,200
2020 DATA192,16,208,236,76,237,195
2030 DATA32,207,255,201,13,240,22
2040 DATA201,44,208,220,32,136,195
2050 DATA41,15,240,233,201,3,240
2060 DATA229,133,186,32,207,255,201
2070 DATA13,96,108,48,3,108,50
2080 DATA3,32,150,194,208,212,169
2090 DATA154,32,210,255,169,0,32
2100 DATA239,194,165,144,41,16,208
2110 DATA196,76,71,193,32,150,194
2120 DATA201,44,208,186,32,121,195
2130 DATA32,105,195,32,207,255,201
2140 DATA44,208,173,32,121,195,165
```

```
2150 DATA193,133,174,165,194,133,175
2160 DATA32,105,195,32,207,255,201
2170 DATA13,208,152,169,154,32,210
2180 DATA255,32,242,194,76,71,193
2190 DATA165,194,32,72,195,165,193
2200 DATA72,74,74,74,74,32,96,-1
2210 DATA195,170,104,41,15,32,96
2220 DATA195,72,138,32,210,255,104
2230 DATA76,210,255,9,48,201,58
2240 DATA144,2,105,6,96,162,2
2250 DATA181,192,72,181,194,149,192
2260 DATA104,149,194,202,208,243,96
2270 DATA32,136,195,144,2,133,194
2280 DATA32,136,195,144,2,133,193
2290 DATA96,169,0,133,42,32,62
2300 DATA193,201,32,208,9,32,62
2310 DATA193,201,32,208,14,24,96
2320 DATA32,175,195,10,10,10,10
2330 DATA133,42,32,62,193,32,175
2340 DATA195,5,42,56,96,201,58
2350 DATA144,2,105,8,41,15,96
2360 DATA162,2,44,162,0,180,193
2370 DATA208,8,180,194,208,2,230
2380 DATA38,214,194,214,193,96,32
2390 DATA62,193,201,32,240,249,96
2400 DATA169,0,141,0,1,32,204
2410 DATA195,32,143,195,32,124,195
2420 DATA144,9,96,32,62,193,32
2430 DATA121,195,176,222,174,63,2
2440 DATA154,169,154,32,210,255,169
2450 DATA63,32,210,255,76,71,193
2460 DATA32,84,198,202,208,250,96
2470 DATA230,195,208,2,230,196,96
2480 DATA162,2,181,192,72,181,39
2490 DATA149,192,104,149,39,202,208
2500 DATA243,96,165,195,164,196,56,-1
2510 DATA233,2,176,14,136,144,11
2520 DATA165,40,164,41,76,51,196
2530 DATA165,195,164,196,56,229,193
2540 DATA133,30,152,229,194,168,5
2550 DATA30,96,32,212,195,32,105
2560 DATA195,32,229,195,32,12,196
2570 DATA32,229,195,32,47,196,32
2580 DATA105,195,144,21,166,38,208
2590 DATA100,32,40,196,144,95,161
2600 DATA193,129,195,32,5,196,32
```

```
2610 DATA51,193,208,235,32,40,196
2620 DATA24,165,30,101,195,133,195
2630 DATA152,101,196,133,196,32,12
2640 DATA196,166,38,208,61,161,193
2650 DATA129,195,32,40,196,176,52
2660 DATA32,184,195,32,187,195,76
2670 DATA125,196,32,212,195,32,105
2680 DATA195,32,229,195,32,105,195
2690 DATA32,62,193,32,136,195,144
2700 DATA20,133,29,166,38,208,17
2710 DATA32,47,196,144,12,165,29
2720 DATA129,193,32,51,193,208,238
2730 DATA76,237,195,76,71,193,32
2740 DATA212,195,32,105,195,32,229
2750 DATA195,32,105,195,32,62,193
2760 DATA162,0,32,62,193,201,39
2770 DATA208,20,32,62,193,157,16
2780 DATA2,232,32,207,255,201,13
2790 DATA240,34,224,32,208,241,240
2800 DATA28,142,0,1,32,143,195,-1
2810 DATA144,198,157,16,2,232,32
2820 DATA207,255,201,13,240,9,32
2830 DATA136,195,144,182,224,32,208
2840 DATA236,134,28,169,154,32,210
2850 DATA255,32,87,198,162,0,160
2860 DATA0,177,193,221,16,2,208
2870 DATA12,200,232,228,28,208,243
2880 DATA32,65,195,32,84,198,32
2890 DATA51,193,166,38,208,141,32
2900 DATA47,196,176,221,76,71,193
2910 DATA32,212,195,133,32,165,194
2920 DATA133,33,162,0,134,40,169
2930 DATA147,32,210,255,q69,154,32
2940 DATA210,255,169,22,133,29,32
2950 DATA106,197,32,202,197,133,193
2960 DATA132,194,198,29,208,242,169
2970 DATA145,32,210,255,76,71,193
2980 DATA160,44,32,194,193,32,84
2990 DATA198,32,65,195,32,84,198
3000 DATA162,0,161,193,32,217,197
3010 DATA72,32,31,198,104,32,53
3020 DATA198,162,6,224,3,208,18
3030 DATA164,31,240,14,165,42,201
3040 DATA232,177,193,176,28,32,194
3050 DATA197,136,208,242,6,42,144
3060 DATA14,189,42,200,32,165,198
```

```
3070 DATA189,48,200,240,3,32,165
3080 DATA198,202,208,213,96,32,205
3090 DATA197,170,232,208,1,200,152
3100 DATA32,194,197,138,134,28,32,-1
3110 DATA72,195,166,28,96,165,31
3120 DATA56,164,194,170,16,1,136
3130 DATA101,193,144,1,200,96,168
3140 DATA74,144,11,74,176,23,201
3150 DATA34,240,19,41,7,9,128
3160 DATA74,170,189,217,199,176,4
3170 DATA74,74,74,74,41,15,208
3180 DATA4,160,128,169,0,170,189
