THE BBC MICRO

TOOLBOX

Aids to more efficient programming

BBC SOFT

THE BBC MICRO

TOOLBOX

Aids to more efficient programming

Ian Trackman

BRITISH BROADCASTING CORPORATION

CONTENTS

INTRODUCTION

The BBC Microcomputer Programmer's Toolbox is a collection of 25 utility routines, divided into two main groups.

The first group consists of routines, either written in BASIC or as assembly language listings, which are intended to be incorporated into your own programs in order to make them more efficient or versatile. The routines are:

- Circle draw and fill
- Double-size characters
- Graphics dump
- Shape maker
- Sideways characters
- Sorting routines
- Speech chip number generator.

The second group of utilities are complete programs in their own right. They will help you to write, test and debug your own application programs. Most of them are loaded into the computer independently of any BASIC program which is already installed. The programs are:

- Character generator
- Cross referencer
- Disassembler
- Global replacer
- Packer
- RAM test
- REM stripper
- Resequencer
- Spacer
- Space remover
- Unpacker
- Variable dump

All of the utilities are recorded on the accompanying tape. Machine-code programs are in the form of object code. This manual describes how to load and use all of the routines. In addition, an annotated and commented source listing is printed at the end of each section so that you can relate the operating instructions to the code itself.

This manual is not intended to be a 'How to Write Programs' guide. We assume that you are reasonably competent in BASIC, so that we do not go into explanations of elementary principles, nor do we repeat information which is set out in detail in the User Guide. On the other hand, we do not expect you to be able to program in assembly language. You can use all of the routines in the Toolbox knowing only BASIC.

Nevertheless, we have included the assembly language source listings for a number of reasons. If you can write in machine-code, we hope that you will be interested in studying how we have used the computer's facilities to produce the desired results. The programs also contain some subroutines which may be useful to you in their own right.

Murphy's Third Law of computer programming - 'There's always one more bug' - probably also applies to the Toolbox. Although we've tested the programs extensively, there's always one situation that we may have overlooked and which will cause a program to fail. If this happens to you, please accept our apologies in advance. However, we hope that after you have roundly cursed us, you might care to spend a few moments in trying to identify and cure the bug by examining the listings. If you do, we'd be very grateful to hear from you. Please write to:

The Software Editor
BBC Publications
35 Marylebone High Street
London W1M 4AA

and he will forward your comments to the author.

One other reason for giving you the source listings is that you might want to use more than one of the programming utilities at the same time. For those of you who can already program in assembly language, all we need say is that you should use the disassembler to create a source listing from the object code, re-set the origin address and re-assemble it. For those of you for whom the last sentence might just as well have been written in Mongolian, there is a step-by-step explanation in the 'Using the Programming Utilities' section.

Finally, a few words about copyright. All of the programs in the Toolbox are copyrighted. However, we have no objection to your including the first group of programs (listed above) in your own programs. If your programs are distributed commercially, all that we require is for you to credit the source of the routines. The programs in the second group are another matter. Since they are not intended to be incorporated into larger programs, if you copy and re-distribute any of them, you are in breach of copyright. Just as 'shop-lifting' is sometimes used as a euphemism for stealing, 'software piracy' perhaps suggests that there is something daring and swashbuckling about it. There isn't - it is just plain theft and we will have no hesitation in bringing legal proceedings against anyone discovered committing the crime.

And after the heavyweight warning - a request. A second Toolbox is already in the planning stage. If you can suggest ways in which the present utilities could be enhanced or some further utilities which you would like us to include or if you have already written a utility that you would like us to consider for publication as part of a future Toolbox, please write to us c/o the Software Editor (address as before).

Some of the programs make calls to the Operating System and make use of facilities available from O.S. 1.0 and above.

USING THE PROGRAMMING UTILITIES

The Programming Utilities are:

	Tape name	*Size (bytes)*
Cross referencer	XREF	&200
Packer	PACKER	&200
REM stripper	REMSTRP	&300
Replacer	REPLACE	&200
Resequencer	RESEQ	&300
Spacer	SPACER	&200
Space remover	CRUNCH	&100
Unpacker	UNPACK	&300
Variable dump	VARDUMP	&100

INSTALLING THE PROGRAMMING UTILITIES

The Programming Utilities are the programs in the Toolbox which, in some way, operate on a BASIC program or its variables but remain independent of it. XREF and VARDUMP simply scan the memory and report on what has been found, whereas the others make actual changes to the BASIC program. All of them are in machine-code.

Since the BASIC program must be in the computer's memory ('co-resident') in order for the utilities to work, they cannot be loaded into the area of RAM normally occupied by the BASIC program. We have to find some other space for them.

Tape-based Systems

In a tape-based system, there are 128 free bytes between locations &D00 and &D7F, which, regrettably, are not enough for our utilities. On a 32K computer, that really only leaves the BASIC workspace between &E00 and the bottom of the display area (HIMEM). You can move HIMEM down and BASIC will then not use the area of memory above it. However, HIMEM is reset whenever the Mode is changed. If you change from a low-resolution Mode to a higher-resolution Mode (e.g. Mode 5 to Mode 1), you will also over-write anything that was stored in the memory which now forms part of the enlarged screen display RAM (e.g. between locations &3000 and &57FF). It is therefore impractical to store a co-resident utility above HIMEM.

The alternative is to store it where the BASIC program would normally reside and force the BASIC program to move elsewhere. **PAGE** is the pseudo-variable which controls where a BASIC program is loaded and is normally automatically set by a tape-based system to location &E00.

The utilities need &100, &200 or &300 bytes of memory – the table above and the utility's instruction notes will tell you its particular memory requirements. If you set **PAGE=&F00**, you can use the &100-byte utilities. **PAGE=&1000** will give you room for both the &100- and &200-byte utilities and **PAGE=&1100** will let you use all of them.

You must set PAGE *before* you load or enter a BASIC program. If you set it after the program is in memory, you'll get a 'Bad program' error message. Once set, PAGE remains fixed until you alter it or until you press the Break key, even if you load, run or save several BASIC programs. Therefore, set PAGE to the required value at the very start of a programming session. Of course, you'll lose &100 to &300 bytes of otherwise free memory, but this should not be too much of a problem unless you are running a very big program with many variables or large arrays in a high-resolution Mode.

Disk-based Systems

We explain below how to transfer the programs from tape to disk. The tape includes versions of all the programming utilities suitable for use with disk-based systems.

The Disk Filing System has already grabbed a large chunk of memory from &E00 to &18FF but we can, with care, borrow some of it back.

There are five file buffers between &1400 and &18FF. Provided that your program doesn't open any files, this area should be a reasonably safe place to install one or more Toolbox utilities. We have assembled the disk versions of the &300-byte programs at &1600, the &200-byte programs at &1700 and the &100 programs at &1800. Disk-users will be only too well aware of the problems of the additional memory constraints imposed by the DFS. You may already have been forced to write or run programs which borrow the disk buffers for variable storage (e.g. **LOMEM = &1400 : HIMEM = &1900**). Running such a program will, of course, over-write any utility stored in the same work-space.

Since we are not using the cassette filing system, we can also take over the two &100-byte buffers that are normally reserved for it. The two pages of memory from &900 to &AFF can be used to store all of the &100- and &200-byte programs. These buffers are also the RS423 input and output buffers and if, for instance, you use a serial printer, you should be aware that you will over-write any utilities stored there. It would be convenient to take over the next &100 bytes for our &300-byte utilities but the next page at &B00 contains the function key definitions and putting bytes in there creates unexpected results.

In fact, when using the utilities ourselves, we keep more than one in memory at the same time by moving one of them into the tape buffers, moving another to &1400 and loading a third utility at &1600. For example, a most useful combination when working on a BASIC program is XREF at &900, REPLACE at &1400 and RESEQ at &1600. We explain below how to move the programs around in the computer's memory.

As you now appreciate that we are taking over Operating System buffers for the utilities, you will see why we cannot unconditionally guarantee that they will work there properly. If you want to be absolutely safe, you must relocate the programs to &1900 (as described below), change PAGE and move your BASIC programs even higher up the memory.

BACK-UP COPIES AND TAPE-TO-DISK TRANSFERS

You should always make back-up copies of your programs.

First, we remind you that most of the programming utilities alter your BASIC programs. *Always* save a copy of your own program *before* you start making changes to it, in case you want to back-track or in the unlikely (we hope!) event of the utility crashing and damaging your program.

Secondly, you should make back-up copies of the Toolbox programs - as we said in the Introduction, not so that you can start up an illegal software factory, but to protect yourself against accidental damage to your tapes.

To copy a machine-code program, you need to know its size. You can find this out by using the *OPT 1,2 command. (See page 398 of the User Guide.) When you load the program (as described in the next section), the computer will display the start and end addresses of the program. You can then re-save it, using these addresses, in accordance with the details on page 392 of the User Guide.

To transfer the programs to disk, change to tape with a *TAPE command, load the programs as described in the next section, change back to disk with *DISK, then save the programs with the *SAVE command as detailed on page 53 of the Disk System User Guide. Do not re-define the re-load or execution addresses. You will also need to use short file names - the tape names will usually do. For example, here are the steps that you would follow to transfer XREFDISK from tape to disk:

***TAPE**
***OPT 1,2**
***LOAD XREFDISK** (the display will inform you that the start address is &1700 and the length is &1E4)
*Press BREAK (to reset to DISK and to clear the Disk Control Block)
***SAVE XREF 1700 18E4**

LOADING THE UTILITIES

With tape-based systems, remember to set PAGE as necessary - see above.

If you are loading a program from tape in order to copy or disassemble it, type
***OPT 1,2**

The command to load a machine-code program (for both tape- and disk-based systems) is
***LOAD** filename
If you want to load-and-run it, type
***RUN** filename
(Don't try CHAIN - it won't work and it will splatter the utility all over your BASIC program.)

With disk-based systems instead of ***RUN** filename, you can type
***filename**
provided that, with two drives, the utility is on the library disk (see page 45 of the Disk System User Guide).

Also with disks, you can make use of the ***EXEC** command. For example, you could create a SQUEEZE file which contains the commands:

***REMSTRP**
Y
***CRUNCH**
***PACKER**
REN.

This would give you an automated link between the three program-squeezing utilities. (Please refer further to page 30 of the DFS Manual.)

RE-LOCATING THE PROGRAMS

You may decide that you would like to have two (or more) of the utilities in memory at the same time, so that you can use them together on a BASIC program, instead of having to use one, then load the second, use it, and so on.

This section assumes that even if you cannot program fluently in assembly language, you have at least read the relevant section of the User Guide (pages 442–9) and understand the principles of the use of the BBC assembler.

As mentioned above and subject to the limitations referred to there, disk users can put the re-located utilities into the areas of memory between &900 and &AFF and between &1400 and &18FF. Tape users will have to raise the position of PAGE to give themselves more memory for the extra utilities, as will disk users who want to install, say, two &300-byte programs.

Unless a machine-code program is written in what is known as 'relocatable' code, it must be loaded into the same addresses in memory for which it was written. This means that if you want to load one of the utilities somewhere other than at its original location, you will need to re-assemble it with a new starting address.

Begin by resetting PAGE if necessary. Otherwise, when you re-assemble the machine-code program, you could find it writing all over your source code!

Secondly, disassemble the object code (the actual program on tape), using the Toolbox Disassembler. You should now have an assembly language listing in the form of a BASIC program. It should resemble the source listing printed in this manual, but without the variable names. Add these with the help of Replacer. You needn't add anything starting with a reverse oblique ('\'), since those are only comment statements.