3190 DATA29,200,133,42,41,3,133
3200 DATA31,152,41,143,170,152,160
3210 DATA3,224,138,240,11,74,144
3220 DATA8,74,74,9,32,136,208
3230 DATA250,200,136,208,242,96,177
3240 DATA193,32,194,197,162,1,32
3250 DATA254,195,196,31,200,144,241
3260 DATA162,3,192,4,144,242,96
3270 DATA168,185,55,200,133,40,185
3280 DATA119,200,133,41,169,0,160
3290 DATA5,6,41,38,40,42,136
3300 DATA208,248,105,63,32,210,255
3310 DATA202,208,236,169,32,44,169
3320 DATA13,76,210,255,32,212,195
3330 DATA32,105,195,32,229,195,32
3340 DATA105,195,162,0,134,40,169
3350 DATA154,32,210,255,32,87,198
3360 DATA32,114,197,32,202,197,133
3370 DATA193,132,194,32,225,255,240
3380 DATA5,32,47,196,176,233,76
3390 DATA71,193,32,212,195,169,3
3400 DATA133,29,32,62,193,32,161,-1
3410 DATA193,208,248,165,32,133,193
3420 DATA165,33,133,194,76,70,197
3430 DATA197,40,240,3,32,210,255
3440 DATA96,32,212,195,32,105,195
3450 DATA142,17,2,162,3,32,204
3460 DATA195,72,202,208,249,162,3
3470 DATA104,56,233,63,160,5,74
3480 DATA110,17,2,110,16,2,136
3490 DATA208,246,202,208,237,162,2
3500 DATA32,207,255,201,13,240,30
3510 DATA201,32,240,245,32,208,199
3520 DATA176,15,32,156,195,164,193
```

```
3530 DATA132,194,133,193,169,48,157
3540 DATA16,2,232,157,16,2,232
3550 DATA208,219,134,40,162,0,134
3560 DATA38,240,4,230,38,240,117
3570 DATA162,0,134,29,165,38,32
3580 DATA217,197,166,42,134,41,170
3590 DATA188,55,200,189,119,200,32
3600 DATA185,199,208,227,162,6,224
3610 DATA3,208,25,164,31,240,21
3620 DATA165,42,201,232,169,48,176
3630 DATA33,32,191,199,208,204,32
3640 DATA193,199,208,199,136,208,235
3650 DATA6,42,144,11,188,48,200
3660 DATA189,42,200,32,185,199,208
3670 DATA181,202,208,209,240,10,32
3680 DATA184,199,208,171,32,184,199
3690 DATA208,166,165,40,197,29,208
3700 DATA160,32,105,195,164,31,240,-1
3710 DATA40,165,41,201,157,208,26
3720 DATA32,28,196,144,10,152,208
3730 DATA4,165,30,16,10,76,237
3740 DATA195,200,208,250,165,30,16
3750 DATA246,164,31,208,3,185,194
3760 DATA0,145,193,136,208,248,165
3770 DATA38,145,193,32,202,197,133
3780 DATA193,132,194,169,154,32,210
3790 DATA255,160,65,32,194,193,32
3800 DATA84,198,32,65,195,32,84
3810 DATA198,169,159,32,210,255,76
3820 DATA176,198,168,32,191,199,208
3830 DATA17,152,240,14,134,28,166
3840 DATA29,221,16,2,8,232,134
3850 DATA29,166,28,40,96,201,48
3860 DATA144,3,201,71,96,56,96
3870 DATA64,2,69,3,208,8,64
3880 DATA9,48,34,69,51,208,8
3890 DATA64,9,64,2,69,51,208
3900 DATA8,64,9,64,2,69,179
3910 DATA208,8,64,9,0,34,68
3920 DATA51,208,140,68,0,17,34
3930 DATA68,51,208,140,68,154,16
3940 DATA34,68,51,208,8,64,9
3950 DATA16,34,68,51,208,8,64
3960 DATA9,98,19,120,169,0,33
3970 DATA129,130,0,0,89,77,145
3980 DATA146,134,74,133,157,44,41
```

```
3990 DATA44,35,40,36,89,0,88
4000 DATA36,36,0,28,138,28,35,-1
4010 DATA93,139,27,161,157,138,29
4020 DATA35,157,139,29,161,0,41
4030 DATA25,174,105,168,25,35,36
4040 DATA83,27,35,36,83,25,161
4050 DATA0,26,91,91,165,105,36
4060 DATA36,174,174,168,173,41,0
4070 DATA124,0,21,156,109,156,165
4080 DATA105,41,83,132,19,52,17
4090 DATA165,105,35,160,216,98,90
4100 DATA72,38,98,148,136,84,68
4110 DATA200,84,104,68,232,148,0
4120 DATA180,8,132,116,180,40,110
4130 DATA116,244,204,74,114,242,164
4140 DATA138,0,170,162,162,116,116
4150 DATA116,114,68,104,178,50,178
4160 DATA0,34,0,26,26,38,38
4170 DATA114,114,136,200,196,202,38
4180 DATA72,68,68,162,200,58,59
4190 DATA82,77,71,88,76,83,84
4200 DATA70,72,68,80,44,65,66
4210 DATA194,53,194,204,193,247,193
4220 DATA86,194,137,194,244,194,12
4230 DATA195,62,196,146,196,192,196
4240 DATA56,197,91,198,138,198,172
4250 DATA198,70,193,255,192,237,192
4260 DATA13,32,32,32,80,67,32
4270 DATA32,83,82,32,65,67,32
4280 DATA88,82,32,89,82,32,83
4290 DATA80,108,239,255,254,221,222,-1
```

Relocation

If you need to relocate Supermon to another area of memory, the following program will assist. First load in SUPERMON and use the TRANSFER option to copy it to the desired location. Now run the program below and it changes the appropriate locations so that it will run at its new address.

```
10 PRINT"[CLS][REV]SUPERMON RELOCATION":
PRINT:PRINT
20 INPUT"NEW START ADDRESS - DECIMAL";AD
30 IFAD>40960THENPOKEAD+234,160:GOTO90
40 PRINT"DO YOU WANT IT PROTECTED FROM BASIC"
50 PRINT"Y OR N"
```

```
60 GETA$:IFA$<>"Y"ANDA$<>"N"THEN60
70 IF A$="N"THEN POKE AD,160:GOTO90
80 POKEAD+234,INT((AD/256+.5))
90 READ OF:IF OF=-1 THEN130
100 READ DA:DA=DA+AD:D2=INT(DA/256):D1=D
A-(D2*256)
110 POKE OF+AD,D1:POKE OF+AD+1,D2
120 GOTO90
130 END
140 CO=0
150 READ A:IF A=-1 THEN170
160 CO=CO+A:GOTO150
170 IF CO=451387THENPRINT "[REV]DATA OK"
:END
180 PRINT"[REV]DATA INCORRECT"
190 REM ***** START OF DATA *****
200 DATA1,233,238,2278,244,2279,294,1623
,301,855,341,855,349,318,370,2247
210 DATA374,2246,382,1005,402,1620,407,8
40,410,307,418,904,431,1005,434,307
220 DATA453,1623,459,855,469,2282,482,45
0,488,840,494,840,497,439,500,397
230 DATA505,318,508,889,513,873,516,318,
519,889,524,873,553,450,556,833,559,395
240 DATA564,1005,567,889,572,384,575,439
,580,889,589,318,592,417,597,327,611,889
250 DATA616,384,719,1005,733,904,758,662
,770,751,779,327,782,662,789,889
260 DATA792,873,802,889,813,873,828,754,
831,327,836,840,846,864,853,864
270 DATA890,904,897,904,909,318,925,943,
934,318,937,943,973,318,986,972
280 DATA989,911,992,892,998,318,1001,889
,1020,327,1023,1620,1069,1075
290 DATA1088,980,1091,873,1094,997,1097,
1036,1100,997,1103,1071,1106,873
300 DATA1115,1064,1124,1029,1127,307,113
2,1064,1147,1036,1158,1064
310 DATA1163,952,1166,955,1169,1149,1172
,980,1175,873,1178,997,1181,873
320 DATA1184,318,1187,904,1198,1071,1207
,307,1212,1005,1215,327,1218,980
330 DATA1221,873,1224,997,1227,873,1230,
318,1235,318,1242,318,1265,911
340 DATA1281,904,1297,1623,1317,833,1320
,1620,1323,307,1335,327,1338,980
```

```
350 DATA1365,1386,1368,1482,1384,327,138
9,450,1392,1620,1395,833,1398,1620
360 DATA1405,1497,1409,1567,1413,1589,14
34,1474,1444,2090,1447,1701
370 DATA1450,2096,1455,1701,1462,1485,14
71,1474,1477,840,1515,2009,1533,2077
380 DATA1570,1474,1575,1022,1591,2103,15
96,2167,1629,980,1632,873,1635,997
390 DATA1638,873,1650,1623,1653,1394,165
6,1482,1668,1071,1673,327,1676,980
400 DATA1683,318,1686,417,1699,1350,1710
,980,1713,873,1721,972,1762,2000
410 DATA1767,924,1866,985,1814,2103,1817
,2167,1820,1977,1843,1983,1848,1985
420 DATA1860,2096,1863,2090,1866,1977,18
76,1976,1881,1976,1892,873,1905,1052
430 DATA1917,1005,1943,1482,1957,450,196
0,1620,1963,833,1966,1620
440 DATA1974,1712,1978,1983,1330,1071
450 DATA362,2231,916,318,1806,1497
460 DATA2246,578,2248,565,2250,460
470 DATA2252,503,2254,598,2256,649
480 DATA2258,756,2260,780,2262,1086
490 DATA2264,1170,2266,1216,2268,1336
500 DATA2270,1627,2272,1674,2274,1708
510 DATA2276,326,2278,255,2280,237,-1
```

APPENDIX E: Instructions for the use of Supermon

TO USE SUPERMON (relocated version)

```
LOAD "SUPERMON",DEVICE,1
NEW
SYS49152
```

GENERAL NOTE

On entering Supermon it will save the stack which is restored on exit. It further changes the BREAK vector so when a BRK is met in a program Supermon is called.

All values are entered in hex. Only in ASSEMBLER mode do they have to be prefixed with '$' and then only for the operand.

Once Supermon has been loaded, it is resident until the 64 is either turned off or a program loaded which uses memory from $C000.

INSTRUCTIONS

A – ASSEMBLER – Allows simple assembly of machine code.

A $START OPCODE OPERAND.

For example, A 8000 LDA #$0A

Supermon will prompt with the next address.
Entering <RETURN> after address will exit assembler mode.
Branches are written with the destination address and not its displacement, that is, BEQ $C456.

D – DISASSEMBLER – Disassembles 22 instructions from the address specified.

D $START

for example, D 8000

Hex values may be changed by overtyping and on <RETURN> the same locations will again be disassembled.
Typing D on the bottom line will disassemble the next 22 instructions.
Typing <SPACE> <RETURN> will exit disassembler mode.

F – FILL MEMORY – Fill an area of memory with a specified value.

F $FROM $TO BYTE

for example, F 5000 6000 FF

Useful to set up defaults prior to assembly, in particular to fill with NOPs ($EA).

G – GO RUN – execute machine code.

G – Starts execution at address currently in the Program Counter Register (PC).

G $START – Starts execution at specified address.

H – HUNT – search memory for specified bytes.