Next, add the FOR . . . NEXT loop commands and, if applicable, the procedures for adding strings to the end of the programs. You can test the assembly at this stage by running the BASIC program. It should assemble without error messages, simply displaying the start address twice on the screen. Now find the line near the start of the listing:

org=& . . . .

This is where to set the new starting address of your program. If you re-run the program with a new address assigned to the variable **org**, the machine-code will be assembled there. For example, if you want to locate the program at &1100, you would change the line to:

org=&1100

All that remains is to save the new object code with the *SAVE command. The starting address is, of course, set by **org** and the end address is the value of the variable **P%** after assembly. (Remember to use hexadecimal numbers - **PRINT ~P%**.)

Here are the steps that you would go through in order to re-locate the tape version of VARDUMP to a new location at &1000:

PAGE=&1100 (&1000+&100 bytes for the machine-code program)
***OPT 1,2** (to display the start address and length of the machine-code)
RUN "DISASS" (with the 'SPOOL' option on - see DISASS instructions) and disassemble the machine-code
NEW
***EXEC DUMP**
Add assembly instructions and variable names (copy them from VARDUMP's source listing)
Edit '**org=&1000**'
RUN (to assemble the code)
PRINT ~P%
***SAVE VARDUMP 1000 aaaa** (where aaaa is the address obtained from the last step)

'TACKED-ON' BYTES

All of the utilities which scan BASIC programs check whether they have reached the end by looking for the 'end-of-program' byte &FF. There are techniques for adding extra bytes to the end of a BASIC program to hold, perhaps, a machine-code subroutine or some data. These involve moving the &FF byte to beyond the extra items to be added to the BASIC program. When the Toolbox utilities reach these extra bytes, they are treated just as if they are lines of a BASIC program. Since they aren't, the utilities will be unable to interpret them properly and will probably crash.

Unless you understand how to decode the raw bytes of a BASIC program, we are sorry that we cannot offer you much help if you are trying to cope with such a BASIC program.

HIDDEN CONTROL CHARACTERS

A BASIC program, in the form that is stored in the computer's memory, contains single 'token' bytes which correspond to the BASIC keywords. There is a table of the keywords and tokens in the SPACER utility instructions.

Normally, these tokens are unique and do not appear elsewhere in a BASIC program. For example, line-number references are encoded into three bytes, preceded by the token &8D (141 in decimal). In order to check line-references, some of the utilities scan the BASIC program, looking for &8D tokens. You may recognise CHR$141 as the Mode 7 control code for double-height characters. There will not be a problem with a line containing a command such as:
PRINT CHR$141 "HELLO"
However, if you include the actual code *within* a string, as in :
PRINT "cHELLO"
where **c** represents the (invisible) ASCII character 141, the utilities will be unable to decode the following three bytes and might crash.

ESCAPE AND BREAK

You cannot interrupt the programming utilities with the Escape key, only the Break key. If you do stop, say, PACKER in the middle of its run, you will probably corrupt the BASIC program on which it is working. That is another reason why we recommend that you should always make back-up copies of your BASIC programs before using the programming utilities on them.

THE INSTRUCTIONS

When you read through the instructions, you'll notice that we keep repeating sections of the notes. We make no excuse for failing to be creatively original! Our aim was to try to make the method of using the Toolbox utilities as similar as possible, so you can move easily from one to the other.

USER-DEFINABLE CHARACTER GENERATOR

The purpose of this program is to provide you with a simple way of creating your own character set and to edit individual character shapes without the need to carry out any of the calculations described on pages 170–1 of the User Guide.

The program is called CHARGEN. It is written in BASIC, so CHAIN or LOAD-and-RUN it.

When the program starts, you'll see a grid on the left of the screen. This is used to display an enlarged 8 × 8 pattern of your selected character.
Below it is the prompt **CHR$?**.
Type in the ASCII number (in decimal) of the character that you want to create or edit. The program will only accept numbers in the range 32 to 255.

When you enter a number, the program will display a 'magnified' view of the corresponding character in the large grid. It will also display the character, at normal size, in the small red box below the prompt line. The normal-sized character is re-displayed whenever you make an editing change in the large grid so that you can constantly review the effect of the alterations that you are making.

In the bottom left-hand corner of the screen are the eight hexadecimal double-byte numbers which represent the bit pattern of the current shape, preceded by '**VDU 23**' and the character's ASCII number. This command line is what you will need to insert into your own program in order to re-create the character.

On the right-hand half of the screen is a display of 48 characters as currently defined together with their ASCII numbers. When the program starts, ASCII characters 208 to 255 will be shown.

Characters 224 to 255 will probably be the characters that you will use the most, since they are not pre-defined within the Operating System. Furthermore, you do not need any extra memory to use them.

With Operating System 1.0 and above, if you re-define characters 32 to 127, you will change the characters which you can type on the keyboard. You can also re-define characters 128 to 223. If you want to do so, you'll first need to reset PAGE in order to make room for the new character definitions. You should refer to pages 427 and 428 of the User Guide for full details. Note, however, that there is a printing error at the beginning of the seventh paragraph on page 427. It should read 'After a *FX 20,6 command' instead of 'After a *FX 20,1 command'. The second parameter of the *FX 20 command depends on how many extra character blocks you want to use. The table, which you could incorporate into the table at the bottom of page 427, is as follows:

ASCII code	
&80 to &9F	*FX 20,0
&A0 to &BF	*FX 20,1
&C0 to &DF	*FX 20,2
&E0 to &FF	*FX 20,3
&20 to &3F	*FX 20,4
&40 to &5F	*FX 20,5
&60 to &7F	*FX 20,6

Remember that if you leave the character set in its normal, 'imploded' form, any new characters created by you will be automatically mapped on to four other characters at 32-byte intervals below it (please see the preceding paragraph on page 427 of the User Guide).

Here are the editing and other commands that the program uses:

The **cursor move keys** (up, down, left and right) will move the cursor around the grid. The cursor wraps around at the end of each line and at the top and bottom of the grid.

The **space bar** fills one small square in the grid at the current position of the cursor.

X clears the bit at the cursor's position.

N signifies acceptance of the current shape and the start of a new character definition.

C cancels any editing done on the current character and restores the bit pattern as it was when the character was first loaded into the grid, even if another character has been subsequently 'copied' into the grid (as explained below).

I inverts the bit pattern of the character. It effectively transforms the character from white-on-black to black-on-white.

R rotates the character shape 90 degrees in a clockwise direction.

M creates a mirror image of the character.

F causes the ASCII character display on the right of the screen to scroll forward. **B** causes it to scroll back. These two commands will not work until you have specified an ASCII number in response to the command line prompt. You cannot scroll below 32 or above 255.

The **COPY** key will copy the next character typed into the editing grid. For example, if you wanted to make up, say, a lower-case 'a' with a German umlaut (two dots) over it as ASCII character 224, you would first enter 224 in response to the **CHR$?** prompt. Then press the COPY key followed by a lower-case letter 'a'. You can now add the dots. The original lower-case 'a' will not be affected.

You can also use the COPY key facility to create a whole series of similar characters. On Operating System 1.0, the command *FX 225,224 in the program causes the red function keys, when pressed at the same time as the CTRL key, to produce ASCII characters 224 to 233. So, you can create character 224, type **N** for a new character, specify 225 as the next character to be defined, then press **COPY** followed by **CNTRL** and red key f0 (pressed together). In that way, you can bring copies of characters 224 to 233 back into the editing grid without re-defining the original shape.

P causes the **VDU 23 . . .** command displayed in the bottom left-hand corner of the screen to be sent to a printer (as a single line). The program assumes that a parallel printer is connected. You will need to insert appropriate commands into the program for a serial printer (***FX 5** and ***FX 8**) or to enable line-feeds (***FX 6**). There is a bug in some early versions of the Operating System. **VDU 21**, which causes output to be sent to the printer without being echoed on the screen, inhibits line-feeds on the bugged versions. If you have this problem, you'll need to patch **PROCvdu23** and use the **VDU 1** command inside of the printing loop to send output, one character at a time, to the printer.

Exit the program by pressing Escape. The effect of your editing will remain in memory. However, you should now type into your program the VDU 23 commands to re-create the new character shapes, in case they get over-written by another program. You can try the shapes out in your program and then return to CHARGEN to edit them further.