H $START $TO DATA

for example, H 5000 6000 'READ – Hunts for ASCII string "READ"
H 5000 6000 A9 0A – Hunts for LDA #$0A

A maximum of 32 bytes may be set.

L – LOAD – Loads a program at its secondary address, leaving BASIC pointers unchanged.

L "filename" ,DEVICE – Device in hex.

08 disk 01 tape

M – MEMORY DISPLAY – Displays hex values.

M $FROM $TO

for example, M 0801, 0891

11 Bytes may be overtyped to change.

P – PRINT DISASSEMBLY – Output hard copy of disassembled listing.
If in Supermon then exit (see below) and set up printer as for normal listing. Re-enter Supermon with subsequent output being directed to the specified device.

For example OPEN4,4:CMD4:SYS49152
P $FROM $TO
When complete, exit Supermon and close printer channel.

R – REGISTER DISPLAY – Displays current register values. This displays the PC, IRQ, Status Register (SR), A, X, Y and Stack Pointer (SP).

R

Values can be overtyped to change. This is of particular use in debugging operations where any of the registers may be altered and program execution continued with a GO command.

S – SAVE – Saves an area of memory to tape or disk.

S "filename",DEVICE,$START,$END

Saves from the start up to, but not including the end address.
For example, S "NAME",08,5000,6001 – Saves from $5000 to $6000, but not the byte at $6001.

T – TRANSFER – Transfers an area of memory to another leaving the original intact.

T $FROM $TO $START

for example, T 5000 6000 1000

You can also use MEMORY DISPLAY this way.
This option may be used in conjunction with the relocator to generate versions of Supermon for use at other locations.

X – EXIT SUPERMON

X

The stack saved when Supermon was entered will be restored. A CLR from BASIC should tidy up any stack problems.

COPYING SUPERMON

Use the save command as normal with the following addresses:

SUPERMON $C000 $C900

APPENDIX F: Extended BASIC memory map

The following gives the main entry points for the UTILITY:

Address	Description
$8000	Initialize Extended BASIC
$800F	Set up Keyword Vectors
$8034	Set Top of Memory
$8041	Set NMI and BRK Vectors
$8054	Set Keyboard Table Set-up Vector
$8061	NMI Routine
$807E	BRK Routine

Address	Description
$8090	Keyword Vector Table
$80F6	Keyword Table – Command Keywords
$81C7	Keyword Table – Function Keywords
$81F5	Routine – GET PARAMETER
$81FB	Switch off BASIC
$8202	Switch on BASIC
$8209	CRUNCH Tokens
$82BC	PRINT Tokens
$8302	Token DISPATCH – Command Keywords
$8329	Token DISPATCH – Function Keywords
$8352	Perform COLOUR
$8381	Perform PLOT
$83A7	Perform WRITE
$83AD	Perform ENTER
$83B3	Perform DOKE
$83D7	Perform DEEK
$8401	Routine – Convert to Positive
$8415	Perform OLD
$842E	Perform AUTO
$847F	Routine – Convert to ASCII
$84A0	Perform TEN
$84EC	Perform TWO
$8537	Perform HEX
$85BF	Perform BIN
$85FC	Perform MEM
$8611	Perform RESET
$8631	Perform POP
$864D	Perform KEY
$8722	KEY Interrupt Routine
$8799	Perform OFF
$87A7	Perform MERGE
$886F	Perform APPEND
$888B	Routine to close up memory and rechain
$8933	Routine to open up memory and rechain
$89C5	Perform RENUM
$8B93	Perform CODER
$8D3A	Perform TRACE
$8D61	Perform TROFF
$8E52	Perform DUMP
$8F44	Perform DELETE
$8FAF	Perform CGOTO
$8FB5	Perform CGOSUB
$8FD2	Perform PROC
$904E	Perform INKEY$

Address	Description
$9080	CHAIN routine
$9169	HIMEM/LOMEM routine
$9181	Perform HIMEM
$918D	Perform LOMEM
$91B7	Perform QUIT
$9200	Start up message
$927D	Completion of DELETE
$92B6	Perform CHAIN
$92DA– $9FFF	unused (expansion for sound/graphic/disk)

Appendix G: Reading an assembler listing

The machine code routines in this book have been presented in two formats. The first was generated using Supermon which is given in Appendix D and instructions for its use in Appendix E. The second was produced using Supersoft's MIKRO assembler cartridge. This appendix deals with listings generated using MIKRO as we feel they require some explanation.

PSEUDO-OPS

These are instructions to the assembler and are not directly executable op-codes.

*=$C000

This tells the assembler to start its assembly at address $C000.

WOR, BYT, and TXT

These instructions reserve bytes in memory. Both WOR and BYT may be followed by any number of arguments separated by commas up to the limit of two screen lines. WOR reserves two bytes and is used to store absolute addresses in low/high format.

WOR $C000,$0100

Puts the four bytes $00,$C0,$00,$01 in four consecutive addresses. BYT reserves single bytes.

BYT $A9,$FF

TXT is followed by a quoted string and places the hex values of the ASCII codes sequentially in memory.

TXT "ABCD"

Puts $41,$42,$43 and $44 into memory.

Each of these directives allocates bytes from the address at which it appears.

LABELS

Labels are used to identify an absolute address in memory. They are normally used as the destination for branches and jumps. They may also be used as operands.

```
LDA #$00     BYTE=#$FF
BEQ ZERO or LDX BYTE
......

ZERO RTS
```

The absolute value of an address may be divided into low/high format by the use of '#<' and '#>' operators.

```
      *=$C000
START LDA #<START
      LDX #>START
```

This loads A with $00 and X with $C0.
Simple numerical calculations may be performed.

```
STORE BYT $00,$FF
      LDA STORE
      LDX STORE+1
```

This loads A with the value held in STORE which has been set to $00 and X with the byte from the next location, that is, $FF.

ADDRESS TABLES

Where labels have been used their values, starting at an arbitrary address, have been given. This is useful to determine the hex values for all branches and jumps.

LINE NUMBERS

The assembler code is entered exactly as one would type in a BASIC program. The same editing rules apply to a MIKRO program as to a BASIC program. Generally, we have retained these line numbers in the listings given for clarity and to aid description.

COMMENTS

An exclamation mark is used in the same way as a REM from BASIC and prefixs comment statements. It tells MIKRO to ignore anything which follows it.

This is by no means the definitive MIKRO manual. We have limited ourselves to using only a few of the options available to allow easier conversion to other assemblers.

Appendix H: Mnemonics generated by CODER

The codes generated are:

[BLK]	– BLACK	[GR1]	– GRAY1	[DEL]	– DELETE
[WHT]	– WHITE	[GR2]	– GRAYS2	[INS]	– INSERT
[RED]	– RED	[LT GRN]	– LIGHT GREEN	[REV]	– REVERSE ON
[CYN]	– CYAN	[LT BLU]	– LIGHT BLUE	[OFF]	– REVERSE OFF
[PUR]	– PURPLE	[GR3]	– GRAY3	[SPC]	– SPACE
[GRN]	– GREEN	[CLS]	– CLEAR SCREEN	[G>SPC]	– SHIFTED SPACE
[BLU]	– BLUE	[HOM]	– HOME CURSOR	[G>?]	– GRAPHIC WITH SHIFT
[YEL]	– YELLOW	[CU]	– CURSOR UP	[G<?]	– GRAPHIC WITH LOGO
[ORG]	– ORANGE	[CD]	– CURSOR DOWN	[CTRL?]	– CONTROL WITH LETTER
[BRN]	– BROWN	[CR]	– CURSOR RIGHT	[F?]	– FUNCTION KEYS
[LT RED]	– LIGHT RED	[CL]	– CURSOR LEFT	[PI]	– PI 3.1416

MULTIPLE CHARACTERS are coded as [10CD]

SPACES
Single, unshifted spaces are not coded. We thought it unnecessary as it detracted from the legibility of the listing.

SPECIAL CODES
The following is an extract from the *Programmer's Reference Guide*, page 74:

> There are some other characters that can be PRINTed for special functions, although they are not easily available from the keyboard. In order to get these into quotes, you must leave empty spaces for them in the line, hit <RETURN> or <SHIFT><RETURN>, and go back to the spaces with the cursor controls. Now you must hit <CTRL> <RVS/ON>, to start typing reversed characters, and type the keys shown below:

Function	Type	Appears As
SHIFT RETURN	SHIFT M	\
switch to lower case	N	N
switch to upper case	SHIFT N	/
disable case switching keys	H	H
enable case switching keys	I	I

Functions 1 and 3 of the above are achieved as stated. CODER replaces them with:

[CRG>M] – SHIFT RETURN
[CRG>N] – SWITCH TO UPPER CASE

The other three can be achieved far more easily. Whilst PRINTing in quotes mode, press <CTRL> and the appropriate letter.

Appendix I: Key codes

The following are the values assigned to keys in locations SFDX and LSTX ($CB/203 & $C5/197):

dec	key	dec	key	dec	key
0	INST/DEL	22	T	44	.
1	RETURN	23	X	45	:
2	CRSR R/L	24	7	46	@
3	F7	25	Y	47	,
4	F1	26	G	48	£
5	F3	27	8	49	*
6	F5	28	B	50	;
7	CRSR U/D	29	H	51	CLR/HOME
8	3	30	U	52	None
9	W	31	V	53	=
10	A	32	9	54	^
11	4	33	I	55	/
12	Z	34	J	56	1
13	S	35	0	57	←
14	E	36	M	58	None
15	None	37	K	59	2
16	5	38	O	60	SPACE
17	R	39	N	61	None
18	D	40	+	62	Q
19	6	41	P	63	RUN/STOP
20	C	42	L	64	No key press
21	F	43	–		

The following are the values of the shift registers SHFLAG and LSTSHF ($028D/653 and $028E/654):

dec	key pattern
0	NO SHIFTS
1	SHIFT
2	LOGO
3	SHIFT AND LOGO
4	CTRL
5	CTRL AND SHIFT
6	CTRL AND LOGO
7	CTRL, SHIFT AND LOGO

Appendix J: Summary of UTILITY commands

APPEND "program name", device
As for merge except that the appended program is tagged on the end of the memory program. Line numbers are not altered. Peculiar listings can be the result. Use RENUM after an append.

AUTO first line number,increment
Automatic line numbering when entering code.

BIN 8 bit binary number[,....
Prints out decimal conversion of binary number in two forms. The first as a low byte conversion and then, separated by an oblique, the high byte conversion (low * 256). The binary number must be of eight bits.

CGOTO variable, calculation or line number
Line numbers can be mathematical equations.

CGOSUB variable, calculation or line number
Line numbers can be mathematical equations.

CHAIN ["filename"][,device]
Will load and run a BASIC program. It also transfers most variables from one program to another.

CODER
Will replace non-standard ASCII and graphic codes with mnemonics. See Appendix H for full list.

COLOUR screen[,border][,text]
Values over 15 can be input, but only the lower four bits will be considered. Border and text parameters are optional.

DEEK(address)
Two byte PEEK. Returns memory location held in address and address+1.

DELETE first line to be deleted, [last line to be deleted]
Deletes lines in the range specified. No last line parameter, it will delete to the end of program.

DPROC name
Start of procedure called 'name'.

DOKE address, value
Two byte POKE. Stores value (0–65535) in address and address+1.

DUMP
Displays the values of all simple variables currently in use.