There are several techniques that you can use to join characters together in order to make up larger shapes for animation. Some of these ideas are contained in programs which form part of the 'Making the Most of the Micro' package.

```
 10 REM USER-DEFINABLE CHARACTER
    GENERATOR
 20 :
 30 REM (c) Ian Trackman 1982
 40 :
 50 ON ERROR GOTO 940
 60 :
 70 MODE 1
 80 :
 90 REM Shift-function keys for
    CHR$224/233 (OS 1+)
100 *FX 225,224
110 :
120 REM Cursor-move keys off
130 *FX 4,1
140 :
150 @% = &304 : REM Print field width
160 :
170  DIM
     BIT%(7,7),HOLD%(7,7),TEMP%(7,7),
     BYTE%(7)
180 :
190 REM Set up A,X,Y registers for
    osword call
200 A% = 10
210 DIM X%8
220 Y% = X% DIV &100
230 :
240 TBL = 224 : REM Table pointer
250 :
260 REPEAT
270   PROCframe
280   PROCtable (TBL-16)
290   VDU 23;&670A;0;0;0; : REM Normal
      cursor
300   VDU 23,1,1;0;0;0;
310   *FX 15,1
320   :
```

```
330   REPEAT
340     PRINT TAB(11,21) SPC3 : REM
        Wipe previous number
350     PRINT TAB(4,21) "CHR$ ? ";
360     N = FNinput
370     IF N > 31 AND N < 256 THEN OK =
        TRUE ELSE OK = FALSE : VDU 7
380     UNTIL OK
390   :
400   PRINT TAB(9,21) ": ";N
410   VDU 23,1,0;0;0;0;
420   :
430   PROCchar (N)
440   PROCvdu23 (TRUE)
450   :
460   REM Save starting pattern
470   FOR I% = 0 TO 7
480     FOR J% = 0 TO 7
490       HOLD%(I%,J%) = BIT%(I%,J%)
500       NEXT
510     NEXT
520   :
530   X = 0
540   Y = 0
550   VDU 23,1,1;0;0;0;
560   :
570   REPEAT
580     PRINT TAB(X*2 + 1,Y*2 + 4);
590     VDU 23;&650A;0;0;0; : REM Fat
        cursor
600     *FX 15,1
610     K = GET
620     VDU 23,1,0;0;0;0;
630     :
640     IF K = &87 THEN PROCchar (GET)
        : PROCcreate : REM Copy
650     IF K = &88 THEN X = X-1 : REM
        Left
660     IF K = &89 THEN X = X+1 : REM
        Right
```

```
670       IF K = &8A THEN Y = Y+1 : REM
          Down
680       IF K = &8B THEN Y = Y-1 : REM
          Up
690       :
700       IF K = 32 THEN PROCbit_on : REM
          Space
710       K$ = CHR$(K AND &DF) : REM Mask
          input to upper-case
720       IF K$ = "X" THEN PROCbit_off
730       IF K$ = "C" THEN PROCcancel
740       IF K$ = "I" THEN PROCinvert
750       IF K$ = "M" THEN PROCmirror
760       IF K$ = "R" THEN PROCrotate
770       IF K$ = "F" AND TBL < 224 THEN
          PROCtable (TBL)
780       IF K$ = "B" AND TBL > 63 THEN
          PROCtable (TBL - 32)
790       IF K$ = "P" THEN PROCvdu23
          (FALSE)
800       PROCvdu23 (TRUE)
810       :
820       IF X > 7 THEN X = 0 : Y = Y +
          1
830       IF X < 0 THEN X = 7 : Y = Y -
          1
840       IF Y > 7 THEN Y = 0
850       IF Y < 0 THEN Y = 7
860       UNTIL K$ = "N"
870     :
880     UNTIL FALSE
890 :
900 END
910 :
920 :
930 REM Error trap
940 MODE 6
950 @% = &90A
960 *FX 4
970 *FX 225,1
```

```
 980 IF ERR <> 17 THEN REPORT : PRINT "
     at line "; ERL
 990 END
1000 :
1010 :
1020 DEF PROCalter
1030 VDU 23,N
1040 :
1050 FOR I% = 0 TO 7
1060   VDU BYTE%(I%)
1070   NEXT
1080 :
1090 ENDPROC
1100 :
1110 :
1120 DEF PROCbit_off
1130 BIT%(X,Y) = FALSE
1140 PROCblock (X,Y)
1150 BYTE%(Y) = BYTE%(Y) AND (&FF -
     2^(7-X)) : REM Mask bit off
1160 PROCalter
1170 X = X + 1
1180 ENDPROC
1190 :
1200 :
1210 DEF PROCbit_on
1220 BIT%(X,Y) = TRUE
1230 PROCblock (X,Y)
1240 BYTE%(Y) = BYTE%(Y) OR 2^(7-X) :
     REM Mask bit on
1250 PROCalter
1260 X = X + 1
1270 ENDPROC
1280 :
1290 :
1300 DEF PROCblock (X,Y)
1310 IF BIT%(X,Y) THEN GCOL 0,131 ELSE
     GCOL 0,128
1320 VDU 24,X*&40;-Y*&40;X*&40 + &38;&38
     - Y*&40;
```

```
1330 CLG
1340 COLOUR 128
1350 ENDPROC
1360 :
1370 :
1380 DEF PROCcancel
1390 :
1400 FOR I% = 0 TO 7
1410   FOR J% = 0 TO 7
1420     BIT%(I%,J%) = HOLD%(I%,J%)
1430     NEXT
1440   NEXT
1450 :
1460 PROCcreate
1470 PROCfill
1480 PROCvdu23 (TRUE)
1490 ENDPROC
1500 :
1510 :
1520 REM Explode the character's bits
     into an array
1530 DEF PROCchar (N)
1540 ?X% = N
1550 CALL &FFF1
1560 :
1570 FOR I% = 0 TO 7
1580   BYTE%(I%) = X%?(I%+1)
1590   M% = &80
1600   FOR J% = 0 TO 7
1610     IF (X%?(I%+1) AND M%) THEN
         BIT%(J%,I%) = TRUE ELSE
         BIT%(J%,I%) = FALSE
1620     M% = M% DIV 2
1630     NEXT
1640   NEXT
1650 :
1660 PROCfill
1670 ENDPROC
1680 :
1690 :
```

```
1700 REM Create VDU 23 line from array
1710 DEF PROCcreate
1720 VDU 23,N
1730 :
1740 FOR I% = 0 TO 7
1750   B% = 0
1760   M% = &80
1770   FOR J% = 0 TO 7
1780     IF BIT%(J%,I%) THEN B% = B% +
         M%
1790     M% = M% DIV 2
1800     NEXT
1810   VDU B%
1820   BYTE%(I%) = B%
1830   NEXT
1840 :
1850 ENDPROC
1860 :
1870 :
1880 DEF PROCfill
1890 :
1900 FOR I% = 0 TO 7
1910   FOR J% = 0 TO 7
1920     PROCblock (I%,J%)
1930     NEXT
1940   NEXT
1950 :
1960 ENDPROC
1970 :
1980 :
1990 DEF PROCframe
2000 VDU 23,1,0;0;0;0;
2010 CLS
2020 :
2030 REM Big box
2040 COLOUR 129
2050 VDU 28,0,2,17,2
2060 CLS
2070 VDU 28,17,19,17,2
2080 CLS
```

```
2090 VDU 28,0,19,17,19
2100 CLS
2110 VDU 28,0,19,0,2
2120 CLS
2130 VDU 26
2140 :
2150 COLOUR 128
2160 GCOL 0,2
2170 :
2180 VDU 29,0;-&40;
2190 :
2200 REM Horiz. grid
2210 FOR I% = &60 TO &1E0 STEP &40
2220   MOVE I%,&3DF
2230   DRAW I%,&1E0
2240   NEXT
2250 :
2260 REM Vert. grid
2270 FOR I% = &220 TO &3A0 STEP &40
2280   MOVE &20,I%
2290   DRAW &21F,I%
2300   NEXT
2310 :
2320 REM Small box
2330 GCOL 0,1
2340 MOVE &110,&130
2350 PLOT 1,&40,0
2360 PLOT 1,0,-&40
2370 PLOT 1,-&40,0
2380 PLOT 1,0,&40
2390 VDU 29,&24;&364;
2400 ENDPROC
2410 :
2420 :
2430 DEF FNinput
2440 IN$ = ""
2450 SIZE = 0
2460 :
```

```
2470 REPEAT
2480   K$ = GET$
2490   OK = FALSE
2500   IF INSTR("0123456789",K$) AND
       SIZE < 3 THEN SIZE = SIZE + 1 :
       IN$ = IN$ + K$ : OK = TRUE
2510   IF K$ = CHR$127 AND SIZE > 0 THEN
       SIZE = SIZE - 1 : IN$ =
       LEFT$(IN$,SIZE) : OK = TRUE
2520   IF SIZE AND K$ = CHR$13 THEN OK =
       TRUE
2530   IF OK THEN PRINT K$; ELSE VDU 7
2540   UNTIL K$ = CHR$13
2550 :
2560 = VAL IN$
2570 :
2580 :
2590 DEF PROCinvert
2600 :
2610 FOR I% = 0 TO 7
2620   FOR J% = 0 TO 7
2630     BIT%(I%,J%) = NOT BIT%(I%,J%)
2640     NEXT
2650   NEXT
2660 :
2670 PROCcreate
2680 PROCfill
2690 ENDPROC
2700 :
2710 :
2720 DEF PROCmirror
2730 PROCtemp
2740 :
2750 FOR I% = 0 TO 7
2760   T% = 7 - I%
2770   FOR J% = 0 TO 7
2780     BIT%(I%,J%) = TEMP%(T%,J%)
2790     NEXT
2800   NEXT
2810 :
```

```
2820 PROCcreate
2830 PROCfill
2840 ENDPROC
2850 :
2860 :
2870 DEF PROCrotate
2880 PROCtemp
2890 :
2900 FOR I% = 0 TO 7
2910   T% = 7 - I%
2920   FOR J% = 0 TO 7
2930     BIT%(I%,J%) = TEMP%(J%,T%)
2940     NEXT
2950   NEXT
2960 :
2970 PROCcreate
2980 PROCfill
2990 ENDPROC
3000 :
3010 :
3020 DEF PROCtable (N)
3030 REM Show 48 characters
3040 :
3050 FOR I% = 0 TO 18 STEP 8
3060   FOR J% = 0 TO 30 STEP 2
3070     PRINT TAB(18 + I%,J%) N " "
         CHR$N;
3080     N = N + 1
3090     NEXT
3100   NEXT
3110 :
3120 TBL = N - 32
3130 ENDPROC
3140 :
3150 :
3160 DEF PROCtemp
3170 :
```

```
3180 FOR I% = 0 TO 7
3190   FOR J% = 0 TO 7
3200     TEMP%(I%,J%) = BIT%(I%,J%)
3210     NEXT
3220   NEXT
3230 :
3240 ENDPROC
3250 :
3260 :
3270 DEF PROCvdu23 (SCRN)
3280 IF SCRN THEN PRINT TAB(9,25) CHR$N
     TAB(0,28); ELSE VDU 2 : VDU 21
3290 PRINT "VDU 23,";N ",";
3300 IF SCRN THEN PRINT
3310 :
3320 FOR I% = 0 TO 7 STEP 2
3330   PRINT "&";
3340   FOR J% = 1 TO 0 STEP -1
3350     IF BYTE%(I%+J%) < &10 THEN
         PRINT; 0;
3360     PRINT; ~BYTE%(I%+J%);
3370     NEXT
3380   PRINT ";";
3390   IF I% = 2 AND SCRN THEN PRINT
3400   NEXT
3410 :
3420 IF NOT SCRN THEN PRINT : VDU 6 :
     VDU 3
3430 ENDPROC
```

CIRCLE DRAW AND FILL

This is a short BASIC program which demonstrates two ways in which to draw circles quickly and then fill them with solid colour.

So as not to slow the program down unnecessarily, we have not added any REM statements to the code itself and we have used one- and two-character integer variable names.

The radius of the circles is set by the variable **R%** in line 80.

PROCcircle1 uses trigonometry to calculate the x,y co-ordinates of one quadrant of the circle. If the x,y graphics origin is set to the centre of the circle with a **VDU 29** command, it is very easy to plot the other three quadrants at the same time, since they are all reflected images of the first quadrant.

PROCcircle2 uses Pythagoras to calculate the circumference. Again, the idea of mirroring the quadrants is used. Notice that **R% * R%** only needs to be evaluated once and so a new variable **R2%** is created outside of the drawing loop.

Sines and cosines produce the best-shaped circles, but the calculation time is longer than with the Pythagoras algorithm. (Compare the displayed times and the circumference lines, particularly around the ends of the horizontal diameter.) One way of reducing the drawing time is to carry out the calculations in advance of the drawing. Create two arrays **X%()** and **Y%()** and store the values of the x,y co-ordinates in them. Then, when it is time to draw the circles, a loop containing the single command **DRAW X%(I%),Y%(I%)** will show a noticeable increase in speed. As before, you could use the 'mirror' idea to save having to calculate more than one quadrant. It would also cut down the amount of memory that you would need for the array, although it will mean that you will still need the two nested loops for the plus and minus multiplications.

What you are effectively doing is creating a so-called 'look-up table'. BBC BASIC, in common with almost all other microcomputer BASICs, calculates trigonometric values whenever they are called for by the user's program. However, if you know that you are going to use a limited number of such values (such as for angles between 0 and 90 degrees in 5-degree steps) several times over in your program, you could effect a significant saving in drawing time by creating a look-up table. If your circles (or arcs or quadrants for that matter) have different radii, only store the sine and cosine values in the arrays. Since the values will be floating point numbers, you mustn't use integer arrays in this instance. When the program needs to do the drawing, do a simple multiplication of the radius against the array values inside of your drawing loop, e.g. **DRAW**

R% * X(I%), R% * Y(I%). If you experiment with a series of concentric circles, you'll soon see the improvement! We have used this technique in the SHAPER utility.

Both of our circle-drawing procedures end with a call to **PROCfill**. It utilises a facility available in O.S 1.2, the line-fill **PLOT** option. **PLOT** commands in the range 72 to 79 will cause a horizontal line to be drawn outwards in both directions from the x,y co-ordinate specified until there is a collision with a pixel of a different colour. One special requirement of the command is that the current foreground and background colours set by **GCOL** must be different, even if the area to be filled is already a different colour from the filling colour. The seven variations in the command are the same as for other **PLOT** commands (see page 319 of the User Guide).

```
 10 REM **** CIRCLE FILL ****
 20 :
 30 REM (c) Ian Trackman 1983
 40 :
 50 MODE 1
 60 VDU 23,1,0;0;0;0;
 70 :
 80 R% = &100
 90 :
100 TIME = 0
110 PROCcircle_1 (R%,&140,&200,1)
120 PRINT TAB(9,0); TIME
130 :
140 TIME = 0
150 PROCcircle_2 (R%,&3C0,&200,2)
160 PRINT TAB(28,0); TIME
170 END
180 :
190 :
200 DEF PROCcircle_1 (R%,X%,Y%,C%)
210 LOCAL A,x%,y%
220 :
230 VDU 29,X%;Y%;
240 GCOL 0,C%
250 x% = 0
```

```
260 y% = R%
270 :
280 FOR A = 0 TO RAD 91 STEP RAD 2
290   X% = R% * SIN A
300   Y% = R% * COS A
310   FOR QX% = -1 TO 1 STEP 2
320     FOR QY% = -1 TO 1 STEP 2
330       MOVE x% * QX%,y% * QY%
340       DRAW X% * QX%,Y% * QY%
350       NEXT
360     NEXT
370   x% = X%
380   y% = Y%
390   NEXT
400 :
410 PROCfill
420 ENDPROC
430 :
440 :
450 DEF PROCcircle_2 (R%,X%,Y%,C%)
460 LOCAL x%,y%,R2%
470 :
480 VDU 29,X%;Y%;
490 GCOL 0,C%
500 R2% = R% * R%
510 x% = 0
520 y% = R%
530 :
540 FOR X% = 0 TO R% STEP 4
550   Y% = SQR(R2% - X%*X%)
560   FOR QX% = -1 TO 1 STEP 2
570     FOR QY% = -1 TO 1 STEP 2
580       MOVE x% * QX%,y% * QY%
590       DRAW X% * QX%,Y% * QY%
600       NEXT
610     NEXT
620   x% = X%
630   y% = Y%
640   NEXT
650 :
```

```
660 PROCfill
670 ENDPROC
680 :
690 :
700 DEF PROCfill
710 :
720 FOR Y% = -R% TO R% STEP 4
730   PLOT 77,0,Y%
740   NEXT
750 :
760 ENDPROC
```

PROGRAM CROSS-REFERENCER

XREF is a machine-code utility which will produce a list of all the line-numbers of lines in a BASIC program which contain a variable, a keyword or text designated by you. Alternatively, instead of displaying the line numbers, it will list out the lines themselves.

There are two versions of the program on the tape, XREF is the version for use with tape-based computers. It resides between &E00 and &FFF. XREFDISK is for use with disks and is loaded between &1700 and &18FF. Please refer to 'Using the Programming Utilities' for installation instructions. Other than the addresses at which, the tape and disk versions of the programs operate identically.

The utility is co-resident, that is, it will remain in the computer's memory whilst you load, run and save BASIC programs, until you over-write it.

Before you use XREF, you'll need to set up three function keys - any keys will do. Program the keys:

***KEY 1 0|HFind?**
***KEY 2 CALL&E00|K|K|M**
***KEY 3 CALL&E06|K|M**

If you like, you can add a space after the question mark at the end of the first key's string. If you want to use keys other than 1, 2 and 3, you'll obviously use different key numbers when setting them up. If you are using the disk version of the program, the second and third key strings will be

***KEY 2 CALL&1700|K|K|M**
***KEY 3 CALL&1706|K|M**

The utility temporarily adds a new line 0 to your BASIC program (which it subsequently deletes) and so your program must not already contain a line 0. If it does, it will be lost.

To use the utility, press the first function key and the message **Find ?** will appear on the screen. Type in what you want to find (let's call it the 'target' from now on) followed by Return. Subject to what we have to say below about specifying numbers and keywords, you can enter whatever you want - variable names, keywords, text or any combination of them. The target is stored by the utility in half of the computer's keyboard input buffer, so don't enter more than 127 characters at a time or you'll start to over-write other buffers with undefined results.

What you have done at this point is to cause a new line 0 to be added to the program in memory, containing the target. (If you wonder why this is necessary, the reason is in order to use the parser in ROM to create tokens from BASIC keywords.)

There are a number of points to bear in mind from this process.

Be careful if your target might appear in different contexts in the program. For example, if you want to find the variable **ABZ**, and you give just the letter **A** as your target, every occurrence of that character in your program will be listed.

If you do not proceed to the second stage of the utility (by pressing the second function key), you will be left with an unwanted line 0, which will probably cause a syntax error unless you delete it before running the program.

The utility will search for the target in the exact form in which you have typed it. Therefore, be accurate - particularly with spaces.

One advantage of using the parser is that you can type in the truncated form of BASIC keywords. If you are looking for, say, **PRINT "HELLO"**, you can type: **P. "HELLO"**

Because your target is tokenised, you must include the full word (don't type **SUB 1000** instead of **GOSUB 1000**) and all necessary brackets in accordance with the list of tokens on pages 483-4 of the User Guide. For instance, if you want to find **LEFT$**, you must type **LEFT$(** and not just **LEFT$**.

The final point is that you must not begin your target with a number, since it will be parsed as part of the line number and you will add a new and unwanted line somewhere else in the program! Since numbers within a line can almost always be related to another command, e.g. **GOSUB**, or a mathematical symbol, include that as the start of the target.

Having entered your target, press the second function key if you want a 'full' listing or the third function key if you only want a line-number list.

If the utility cannot find your target anywhere in the BASIC program, it will respond with **Not Found**. (It will also give this message if you don't enter a target in the first place.) Otherwise, it will either list out the individual lines or produce a single list of line-numbers in which your target occurs.

If you are using the full listing option, XREF will stop after every line and wait for a key-press. We suggest that you use the space-bar. Do not use the function keys for these key-presses, since they will send a command line into the input buffer and create havoc. If you incorrectly use the function keys, you may see the message **Mistake** on the screen. However, a few more key-presses will make XREF crash and you'll have to press the Break key to recover.

XREF is not intended to be used with a printer. The reason is that it generates BASIC commands (which you may see flash on to the screen before a reverse line-feed obscures them) and these would be displayed in a print-out.

Please refer to 'Using the Programming Utilities' for notes on tacked-on bytes, hidden control characters and other general hints.