ENTER (x,y)
Same as INPUT, but first sets cursor position as in PLOT.

EPROC
End of a procedure.

HEX hexadecimal number[,hex number][,...
Prints decimal conversion of hex input. The hex input can be of either two or four characters, but does not require a prefix of '$'.

HIMEM address
Will set the top of memory to the given address, within the range of 1024 to 32767.

INKEY$ [string or string variable]
Will wait for a key press. With no parameter, it will wait for any key. With a parameter, it will wait for a key to correspond to any character in the string. The ASCII value of the key press is placed in reserved variable 'ST'.

KEY 1 to 16,"data"
Loads function keys with data. Maximum of ten characters per key is permissible. Inputs over ten characters will generate a SYNTAX ERROR, but the first ten characters of data will be assigned to the particular key. To generate a return in the data, use "←".
for example, KEY 7,"LIST←"

KEY
Will display the data assigned to all 16 keys in the format they were first entered. This will allow you to overtype the displayed data to amend the key assignations.

To obtain keys: KEY 1–8 as marked on keys
KEYS 9,11,13,15 key with logo
KEYS 10,12,14,16, key with shift and logo

Note: any KEY command will enable the keys if they have previously been disabled.

LOMEM address
Will set the start of BASIC to the given address, within the range of 1024 to 32767.

MEM
Display amount of memory free as an unsigned number.

MERGE ["program name"],[device]
Merges a stored program with that currently in memory according to their line numbers. Lines numbers of the merging program take precedence. If no program name and/or device then the command will default to tape. With no name then first program on tape will be merged.

OFF
Disable the function keys.

OLD
Restores a BASIC program after a NEW or system reset have been actioned. This will not work if an edit has been carried out before OLD is actioned.

PLOT (x,y)
Sets the cursor column and row position. x–0 to 39 and y–0 to 24. 0,0 is the top left hand corner of the screen (cursor home).

POP
Rectifies stack on leaving a subroutine before a RETURN has been called.

PROC name
Calls a procedure called 'name'.

QUIT
Disables the utility and its commands, but protects the area that it uses. The UTILITY can be initialized again by SYS 32768.

RENUM first line number to be changed or 0, increment, new start line number
If first parameter is 0, the whole program will be renumbered, otherwise, from designated line to the end of program. Renumbers the following tokens:GOTO,GO TO, GOSUB, IF THEN, RUN, ON GOTO, ON GOSUB and RESET. It will not renumber CGOTO or CGOSUB.

RESET [line number]
Restore DATA pointer to specific line or start of program.

TEN decimal number[,......
Prints hex conversion of a decimal number.

TRACE
A diagnostic to follow the execution of a BASIC program as it runs.

TROFF
Disables TRACE function.

TWO decimal number[,....
Prints binary conversion of decimal number.

WRITE (x,y)
Same as PRINT, but sets cursor position first as in PLOT.

Note: All commands performing number conversions will do more than one conversion if the values are separated by commas.

ERRORS

These are particular to the UTILITY.

CODER
STRING TOO LONG – more than 254 bytes have been generated by the mnemonics for one program line.
OUT OF DATA – found a character not handled by CORDER.

RENUM
ILLEGAL DIRECT – line at which to start renumbering does not exist.
UNDEF'D STATEMENT – no destination found for a GOTO or GOSUB directive.

APPENDIX K: 64 low memory map

The following is the first few pages of the memory map in the *Programmer's Reference Guide* (PRG), Chapter 5. It is included to avoid continual reference to the PRG to look up label addresses. Some of the descriptions have been changed through personal experience or preference to those in J. Butterfield's map.

LABEL	hex	decimal	Description
D6510	0000	0	6510 direction register
R6510	0001	1	6510 I/O, memory and tape
ADRAY1	0003–0004	3–4	Float to fixed vector
ADRAY2	0005–0006	5–6	Fixed to float vector
CHARAC	0007	7	Search character
ENDCHR	0008	8	End of quote flag
TRMPOS	0009	9	Save screen last TAB
VERCK	000A	10	Flag: LOAD=0 VERIFY=1
COUNT	000B	11	Ptr input buffer/#subscripts
DIMFLG	000C	12	Default DIM to 10 flag
VALTYP	000D	13	DATA type: string=255 numeric=0
INTFLG	000E	14	:integer=128 float=0

LABEL	hex	decimal	Description
GARBFL	000F	15	DATA scan/LIST quote/garbage collection flag
SUBFLG	0010	16	subscript/user fn call
INPFLG	0011	17	$00=INPUT $40=GET $80=READ
TANFLG	0012	18	TAN sign/comparison
	0013	19	current I/O prompt
LINNUM	0014–0015	20–21	integer value
TEMPPT	0016	22	pointer:temp string stack
LASTPT	0017–0018	23–24	last temp string address
TEMPST	0019–0021	25–33	stack for temp strings