The utility contains routines which demonstrate how to generate BASIC commands from within a machine-code program. We decided to take this approach rather than to access the BASIC ROM directly, so that the utility would not be dependent on the existence of code at specific addresses in the BASIC ROM. There are also useful routines in XREF for hex to decimal number conversion and string searching.

```
   10 REM LISTER & CROSS REFERENCER
   20 :
   30 REM (c) Ian Trackman 1982
   40 :
   50 DIM msg(4),OSCL 21
   60 find$ = "Find ? "
   70 PROCoscli ("KEY 1 0|H" + find$)
   80 :
   90 page = &18
  100 :
  110 memloc   = &70 : REM &71
  120 linenum  = &72 : REM &73
  130 number   = &74 : REM &75
  140 mod10    = &76 : REM &77
  150 temp     = &78 : REM &79
  160 length   = &80
  170 lenfake  = &81
  180 size     = &82
  190 ysave    = &83
  200 flag     = &84
  210 chars    = &85
  220 :
  230 buffer = &780 : REM In keyboard
      buffer
  240 :
```

```
250 osrdch = &FFE0
260 osnewl = &FFE7
270 oswrch = &FFEE
280 osbyte = &FFF4
290 :
300 eol     = &0D
310 space   = ASC " "
320 numsize = 5 : REM For output formatting
330 :
340 org = &E00
350 PROCoscli ("KEY 2 CALL&" + STR$^org + "|K|K|M")
360 PROCoscli ("KEY 3 CALL&" + STR$^(org + 6) + "|K|M")
370 :
380 opt = 2
390 :
400 FOR I% = 0 TO opt STEP opt
410 P% = org
420 [
430 OPT I%
440 :
450  JMP entry1 \ List xref
460  JMP entry2 \ - " -  re-rentry
470  JMP entry3 \ Number xref
480 :
490 .entry1 LDY #ASC "0" \ Delete line 0
500  JSR bufchar
510  LDY #eol
520  JSR bufchar
530  JSR osnewl
540 :
550  JSR setup
560  BCC look \ OK
570  BCS finish \ Null string
580 \ End of first entry routine
590 :
600 :
```

```
610 \ Subsequent passes start here
620 .entry2 LDY #0
630 .msg1loop LDA msg(1),Y \ Delete
     "CALL"
640  BEQ look
650  JSR oswrch
660  INY
670  BNE msg1loop
680 :
690 .look JSR search
700  BCS finish \ End of program
710 :
720  JSR match \ If no carry from
     search
730  RTS \ Temporary exit to Basic
     interpreter
740 :
750 .finish LDY flag \ Target ever
     found ?
760  BEQ exit \ If not
770  RTS
780 :
790 .exit JSR notfound
800  JMP exit3
810 :
820 :
830 \ **** Line number
      cross-referencer
840 :
850 .entry3 JSR setup
860 :
870 .xref JSR search
880  BCS finish2 \ No match
890 :
900  LDY flag \ First find ?
910  BNE msg4done \ Else Y = 0
920 :
930 .msg4loop LDA msg(4),Y \ "Line"
940  BEQ msg4done
950  JSR oswrch
```

```
 960  INY
 970  BNE msg4loop
 980 :
 990 .msg4done LDA #0 \ eol delimiter
1000  PHA
1010  LDX #numsize
1020  STX chars \ ASCII digit counter
1030  DEC flag \ Set it
1040 :
1050  JSR convert
1060 :
1070  LDX chars
1080  BEQ display \ No padding needed
1090 :
1100 \ Pad to right justify
1110  LDA #space
1120 .blank PHA
1130  DEX
1140  BNE blank
1150 :
1160 .display PLA \ Unstack ASCII
1170  BEQ xref \ Continue looking after
      eof
1180  JSR oswrch
1190  JMP display
1200 :
1210 .finish2 LDY flag \ Success ?
1220  BNE exit2
1230  JSR notfound
1240 :
1250 .exit2 LDA #21 \ VDU off
1260  JSR oswrch
1270  LDY #ASC "0" \ Delete line 0
1280  JSR bufchar
1290 :
1300 .exit3 LDY #6 \ VDU on
1310  JSR bufchar
1320  LDY #eol
1330  JMP bufchar \ Exit to Basic
1340 :
```

```
1350 :
1360 \ **** Sub-routines ****
1370 :
1380 \ **** Put Y into keyboard
       buffer
1390 :
1400 .bufchar LDA #&8A
1410  LDX #0
1420  JMP osbyte
1430 :
1440 :
1450 \ **** Convert 2-byte hex
       line-number to decimal ASCII
1460 :
1470 .convert PLA \ Save return
      address
1480  STA temp
1490  PLA
1500  STA temp+1
1510 :
1520  LDA linenum
1530  STA number
1540  LDA linenum+1
1550  STA number+1
1560 :
1570 .convert2 DEC chars \ Only used in
      Number xref
1580  LDA #0
1590  STA mod10
1600  STA mod10+1
1610  LDX #&10 \ Double byte
1620  CLC
1630 :
1640 .divloop ROL number \ Bit 0
      (carry) becomes quotient
1650  ROL number+1
1660  ROL mod10
1670  ROL mod10+1
1680 :
```

```
1690  LDA mod10
1700  SEC
1710  SBC #10
1720  TAY \ Low byte
1730  LDA mod10+1
1740  SBC #0
1750  BCC deccount \ if dividend < divisor
1760 :
1770  STY mod10 \ Next bit of dividend = 1.
1780  STA mod10+1 \ Dividend = Dividend - divisor
1790 :
1800 .deccount DEX
1810  BNE divloop
1820 :
1830  ROL number \ Shift in last carry for quotient
1840  ROL number+1
1850 :
1860  LDA mod10
1870  ORA #ASC "0" \ ASCII mask
1880  PHA \ Stack it (starts at right-hand digit)
1890 :
1900  LDA number \ Continue if value <> 0
1910  ORA number+1
1920  BNE convert2
1930 :
1940  LDA temp+1 \ Restore stack
1950  PHA
1960  LDA temp
1970  PHA
1980  RTS
1990 :
2000 :
2010 \ **** Update line pointers
2020 :
```

```
2030 .endline LDA memloc
2040  CLC
2050  ADC length
2060  STA memloc
2070  BCC noadd
2080  INC memloc+1
2090 .noadd RTS
2100 :
2110 :
2120 \ **** Match found
2130 :
2140 .match BIT flag
2150  BMI getkey
2160 :
2170 \ Prepare for deleted line 0 on
      second entry (shorter program)
2180  LDA memloc
2190  SEC
2200  SBC lenfake
2210  STA memloc
2220  LDA memloc+1
2230  SBC #0
2240  STA memloc+1
2250  DEC flag \ Set it
2260  BNE keydone
2270 :
2280 .getkey JSR osrdch
2290  CMP #&1B \ escape ?
2300  BNE keydone
2310 :
2320  PLA \ Pop the stack
2330  PLA
2340  RTS \ and exit to BASIC
2350 :
2360 .keydone LDA #11 \ Reverse
      line-feed
2370  JSR oswrch
2380 :
2390  LDA #21 \ VDU off
2400  JSR oswrch
```

```
2410 :
2420 \ Set up end of "LIST" command
2430  LDA #0
2440  PHA
2450  LDA #eol
2460  PHA
2470  LDA #6 \ VDU on
2480  PHA
2490 :
2500  JSR convert
2510 :
2520  LDA #ASC "T"
2530  PHA
2540  LDA #ASC "S"
2550  PHA
2560  LDA #ASC "I"
2570  PHA
2580  LDA #ASC "L"
2590  PHA
2600 :
2610 .outnum PLA \ Unstack ASCII
2620  BEQ numdone
2630  TAY
2640  JSR bufchar
2650  JMP outnum
2660 :
2670 .numdone LDY #0
2680 .msg3loop LDA msg(3),Y \ CALL
      <LISTER>
2690  BEQ msg3done
2700  STY ysave
2710  TAY
2720  JSR bufchar
2730  LDY ysave
2740  INY
2750  BNE msg3loop
2760 :
2770 .msg3done RTS
2780 :
2790 :
```

```
2800 \ ***** Not found message
2810 :
2820 .notfound LDA msg(2),Y
2830  BEQ msg2done
2840  JSR oswrch
2850  INY
2860  BNE notfound
2870 :
2880 .msg2done RTS
2890 :
2900 :
2910 \ ***** Main search loop
2920 :
2930 .search JSR endline
2940  LDY #0
2950  LDA (memloc),Y
2960  CMP #&FF \ End of program flag
2970  BEQ endprog
2980 :
2990  STA linenum+1
3000  INY
3010  LDA (memloc),Y
3020  STA linenum
3030  INY
3040  LDA (memloc),Y
3050  STA length \ Offset to start of
      next line
3060  STY ysave
3070 :
3080 .tryagain LDX size
3090  INC ysave
3100  LDY ysave
3110 :
3120 .nextbyte LDA (memloc),Y
3130  CMP #eol
3140  BEQ search
3150 :
3160  CMP buffer,X
3170  BNE tryagain
3180  INY
```

```
3190  DEX
3200  BNE nextbyte
3210 :
3220  CLC \ As "match" flag
3230  RTS
3240 :
3250 .endprog SEC \ As flag
3260  RTS
3270 :
3280 :
3290 \ **** Transfer user's input to
       our buffer
3300 :
3310 .setup LDA #3 + LEN find$
3320  STA memloc
3330  LDA page
3340  STA memloc+1
3350 :
3360  LDY #0
3370  STY flag
3380  DEY
3390 .loop INY
3400  LDA (memloc),Y
3410  CMP #eol
3420  BNE loop
3430 :
3440  STY size
3450  TYA \ For Z-flag
3460  BEQ null \ Null input
3470  DEC size
3480 :
3490  LDX #0 \ Buffer,0 is never reached
      so drop eol in it
3500 .swap LDA (memloc),Y
3510  STA buffer,X
3520  INX
3530  DEY
3540  BNE swap
3550 :
3560  INY
```

```
3570  STY memloc
3580 :
3590  INY \ Always skip fake line 0
3600  LDA (memloc),Y
3610  STA length
3620  STA lenfake
3630  CLC \ OK flag
3640  RTS
3650 :
3660 .null SEC \ As flag
3670  RTS
3680 ]
3690 :
3700 PROCtext (1,CHR$11 + STRING$(10,"
     ") + CHR$eol)
3710 PROCtext (2,"Not found " + CHR$21)
3720 PROCtext (3,"CA." + STR$(org+3) +
     CHR$eol)
3730 PROCtext (4,"Line ")
3740 :
3750 NEXT
3760 :
3770 END
3780 :
3790 :
3800 DEF PROCoscli (A$)
3810 X% = OSCL MOD &100
3820 Y% = OSCL DIV &100
3830 $OSCL = A$
3840 CALL &FFF7
3850 ENDPROC
3860 :
3870 :
3880 DEF PROCtext (N,A$)
3890 msg(N) = P%
3900 $msg(N) = A$
3910 P% = P% + LEN(A$) + 1
3920 P%?-1 = 0
3930 ENDPROC
```

DISASSEMBLER

The program, called DISASS on the tape, is written in BASIC, so CHAIN or LOAD-and-RUN it.

The program sets the screen to Mode 6 with a blue background. (If you prefer a different Mode or a different background colour, you can easily make the changes at the beginning of the program and at the start of **PROCsetup**.)

The program asks you to enter the start and end addresses of the area of memory that you want disassembled. The addresses must be given in hexadecimal and may be anywhere in the range 0 to &FFFF (use upper-case letters for **A** to **F**). You can - but don't have to - type in leading zeros (**&E00** is as valid as **&0E00**). The program will reject invalid addresses.

You can find out the start and end addresses of a machine-code program by using the ***OPT 1,2** option when loading it from tape (see page 398 of the User Guide) or with the ***INFO** command for disk (see page 44 of the Disk System User Guide).

Bear in mind that the I/O areas are located between addresses &FC00 and &FE00. If you have certain input/output devices connected to your computer, accessing these addresses by disassembling their contents may cause unexpected results.

The disassembler will decode the contents of the memory, starting at the address that you have specified, and will display the results in up to five columns across the screen. The first column is the address (in hex) and the second column contains one, two or three bytes making up an opcode and any associated operand. The third column holds the opcode itself and any operand is shown in the fourth column. These columns correspond to the four columns that will be displayed if you assemble a listing with **OPT** set to 1 or 3 (see page 314 of the User Guide). For example:

1000	**20 EE FF**	**JSR**	**&FFEE**
Column 1	Column 2	Column 3	Column 4
Address	Hex bytes	Opcode	Operand

Relative addresses are resolved and shown as absolute, not offset, addresses.

If an address contains a byte that cannot be decoded - it may be data or an invalid opcode - the third column will contain three question marks. If the byte is in the range &20 to &7F, its ASCII character will also be printed out in the fifth column of the display.

The display is set to 'page mode' with a **VDU 14** command and so will stop and wait for the Shift key to be pressed at the end of each screen page. Disassembly will continue until your specified end address is reached or until you leave the program by pressing Escape. You can re-start simply by typing **RUN.**

If you want to print out the results of the disassembly on your printer, enable the printer with the usual **VDU 2** (or Control B) command before you run the program. You may also want to delete the Page Mode command which appears at the start of **PROCsetup.**

At the start of the program is a variable **SPOOL**, which is normally set to **FALSE**. If you set it to **TRUE**, the program will make use of the Operating System's ***SPOOL** facility in order to save the disassembly to disk or to tape in a file called **DUMP**. This time, the disassembly will start with the word **AUTO** in order to generate line-numbers when the listing is subsequently **EXEC**'d. The first two columns of the disassembly will not be saved.

Once the disassembly has been saved, you can create a BASIC program from it, ready for you to examine and/or edit, with the following steps:
Type **NEW** (to remove the Disassembler)
Rewind the tape
Type ***EXEC "DUMP"**
When no more lines appear on the screen, press Escape to leave **AUTO** Mode.
Add the usual square brackets around the listing and save the new program.

You can now start to add labels to the listings and to convert the absolute addresses to symbolic addresses by using the REPLACE program in the Toolbox.

Please refer to pages 402-3 and pages 442-9 of the User Guide for further details of the ***SPOOL** and ***EXEC** commands and of the creation of assembly language listings.

If you are using disks, remember that you need at least 64 free sectors on your disk (see page 64 of the Disk System User Guide). We suggest that you delete the **DUMP** file as soon as you have created the BASIC program in order to avoid any subsequent **Can't extend** errors.

```
 10 REM **** 6502 DISASSEMBLER ****
 20 :
 30 REM (c) Ian Trackman 1982
 40 :
 50 REM A% = address pointer
 60 REM B% = address contents (byte)
 70 REM D% = input digit counter
 80 REM N% = opcode index
 90 REM T% = opcode addressing type
100 :
110 SPOOL = FALSE
120 :
130 MODE 6
140 PROCsetup
150 PROCaddress : REM Get valid hex
    address
160 ON ERROR GOTO 280
170 :
180 IF SPOOL THEN *SPOOL "DUMP"
190 IF SPOOL THEN PRINT "AUTO"
200 :
210 REPEAT
220   PROCdecode
230   IF T% < 3 THEN A% = A% + 1
240   IF T% > 2 AND T% < 9 OR T% = 13
      THEN A% = A% + 2
250   IF T% > 8 AND T% < 13 THEN A% =
      A% + 3
260   UNTIL FALSE
270 :
280 ON ERROR OFF
290 IF SPOOL THEN *SPOOL
300 REPORT
310 PRINT " at line "; ERL
320 END
330 :
340 :
```

```
350 DEF PROCaddress
360 A% = 0
370 D% = 0
380 PRINT "Start address : &";
390 :
400 REPEAT
410   K$ = GET$
420   IF D% < 1 AND (K$ = CHR$13 OR K$
      = CHR$127) THEN VDU 7
430   IF D% = 4 AND K$ <> CHR$13 AND K$
      <> CHR$ 127 THEN VDU 7
440   IF K$ = CHR$127 AND D% > 0 THEN
      PRINT K$; : A% = A% DIV &10 : D%
      = D% - 1
450   IF K$ <> CHR$ 13 AND K$ < "0"
      THEN VDU 7
460   IF K$ <> CHR$ 127 AND K$ > "F"
      THEN VDU 7
470   IF K$ > "9" AND K$ < "A" THEN VDU
      7
480   IF D% < 4 AND K$ >= "0" AND K$ <=
      "9" THEN A% = A%*&10 + VAL K$ :
      PRINT K$; : D% = D% + 1
490   IF D% < 4 AND K$ >= "A" AND K$ <=
      "F" THEN A% = A%*&10 + ASC(K$) -
      55 : PRINT K$; : D% = D% + 1
500   UNTIL A% AND K$ = CHR$13
510 :
520 PRINT
530 ENDPROC
540 :
550 :
560 DEF PROCdecode
570 B% = ?A%
580 N% = OP%(B%) DIV 100
590 T% = OP%(B%) MOD 100
600 A$ = RIGHT$("000" + STR$~A%,4)
610 B$ = RIGHT$("0" + STR$~B%,2)
620 A1$ = RIGHT$("0" + STR$~A%?1,2)
630 A2$ = RIGHT$("0" + STR$~A%?2,2)
```

```
640 LO$ = A1$
650 HI$ = A2$
660 A1$ = A1$ + "  "
670 :
680 IF SPOOL THEN A1$ = "" : A2$ = ""
    ELSE PRINT A$ " : " B$ "  ";
690 IF N% = 0 THEN PRINT TAB(X%)
    OP$(N%);
700 IF N% = 0 AND B% > &20 AND B% < &80
    THEN PRINT, CHR$B%;
710 IF N% = 0 THEN PRINT : ENDPROC
720 IF T% = 1 THEN PRINT TAB(X%)
    OP$(N%) : ENDPROC
730 IF T% = 2 THEN PRINT TAB(X%)
    OP$(N%) " A" : ENDPROC
740 IF T% = 3 THEN PRINT A1$ TAB(X%)
    OP$(N%) " #&" LO$ : ENDPROC
750 IF T% = 4 THEN PRINT A1$ TAB(X%)
    OP$(N%) " &" LO$ : ENDPROC
760 IF T% = 5 THEN PRINT A1$ TAB(X%)
    OP$(N%) " &" LO$ ",X" : ENDPROC
770 IF T% = 6 THEN PRINT A1$ TAB(X%)
    OP$(N%) " &" LO$ ",Y" : ENDPROC
780 IF T% = 7 THEN PRINT A1$ TAB(X%)
    OP$(N%) " (&" LO$ ",X)" : ENDPROC
790 IF T% = 8 THEN PRINT A1$ TAB(X%)
    OP$(N%) " (&" LO$ "),Y" : ENDPROC
800 IF T% = 9 THEN PRINT; A1$ A2$
    TAB(X%) OP$(N%) " &" HI$ LO$ :
    ENDPROC
810 IF T% = 10 THEN PRINT A1$ A2$
    TAB(X%) OP$(N%) " &" HI$ LO$ ",X" :
    ENDPROC
820 IF T% = 11 THEN PRINT A1$ A2$
    TAB(X%) OP$(N%) " &" HI$ LO$ ",Y" :
    ENDPROC
830 IF T% = 12 THEN PRINT A1$ A2$
    TAB(X%) OP$(N%) " (&" HI$ LO$ ")" :
    ENDPROC
```

```
 840 PRINT A1$ TAB(X%) OP$(N%) " &";
 850 IF A%?1 < &80 THEN PRINT
     RIGHT$("000" + STR$~(A% + A%?1 +
     2),4) ELSE PRINT RIGHT$("000" +
     STR$~(A% + A%?1 - &FE),4) : REM
     Relative addressing
 860 ENDPROC
 870 :
 880 :
 890 DEF PROCsetup
 900 VDU 19,0,4;0;
 910 VDU 14
 920 DIM OP$(56),OP%(255)
 930 IF SPOOL THEN X% = 0 ELSE X% = 18 :
     REM Opcode column tab setting
 940 :
 950 FOR I% = 0 TO 56
 960   READ OP$(I%)
 970   NEXT
 980 :
 990 FOR I% = 0 TO 255
1000   READ OP%(I%)
1010   NEXT
1020 :
1030 ENDPROC
1040 :
1050 :
1060 DATA ???, ADC, AND, ASL, BCC, BCS,
     BEQ, BIT, BMI, BNE, BPL, BRK, BVC,
     BVS, CLC, CLD, CLI, CLV, CMP, CPX,
     CPY, DEC, DEX, DEY, EOR, INC
1070 DATA INX, INY, JMP, JSR, LDA, LDX,
     LDY, LSR, NOP, ORA, PHA, PHP, PLA,
     PLP, ROL, ROR, RTI, RTS, SBC, SEC,
     SED, SEI, STA, STX, STY, TAX, TAY,
     TSX, TXA, TXS, TYA
1080 :
```

```
1090 REM Opcode index
1100 REM n DIV 100 = opcode index : n
     MOD 100 = opcode addressing mode
1110 :
1120 DATA 1101, 3507, 0, 0, 0, 3504,
     0304, 0, 3701, 3503, 0302, 0, 0,
     3509, 0309, 0, 1013, 3508, 0, 0, 0,
     3505, 0305, 0, 1401, 3511, 0, 0, 0,
     3510, 0310, 0
1130 DATA 2909, 0207, 0, 0, 0704, 0204,
     4004, 0, 3901, 0203, 4002, 0, 0709,
     0209, 4009, 0, 0813, 0208, 0, 0, 0,
     0205, 4005, 0, 4501, 0211, 0, 0,0,
     0210, 4010, 0
1140 DATA 4201, 2407, 0, 0, 0, 2404,
     3304, 0, 3601, 2403, 3302, 0, 2809,
     2409, 3309, 0, 1213, 2408, 0, 0, 0,
     2410, 3310, 0, 1601, 2411, 0, 0, 0,
     2410, 3310, 0
1150 DATA 4301, 0107, 0, 0, 0, 0104,
     4104, 0, 3801, 0103, 4102, 0, 2812,
     0109, 4109, 0, 1313, 0108, 0, 0, 0,
     0105, 4105, 0, 4701, 0111, 0, 0, 0,
     0110, 4110, 0
1160 DATA 0, 4807, 0, 0, 5004, 4804,
     4904, 0, 2301, 0, 5401, 0, 5009,
     4809, 4909, 0, 0413, 4808, 0, 0,
     5005, 4805, 4906, 0, 5601, 4811,
     5501, 0, 0, 4810, 0, 0
1170 DATA 3203, 4807, 3103, 0, 3204,
     3004, 3104, 0, 5201, 3003, 5101, 0,
     3209, 3009, 3109, 0, 0513, 3008, 0,
     0, 3205, 3005, 3106, 0, 1701, 3011,
     5301, 0, 3210, 3010, 3111, 0
1180 DATA 2003, 1807, 0, 0, 2004, 1804,
     2104, 0, 2701, 1803, 2201, 0, 2009,
     1809, 2109, 0, 0913, 1808, 0, 0, 0,
     1805, 2105, 0, 1501, 1811, 0, 0, 0,
     1810, 2110, 0
```

```
1190 DATA 1903, 4407, 0, 0, 1904, 4404,
     2504, 0, 2601, 4403, 3401, 0, 1909,
     4409, 2509, 0, 0613, 4408, 0, 0, 0,
     4405, 2505, 0, 4601, 4411, 0, 0, 0,
     4410, 2510, 0, 0
```

DOUBLE-SIZE CHARACTERS

Two routines give you the ability to print double-height characters in Modes 1 to 5 in the same way as you can in Teletext Mode 7 by using CHR$141. You can't usefully use them in Modes 3 or 6, since there will be a horizontal gap in the middle of each enlarged letter.

The routines themselves are in machine-code, but they are embodied in two demonstration programs which are written in BASIC. The idea is for you to be able to see how the routines operate and then incorporate them into your own programs.

The first program is called GIANT1 and is intended for use with Modes 0, 1 and 4.

It needs four user-definable characters. In the demonstration, we have selected characters 224 to 227, but you can set the variable **ascii** at the beginning of the program to use any four free characters.

100 bytes are reserved for the machine-code. (It actually assembles in 94 bytes.) The routine uses a number of locations in zero-page as detailed at the start of the assembly language listing.

The character to be printed could be passed to the machine-code in the form of a parameter to the **CALL** but, since it is a single byte, we think that it is just as easy to 'poke' it into a defined location, **char**, which is set up during the assembly process.

We imagine that you will want to use the routine most of the time to display strings which are pre-defined in your program and so the demonstration begins by doing just that.

We call the machine-code subroutine to set up the new user-definable characters which we then print out, with connecting cursor-move control characters, by using BASIC's **VDU** command. We could have added the printing commands into the machine-code, but we thought that by leaving the four 'quarter' characters defined (but not yet printed), this would give you more flexibility, e.g. to change colours between each quarter.

Notice in particular that the first item in the **VDU** string moves the cursor up a line. The reason for this is that we want to leave the cursor in its 'correct' position after printing, which is one line too low for the next character. That means that you must always start at the top of the screen with a line-feed or **PRINT** command (see line 140) in order to cancel out the effect of the very first **VDU 11**.

In case you should want to print double-size characters in response to keyboard input, the demonstration continues by taking individual characters from the keyboard and displaying them in double size on the screen. Since **GET** is used for this, we need to deal specially with Return **(CHR$13)**. We have also partly simulated the 'Delete' key, although you will have to develop the demonstration routine further to handle back-spacing beyond the left-hand edge of a line. If you want to use double-size characters for a full display of keyboard entry, you will also have to add program lines to handle the cursor-move and copy keys, since they do not work at double-spaced line intervals.

The second program, which is called GIANT2, works in Modes 2 and 5 on the same principles as the first program.

Since characters in these Modes are already double-width, only two characters are needed in order to make up a new double-height character and so the machine-code and **VDU** commands are correspondingly different. The machine-code is shorter and takes up only 45 bytes.

As before, if you want to use the routine in response to keyboard input, you'll have to write code to cope with the cursor-move and edit keys.