INDEX	0022–0025	34–37	utility pointer area
RESHO	0026–002A	38–42	product area for multiply
TXTTAB	002B–002C	43–44	pointer start of BASIC
VARTAB	002D–002E	45–46	pointer start of variables
ARYTAB	002F–0030	47–48	pointer start of arrays
STREND	0031–0032	49–50	pointer end of arrays
FRETOP	0033–0034	51–52	pointer bottom of strings
FRESPC	0035–0036	53–54	utility string pointer
MEMSIZ	0037–0038	55–56	pointer highest address used by BASIC
CURLIN	0039–003A	57–58	current BASIC line number
OLDLIN	003B–003C	59–60	previous BASIC line number
OLDTXT	003D–003E	61–62	BASIC statement for CONT
DATLIN	003F–0040	63–64	current DATA line
DATPTR	0041–0042	65–66	current DATA address
INPPTR	0043–0044	67–68	INPUT vector
VARNAM	0045–0046	69–70	pointer current variable name
VARPNT	0047–0048	71–72	pointer current variable data
FORPNT	0049–004A	73–74	pointer variable for FOR/NEXT
	004B–004C	75–76	Y-save/op-save/BASIC pointer save
	004D	77	comparison symbol accumulator
	004E–0050	78–83	misc work area
	0054–0056	84–86	jump vectors for functions
	0057–0060	87–96	misc numeric work area
FACEXP	0061	97	FPACC#1:exponent
FACHO	0062–0065	98–101	FPACC#1:mantissa
FACSGN	0066	102	FPACC#1:sign
SGNFLG	0067	103	pointer series evaluation constant
BITS	0068	104	FPACC#1:overflow digit
ARGEXP	0069	105	FPACC#2:exponent
ARGHO	006A–006D	106–109	FPACC#2:mantissa
ARGSGN	006E	110	FPACC#2:sign
ARISGN	006F	111	sign comparison result
FACOV	0070	112	FPACC#1:low order rounding
FBUFPT	0072–0072	113–114	pointer cassette buffer

LABEL	hex	decimal	Description
CHRGET	0073–008A	115–138	subroutine: get next byte of BASIC
CHRGOT	0079	121	entry point to get same byte
TXTPTR	007A–007B	122–123	pointer current byte of BASIC
RNDX	008B–008F	139–143	RND seed value
STATUS	0090	144	KERNAL I/O status ST
STKEY	0091	145	switch: STOP and RVS keys
SVXT	0092	146	timing constant for tape
VERCK	0093	147	LOAD=0 VERIFY=1
C3PO	0094	148	serial output: deferred char flag
BSOUR	0095	149	serial output deferred char
SYNO	0096	150	tape EOT received
	0097	151	register save
LDTND	0098	152	how many open files#
DFLTN	0099	153	input device (default=0)
DFLTO	009A	154	output device (default=3)
PRTY	009B	155	tape char parity
DPSW	009C	156	tape byte received flag
MSGFLG	009D	157	BASIC mode flag $00=program $80=direct
PTR1	009E	158	tape pass 1 error log
PTR2	009F	159	pass 2 error log
TIME	00A0–00A2	160–162	real-time jiffy clock
	00A3	163	serial bit count/EOI flag
	00A4	164	cycle count
CNTDN	00A5	165	tape sync countdown/bit count
BUFPNT	00A6	166	pointer tape I/O buffer
INBIT	00A7	167	RS232 input bits tape wrt ldr/rd count
BITCI	00A8	168	RS232 input bit count tape wrt new byte/rd error
RINONE	00A9	169	RS232 start bit flag
RIDATA	00AA	170	RS232 input byte buffer tape scan/counter/ldr
RIPRTY	00AB	171	RS232 input parity tape wrt ldr length'rd checksum
SAL	00AC–00AD	172–173	pointer tape buffer/scrn scroll
EAL	00AE–00AF	174–175	tape end address/end program
CMPO	00B0–00B1	176–177	tape timing constants
TAPE1	00B2–00B3	178–179	pointer start of tape buffer
BITTS	00B4	180	RS232 out bit count/tape enabled=1
NXTBIT	00B5	181	RS232 next bit to send/tape EOT
RODATA	00B6	182	RS232 out byte buffer/rd char error
FNLEN	00B7	183	Length current file name
LA	00B8	184	Current logical file number

LABEL	hex	decimal	Description
SA	00B9	185	Current secondary address
FA	00BA	186	Current device number
FNADR	00BB–00BC	186–187	Ptr current file name address
ROPRTY	00BD	189	RS232 out parity/tape rd input char
FSBLK	00BE	190	tape #blocks left to wrt/rd
MYCH	00BF	191	Serial word buffer
CAS1	00C0	192	Tape motor control
STAL	00C1–00C2	193–194	I/O start address
MEMUSS	00C3–00C4	195–196	KERNAL setup ptr/tape temp address
LSTX	00C5	197	Last key pressed
NDX	00C6	198	#characters in k/b queue
RVS	00C7	199	RVS char print flag 1=yes 0=no
INDX	00C8	200	Ptr end of line for INPUT
LXSP	00C9–00CA	201–202	Cursor row,col at start of INPUT
SFDX	00CB	203	Current key pressed 64=no key
BLNSW	00CC	204	0=blink cursor
BLNCT	00CD	205	Cursor countdown timer
GDBLN	00CE	206	Character at cursor pos
BLNON	00CF	207	Cursor blink flag on/off
CRSW	00D0	208	Flag: INPUT from screen or GET from keyboard
PNT	00D1–00D2	209–210	Ptr current start of screen line add
PNTR	00D3	211	Cursor col on above line
QTSW	00D4	212	Flag: 0=cursor in edit mode else in quote mode
LNMX	00D5	213	Physical screen line length
TBLX	00D6	214	Current row where cursor lives
	00D7	215	Last inkey/checksum/buffer temp data
INSRT	00D8	216	#inserts outstanding
LDTB1	00D9–00F2	217–242	Screen line link table
USER	00F3–00F4	243–244	Ptr screen colour
KEYTAB	00F5–00F6	245–246	K/b decode table vector
RIBUF	00F7–00F8	247–248	RS232 input buffer ptr
ROBUF	00F9–00FA	249–250	RS232 output buffer ptr
FREKZP	00FB–00Fe	251–254	Free zero page area
BASZPT	00FF	255	BASIC temp data area
	0100–010A	256–266	Float to ASCII work area
	0100–013E	256–318	Tape error log