Of course, there is no reason why you shouldn't use either of the two routines in the 'wrong' Mode to produce extra-fat or tall, thin characters.

```
 10 REM DOUBLE HEIGHT CHARACTERS
 20 :
 30 REM (c) Ian Trackman 1983
 40 :
 50 REM Prints double height characters
    in Modes 0 - 5
 60 :
 70 MODE 1
 80 :
 90 ascii = 224 : REM Any four free
    ASCII characters
100 C = ascii
110 :
120 PROCassemble
130 :
140 PRINT TAB(6,2);
150 PROCdemo ("BIG CHARACTERS")
160 PRINT ''
170 :
```

```
180 REPEAT
190   K = GET
200   IF K = 13 THEN PRINT : K = FALSE
      : REM Return
210   IF K = 127 THEN VDU 8,8 : K = ASC
      " " : DEL = TRUE ELSE DEL = FALSE
      : REM delete
220   ?char = K
230   CALL Mcode
240   IF K THEN VDU
      11,C,C+2,8,8,10,C+1,C+3 : REM
      Print the 4 characters with
      cursor moves
250   IF DEL THEN VDU 8,8
260   IF POS = 0 THEN PRINT : REM New
      double line
270   UNTIL FALSE
280 :
290 END
300 :
310 :
320 DEF PROCassemble
330 DIM Mcode 100
340 char = &70 : REM to &78
350 temp = &79 : REM to &80
360 vdu_char = &81
370 :
380 oswrch = &FFEE
390 osword = &FFF1
400 :
410 opt = 2 : REM No display
420 :
430 FOR I% = 0 TO opt STEP opt
440   P% = Mcode
450   [OPT I%
460   :
470   \ Call OS 'read character
        definition' routine with ASCII
        in 'char'
480    LDA #&A
```

```
490     LDX #char
500     LDY #char DIV &100
510     JSR osword
520    :
530     LDA #ascii \ First character
540     STA vdu_char
550    :
560     JSR create
570     JSR vdu23
580     INC vdu_char
590     JSR vdu23
600     INC vdu_char
610     JSR create
620     JSR vdu23
630     INC vdu_char
640     JSR vdu23
650     RTS
660    :
670    .create LDX #8 \ 8 bytes ...
680    .loop1 LDY #4 \ ... in two
        passes
690     LDA #0
700    :
710    .loop2 ROL char,X \ Examine each
        bit
720     BCS carry \ 1 bit ?
730    :
740     ROL A
750     CLC \ Repeat 0 bit
760     BCC next \ Always
770    :
780    .carry ROL A
790     SEC \ Repeat 1 bit
800    :
810    .next ROL A \ Save it
820     DEY
830     BNE loop2
840    :
850     STA temp-1,X \ Store new byte in
        temp array (X = 1 to 8 so -1)
```

```
 860      DEX
 870      BNE loop1
 880      RTS
 890     :
 900     .vdu23 LDA #23
 910      JSR oswrch
 920      LDA vdu_char
 930      JSR oswrch
 940      LDY #2 \ Print 8 bytes ...
 950      JSR print
 960      LDY #2 \ Fall in for second
          print
 970     :
 980     .print LDA temp,X
 990      JSR oswrch \ double each byte
1000      JSR oswrch
1010      INX
1020      DEY
1030      BNE print
1040      RTS
1050     :
1060     ]
1070     :
1080     NEXT
1090   :
1100   ENDPROC
1110   :
1120   :
1130   DEF PROCdemo (A$)
1140   :
1150   FOR I% = 1 TO LEN A$
1160     ?char = ASC MID$(A$,I%,1) : REM
         Pass ASCII to the machine code
         via this byte
1170     CALL Mcode
1180     VDU 11,C,C+2,8,8,10,C+1,C+3 : REM
         Print the 4 characters with
         cursor moves
1190     NEXT
1200   :
1210   ENDPROC
```

```
   10 REM **** CHARACTER EXPANDER ****
   20 :
   30 REM (c) Ian Trackman 1983
   40 :
   50 REM Prints double height characters
      in Modes 2 and 5
   60 :
   70 MODE 2
   80 :
   90 ascii = 224 : REM Any two free
      ASCII characters
  100 :
  110 PROCassemble
  120 :
  130 PRINT TAB(3);
  140 PROCdemo ("BIG CHARACTERS")
  150 PRINT ''
  160 :
  170 REPEAT
  180   K = GET
  190   IF K = 13 THEN PRINT : K = FALSE
  200   IF K = 127 THEN PRINT CHR$8; : K
        = ASC " " : DEL = TRUE ELSE DEL =
        FALSE
  210   ?char = K
  220   CALL Mcode
  230   COLOUR RND(7)
  240   IF K THEN VDU
        11,ascii,8,10,ascii+1
  250   IF DEL THEN PRINT CHR$8;
  260   IF POS = 0 THEN PRINT : REM New
        double line
  270   UNTIL FALSE
  280 :
  290 END
  300 :
  310 :
  320 DEF PROCassemble
  330 DIM Mcode 50
  340 char = &70 : REM to &78
```

```
350 vdu_char = &79
360 :
370 oswrch = &FFEE
380 osword = &FFF1
390 :
400 opt = 2 : REM No display
410 :
420 FOR I% = 0 TO opt STEP opt
430   P% = Mcode
440   [OPT I%
450   :
460   \ Call OS 'read character
        definition' routine with ASCII
        in 'char'
470    LDA #&A
480    LDX #char
490    LDY #char DIV &100
500    JSR osword
510   :
520    LDA #ascii \ First character
530    STA vdu_char
540   :
550    LDX #1
560    JSR vdu23
570    INC vdu_char \ Fall through on
       second pass
580   :
590   .vdu23 LDA #23
600    JSR oswrch
610    LDA vdu_char
620    JSR oswrch
630    LDY #4
640   :
650   .print LDA char,X
660    JSR oswrch \ double each byte
670    JSR oswrch \ double each byte
680    INX
690    DEY
700    BNE print
710    RTS
```

```
720    :
730    ]
740    NEXT
750 :
760 ENDPROC
770 :
780 :
790 DEF PROCdemo (A$)
800 :
810 FOR I% = 1 TO LEN A$
820    ?char = ASC MID$(A$,I%,1) : REM
       Pass ASCII to the machine code
       via this byte
830    CALL Mcode
840    VDU 11,ascii,8,10,ascii+1
850    NEXT
860 :
870 ENDPROC
```

GRAPHICS DUMP

GRAFPRT is a machine-code utility which enables you to 'dump' the contents of the screen to a graphics printer. As written, it works with the Epson MX80, but if you can program reasonably well in assembly language, you should have little difficulty in adapting the program to work with a different printer.

The program resides at &E00 and takes up less than &100 bytes. Once you have loaded it into memory **(*LOAD GRAFPRT)**, a **CALL &E00** command will start it working, assuming, of course, that your printer is connected and running and that you have already set up any necessary output protocols (e.g. ***FX 6** to enable line-feeds).

GRAFPRTDSK is the disk version of the program and loads at &1800.

Since the program copies whatever is on the screen, you cannot produce your picture then type **CALL &E00**, since the command line will appear on the screen. There are a number of ways around this. The simplest is to call the routine from within the program which creates the screen image, immediately after it has been drawn. Another idea is to add something like **ON ERROR CALL &E00** to your program so that, by pressing Escape, you can take a 'snapshot' of the screen when you want. However, you might not want a graphics dump every time that you press Escape and a slightly more sophisticated approach would be to create an error trap, to which the program is directed by an **ON ERROR** command. Having jumped to the error trap, you could then test whether a graphics dump is really required with a line like:
IF GET$ = "G" THEN CALL &E00

So that you can adapt the program, here is a description of how it works. The program starts by calling OSBYTE &8A. This is 'officially' intended to return the position of the text cursor, but it also has the very useful side effect of setting the Y register to the Mode number. The Y register is tested to see whether we are in a non-text Mode (3, 6 or 7), in which case the routine exits with a 'beep'.

If we have a graphics Mode, we do a **VDU 2** to send output to the printer.

Our next task is to differentiate between Mode 0 and all of the other Modes, since the printer must be set to 'double-density' in order to handle Mode 0's high resolution. We also have to work out the pixel spacing of the different Modes. Both conversions are done by machine-code 'compares' and using the carry bit to set appropriate flag bytes.

We next send set-up commands to the printer and this is where you might need to start making changes for other printers.

The body of the program consists of a loop, in which the program reads the screen pixels one at a time, starting at co-ordinate 0,0 and working its way through to co-ordinate &4FF, &3FF.

Any pixel which is not black (i.e. 0) is added to an eight-bit byte for transmission to the printer. You could easily change the test in order to print dots of only one screen colour.

At the end of each line, we send a line-feed followed by the commands to reset graphics Mode on the printer. Here again, you will need to make changes for other printers.

At the end of the dump, the printer is disconnected with a **VDU 3** command and the routine ends. If you want to stop the print-out before it has been completed, you'll have to press Break, since the routine does not slow down to test whether Escape has been pressed.

```
  10 REM **** SCREEN GRAPHICS DUMP ****
  20 REM FOR EPSON MX 80 PRINTER
  30 :
  40 REM (c) Ian Trackman 1983
  50 :
  60 REM Next 5 bytes are also the block
     for osword call
  70 xloc   = &70 : REM &71
  80 yloc   = &72 : REM &73
  90 pixval = &74
 100 :
 110 mode = &80
 120 step = &81
 130 bits = &82
 140 byte = &83
 150 :
 160 esc  = 27
 170 :
 180 oswrch = &FFEE
 190 osword = &FFF1
 200 osbyte = &FFF4
 210 :
 220 ytop = &3FF
 230 :
```

```
240 org = &1800
250 :
260 opt = 2
270 :
280 FOR I%= 0 TO opt STEP opt
290   P% = org
300   [
310   OPT I%
320   :
330   .dump LDA #&87 \ Get text cursor
       position ...
340    JSR osbyte
350    TYA \ ... then Y has Mode
       number
360    CPY #3 \ Text only Mode 3 ?
370    BEQ error \ If so, error
380   :
390    CPY #6 \ Mode 6 or 7 ?
400    BCC setup \ Carry on if less
410   :
420   .error LDA #7 \ Beep and ...
430    JMP oswrch \ exit
440   :
450   .setup LDA #0
460    STA mode \ Assume Mode 0
470    LDA #2
480    STA step
490    JSR oswrch \ VDU 2
500    CPY #1 \ Clears carry if Mode 0
510    BCC save_mode \ Yes, really Mode
       0
520   :
530    ASL step \ Otherwise, double
       step value to 4
540    SEC \ and reset the carry
550   :
560   .save_mode ROR mode \ Save the
       carry status as Mode flag
```

```
570   :
580   \ Set line spacing
590    LDA #esc
600    JSR print
610    LDA #ASC "A"
620    JSR print
630    LDA #8
640    JSR print
650   :
660   \ Start at top of screen
670    LDA #ytop MOD &100
680    STA yloc
690    LDA #ytop DIV &100
700    STA yloc+1
710   :
720   \ Main loop starts here
730   :
740   \ Send printer graphics
        command
750   .newline LDA #esc
760    JSR print
770    BIT mode
780    BMI over_0
790   :
800   \ Mode 0 only
810    LDA #ASC "L" \ Dual density
820    LDX #&80 \ Characters per line
830    BNE setlen \ Always
840   :
850   \ Printer commands for other
        Modes
860   .over_0 LDA #ASC "K" \ Single
       density
870    LDX #&40 \ Characters per line
880   :
890   .setlen JSR print
900    TXA \ Characters (&40 or &80)
910    JSR print
920   :
```

```
 930    \ Reset to left-hand edge
 940     LDA #0
 950     STA xloc
 960     STA xloc+1
 970    :
 980    .newcolumn LDA #7 \ Bits 0 to 7
 990     STA bits \ Counter
1000    :
1010    .readpixel LDA #9 \ Read pixel
1020     LDX #xloc \ Point to 4 XY bytes
1030     LDY #0
1040     JSR osword
1050     LDA pixval
1060     CMP #1 \ Sets carry if not 0
         (black)
1070    :
1080    .setbyte ROL byte
1090     LDA yloc
1100     SEC
1110     SBC #4 \ All modes descend 4
         pixels
1120     STA yloc
1130     BCS decbits \ Test page
         roll-over
1140     DEC yloc+1
1150    :
1160    .decbits DEC bits \ Counter - 1
1170     BPL readpixel \ Done 8 bits ?
1180    :
1190     LDA byte \ Print 8 bits
1200     JSR print
1210    :
1220    \ Next column
1230     LDA xloc
1240     CLC
1250     ADC step
1260     STA xloc
1270     LDA xloc+1
1280     ADC #0
```

```
1290      STA xloc+1
1300      CMP #5 \ Over &4FF ?
1310      BEQ endline \ If so, end of
          line
1320     :
1330     \ Move along to next column
1340     .columntop LDA yloc
1350      CLC
1360      ADC #&20 \ 4 steps * 8 bits
1370      STA yloc
1380      BCC newcolumn
1390      INC yloc+1
1400      BCS newcolumn \ Always
1410     :
1420     .endline LDA #10 \ Line-feed
1430      JSR print
1440      BIT yloc+1 \ Test msb
1450      BPL newline \ Under 0000 ?
1460     :
1470     \ Switch off printer
1480      LDA #3 \ VDU 3
1490      JMP oswrch \ and exit
1500     :
1510     \ Print subroutine
1520     .print PHA \ Save value sent
          here
1530      LDA #1 \ VDU 1
1540      JSR oswrch
1550      PLA \ Recover original byte
1560      JMP oswrch \ JSR + RTS
1570     ]
1580     :
1590     NEXT
1600 :
1610 END
```

MULTI-STATEMENT LINE PACKER

This is one of the three 'squeeze' utilities which will help to shorten a BASIC program and so make it run faster. It packs as many statements as possible on to multi-statement lines, whilst ensuring that the program remains grammatically correct.