	0100–01FF	256–511	Processor stack
BUF	0200–0258	512–600	System input buffer
LAT	0259–0262	601–610	Logical file table
FAT	0263–026C	611–620	Device number table
SAT	026D–0276	621–630	Secondary address table
KEYD	0277–0280	631–640	Keyboard buffer

LABEL	hex	decimal	Description
MEMSTR	0281–0282	641–642	Start of BASIC memory
MEMSIZ	0282–0283	643–644	Top of BASIC memory
TIMOUT	0285	645	Serial bus time out flag
COLOR	0286	646	Current character colour
GDCOL	0287	647	Background colour under cursor
HIBASE	0288	648	Start of screen memory: page number
XMAX	0289	649	Size of k/b buffer
RPTFLG	028A	650	Flag: 128=repeat all keys
KOUNT	028B	651	Repeat speed counter
DELAY	028D	653	Flag: shift/ctrl/logo key
LSTSHF	028E	654	Last shift pattern
KEYLOG	028F–0290	655–656	K/b table setup ptr
MODE	0291	657	Flag: 0=disable shift keys 128=enable
AUTODN	0292	658	0=scroll down enable
M51CTR	0293	659	RS232 control register
M51CDR	0294	660	RS232 command register
M51AJB	0295–0296	661–662	RS232 non-standard baud rate
RSSTAT	0297	663	RS232 status register
BITNUM	0298	664	RS232 bits left to send
BAUDOF	0299–029A	665–666	RS232 Baud rate
RIDBE	029B	667	RS232 index to end of input buffer
RIDBS	029C	668	RS232 page number of start of input buffer
RODBS	029D	669	RS232 page number of start of output buffer
RODBE	029E	670	RS232 index to end of output buffer
IRQTMP	029F–02A0	671–672	IRQ save during tape I/O
ENABL	02A1	673	RS232 enable/CIA 2 (NMI) interrupt control
	02A2	674	CIA 1 timer A control log during tape I/O
	02A3	675	CIA 1 interrupt log tape read
	02A4	676	CIA 1 Timer A enable log tape read
	02A5	677	Screen line marker
	02A6	678	PAL/NTSC flag 0=NTSC 1=PAL
	02A7–02FF	679–767	Unused
	02C0–02FE	704–766	Block 11 for sprites
IERROR	0300–0301	768–769	Vector: BASIC error message ($E3B8)
IMAIN	0302–0303	770–771	Vector: BASIC warm start($A483)
ICRNCH	0304–0305	772–773	Vector: Crunch BASIC tokens($A57C)
IQPLOP	0306–0307	774–775	Vector: Print BASIC tokens($A71A)
IGONE	0308–0309	776–777	Vector: Start new BASIC line($A7E4)
IEVAL	030A–030B	778–779	Vector: BASIC token evaluate($AE86)
SAREG	030C	780	Save A register

LABEL	hex	decimal	Description
SXREG	030D	781	Save X register
SYREG	030E	782	Save Y register
SPREG	030F	783	Save status register
USRPOK	0310	784	USR function jump instrn ($4C)
USRADD	0311–0312	785–786	USR address low/high form($B248)
	0313	787	Unused
CINV	0314–0315	788–789	Vector:Hardware IRQ($EA31)
CBINV	0316–0317	790–791	Vector:BRK interrupt($FE66)
NMINV	0318–0319	792–793	Vector:NMI($FE47)
IOPEN	031A–031B	794–795	Vector:KERNAL OPEN($F34A)
ICLOSE	031C–031D	796–797	Vector:KERNAL CLOSE($F291)
ICHKIN	031E-031F	798–799	Vector:KERNAL CHKIN($F20E)
ICKOUT	0320–0321	800–801	Vector:KERNAL CHKOUT($F250)
ICLRCH	0322–0323	802–803	Vector:KERNAL CLRCHN($F333)
IBASIN	0324–0325	804–805	Vector:KERNAL CHRIN($F157)
IBSOUT	0326–0327	806–807	Vector:KERNAL CHROUT($F1CA)
ISTOP	0328–0329	808–809	Vector:KERNAL STOP($F6ED)
IGETIN	032A–032B	810–811	Vector:KERNAL GETIN$F13E)
ICLALL	032C–032D	812–813	Vector:KERNAL CLALL($F32F)
USRCMD	032E–032F	814–815	Vector:Warm start($FE66)
ILOAD	0330–0331	816–817	Vector:KERNAL LOAD($F4A5)
ISAVE	0332–0333	818–819	Vector:KERNAL SAVE($F5ED)
	0334–033B	820–827	Unused
TBUFFR	033C–03FB	828–1019	Tape I/O buffer
	03FC–03FF	1020–1023	Unused
	0340–037E	832–894	Block 13 sprite data
	0380–03BE	896–958	Block 14 sprite data
	03C0–03FE	960–1022	Block 15 sprite data
VICSGN	0400–07FF	1024–2047	Screen memory
	0400–07E7	1024–2023	Visible memory
	07F8–07FF	2040–2047	Sprite block data pointers 0–7
	0800	2048	Start of BASIC (TXTTAB–1)

THE ULTIMATE PROGRAMMER'S TOOLKIT – INVALUABLE PROGRAMMING AIDS FOR YOUR COMPUTER!

Utilities to take the pain out of programming... Utilities to customise your 64 and explore its hidden potential...

All Commodore 64 programmers will find this software toolkit of programming aids, BASIC enhancements and other utilities truly invaluable.

As well as revealing the inner workings of the 64, BASIC versions of programming utilities such as the auto line number, block delete, renumber and program merge routines are presented and explained, programmable function keys covered, and the 64's peripheral potential investigated.

The BASIC utilities, plus trace, variable dump, procedure, graphics routines and many more are then implemented in machine code. BASIC loaders are provided as well as a complete monitor listing for entering the routines. The separate modules build into a total utility package which overcomes the limitations of the 64's BASIC to give you a powerful programming aid.

Delving deep into the workings of the 64, this book compliments **The Companion to the Commodore 64** to provide the user with the tools and information needed to unleash the full power of this great machine.

ISBN 0-330-28671-4

U.K. £6.95

90000

9 780330 286718