There are two versions of the program on the tape. PACKER is the version for use with tape-based computers and resides between &E00 and &FFF. PACKERDISK is for use with disks and is loaded between &1700 and &18FF. Please refer to 'Using the Programming Utilities' for installation instructions. Other than the addresses at which they start, the two programs operate identically.

As the utility is co-resident, you can load it before or after you get your BASIC program into memory. Once the utility is in memory, start it working with **CALL &E00** (tape version) or **CALL &1700** (disk version). On disk-based systems, it is the last of the utilities used in the ***EXEC SQUEEZE** routine.

The routine begins by setting Mode 7 and displaying the message **Packing** . . . . After a short while - just how long depends on the length of your program and how much packing needs to be done - the prompt will return, leaving the packed BASIC program in memory.

If you are squeezing a program to its smallest size, PACKER should be the last utility that you use, following after REMSTRP and CRUNCH. When you use REMSTRP, remove single-colon lines, otherwise they will be included in the packing process.

PACKER will put a maximum of 237 bytes into a line of BASIC. It calculates the length of the first line, adds the length of the next line to it, and tests whether the limit has been reached. If not, it adds the two lines together and tries to add on another line, progressing in this way through the program. Since it deals in complete lines, rather than in statements separated by colons, it will achieve the optimum result if you start off with a fully unpacked program, consisting of single-statement lines (except for multiple statements following an **IF** or **ELSE**). We recommend that you should always program this way, for ease of editing and debugging. If your program does contain multi-statement lines, use UNPACK before you use PACKER.

There are certain situations when lines must not be packed and PACKER handles these properly. Nothing further will be added to a line after an **IF, REM, DATA** or **ON ERROR** statement. **DEF** statements will always be retained at the start of a new line.

PACKER will start a new line after a line starting with an asterisk (indicating a call to the Operating System). To prevent packing, the asterisk must be the first byte of the line (with no spaces in front of it). If you start with a multi-statement line, such as:

VDU 7 : *FX 15,1

(beep and clear the keyboard buffer), the asterisk will not be at the start of the line and another line could be added, so causing a syntax error when the program is run.

Line-references are also checked. If the program refers to a line-number (e.g. with a **GOTO, GOSUB** or **THEN**), PACKER will ensure that the line in question is not joined up into an.earlier line, as this would otherwise cause a **No such line** error. Of course, if you write well-structured programs using procedures, you won't have **GOTO**s and **GOSUB**s in your programs in the first place, will you? One case which PACKER cannot handle is that of the computed line-reference - **GOTO line** - where **line** is a variable computed at run-time. If you use computed line-references, you are asking for trouble!

Please refer to 'Using the Programming Utilities' for notes on tacked-on bytes, hidden control codes and other general hints.

```
   10 REM **** PACKER ****
   20 :
   30 REM (c) Ian Trackman 1982
   40 :
   60 REM Re-referencing fails on
      untokenized line number e.g. ON A
      GOTO 10,X,20
   70 REM Assumes no hidden &8Ds embedded
      in text
   80 :
  110 DIM msg(1)
  120 :
```

```
130 REM Basic pointers
140 lomem   = &0
150 himem   = &6
160 vartop = &2
170 top     = &12
180 page    = &18
190 :
200 memloc   = &70 : REM + &71
210 linenum = &72 : REM + &73
220 srchloc = &74 : REM + &75
230 binnum   = &76 : REM + &77, &78
240 source   = &7C : REM + &7D
250 destin   = &7E : REM + &7F
260 flag     = &80
270 length   = &81
280 srchlen = &82
290 newlength = &83
300 count    = &84
310 ysave    = &85
320 offset   = &86
330 :
340 oswrch = &FFEE
350 osnewl = &FFE7
360 :
370 REM Constants
380 eol    = &0D
390 space = ASC " "
400 colon = ASC ":"
410 star   = ASC "*"
420 rem    = &F4 : REM Basic tokens
430 if     = &E7
440 data   = &DC
450 error = &85
460 def    = &DD
470 maxsize = &7C : REM for Mode 7
480 :
490 org = &E00
495 :
500 opt = 2
510 :
```

```
520 FOR I% = 0 TO opt STEP opt
530 P% = org
540 [
550 OPT I%
560 :
570  LDY #0
580 .msg1loop LDA msg(1),Y \ Mode 7
     and title
590  BEQ msg1done
600  JSR oswrch
610  INY
620  BNE msg1loop
630 :
640 \ Himem under Mode 7
650 .msg1done LDA #maxsize
660  STA himem+1
670  LDX #0
680  STX himem
690 :
700  INX \ Start at PAGE + 1
710  STX memloc
720  LDA page
730  STA memloc+1
740 :
750 .nextline LDY #0
760  LDA (memloc),Y
770  CMP #&FF \ End of program flag
780  BNE morelines
790  JMP finish
800 :
810 .morelines INY
820  INY
830  STY offset
840  LDA (memloc),Y
850  STA length \ Offset to start of
     next line
860 :
```

```
 870 .nextbyte INY
 880  LDA (memloc),Y
 890  CMP #eol
 900  BEQ measure
 910 :
 920  CMP #if
 930  BEQ endline
 940 :
 950  CMP #rem
 960  BEQ endline
 970 :
 980  CMP #data
 990  BEQ endline
1000 :
1010  CMP #error
1020  BEQ endline
1030 :
1040  CMP #star
1050  BNE nextbyte
1060 :
1070 \ Is it OS call ?
1080  STY ysave
1090 .teststar DEY
1100  CPY offset \ Start of original
      line
1110  BEQ endline
1120 :
1130  LDA (memloc),Y
1140  CMP #space
1150  BEQ teststar
1160 :
1170  CMP #colon
1180  BEQ teststar
1190 :
1200  LDY ysave
1210  BNE nextbyte \ Always
1220 :
```

```
1230 .endline LDA memloc
1240  CLC
1250  ADC length
1260  STA memloc
1270  LDA memloc+1
1280  ADC #0
1290  STA memloc+1
1300  BNE nextline \ Always
1310 :
1320 .measure LDY length
1330  LDA (memloc),Y
1340  CMP #&FF
1350  BNE measure2
1360  JMP finish
1370 :
1380 .measure2 INY
1390  INY
1400  LDA (memloc),Y \ Next line's length
1410  CLC
1420  ADC length
1430  BCS endline \ Too big to pack
1440  CMP #&F0
1450  BCS endline \ Too big to pack
1460 :
1470  SEC
1480  SBC #3 \ Lose start of next line
1490  STA newlength
1500 :
1510 \ Is next line DEF or DATA ?
1520 .defchek INY
1530  LDA (memloc),Y
1540  CMP #space
1550  BEQ defchek
1560 :
1570  CMP #colon
1580  BEQ defchek
1590 :
```

```
1600  CMP #def
1610  BEQ endline \ Mustn't pack
1620 :
1630  CMP #data
1640  BEQ endline \ Mustn't pack
1650 :
1660 \ Is next line referred to ?
1670  LDY length
1680  LDA (memloc),Y
1690  STA linenum+1
1700  INY
1710  LDA (memloc),Y
1720  STA linenum
1730 :
1740 \ Convert linenum to 3-byte code
1750 .convert LDA linenum+1
1760  ORA #&40
1770  STA binnum+2
1780  LDA linenum
1790  AND #&3F
1800  ORA #&40
1810  STA binnum+1
1820  LDA linenum
1830  AND #&C0
1840  STA linenum
1850  LDA linenum+1
1860  AND #&C0
1870  LSR A
1880  LSR A
1890  ORA linenum
1900  LSR A
1910  LSR A
1920  EOR #&54
1930  STA binnum
1940 :
1950 \ Look for that number in
       program
1960  LDA #1
1970  STA srchloc
1980  LDA page
```

```
1990  STA srchloc+1
2000 :
2010 .nextsrch LDY #0
2020  LDA (srchloc),Y
2030  CMP #&FF
2040  BNE skipnum
2050  JMP pack \ Not referred to
2060 :
2070 .skipnum INY
2080  INY
2090  LDA (srchloc),Y
2100  STA srchlen
2110 :
2120 .moresrch INY
2130  LDA (srchloc),Y
2140  CMP #eol
2150  BEQ srchline
2160 :
2170  CMP #&8D \ Line number token
2180  BNE moresrch
2190 :
2200 \ Compare with 3 coded bytes
2210  LDX #0
2220  STX flag
2230 :
2240 .trymatch INY
2250  LDA (srchloc),Y
2260  CMP binnum,X
2270  BEQ morematch
2280  DEC flag
2290 :
2300 .morematch INX
2310  CPX #3
2320  BNE trymatch
2330 :
```

```
2340  LDA flag
2350  BNE srchline
2360  JMP endline \ Match, so can't
      pack
2370 :
2380 .srchline LDA srchloc
2390  CLC
2400  ADC srchlen
2410  STA srchloc
2420  LDA srchloc+1
2430  ADC #0
2440  STA srchloc+1
2450  BNE nextsrch \ Always
2460 :
2470 :
2480 \ Packing routine
2490 .pack LDY length \ Overlay eol
      with colon
2500  DEY
2510  LDA #colon
2520  STA (memloc),Y
2530 :
2540  LDA memloc
2550  CLC
2560  ADC length
2570  STA destin
2580  LDA memloc+1
2590  ADC #0
2600  STA destin+1
2610 :
2620  LDA destin
2630  CLC
2640  ADC #3
2650  STA source
2660  LDA destin+1
2670  ADC #0
2680  STA source+1
2690 :
```

```
2700  LDA top
2710  SEC
2720  SBC source
2730  STA count
2740  LDA top+1
2750  SEC
2760  SBC source+1
2770  LDY #0
2780  TAX \ Pages to move
2790  BEQ shift2
2800 :
2810 .shift1 LDA (source),Y
2820  STA (destin),Y
2830  INY
2840  BNE shift1
2850  INC source+1
2860  INC destin+1
2870  DEX
2880  BNE shift1
2890 :
2900 .shift2 LDX count \ Move odd
      bytes
2910  BEQ shiftdone
2920 :
2930 .shift3 LDA (source),Y
2940  STA (destin),Y
2950  INY
2960  DEX
2970  BNE shift3
2980 :
2990 \ Reset pointers
3000 .shiftdone LDY #2
3010  LDA newlength
3020  STA (memloc),Y
3030  LDY length \ Point where we
      stopped
3040  DEY \ Look at first new byte
3050  STY offset
```

```
3060  STA length \ Now update length for
      endline's use
3070  LDA top
3080  SEC
3090  SBC #3
3100  STA top
3110  STA lomem
3120  STA vartop
3130  LDA top+1
3140  SBC #0
3150  STA top+1
3160  STA lomem+1
3170  STA vartop+1
3180  JMP nextbyte
3190 :
3200 .finish JSR osnewl
3210  JMP osnewl \ & exit to caller
3220 ]
3230 :
3240 PROCtext (1, CHR$22 + CHR$7 +
     CHR$31 + CHR$13 + CHR$13 + "Packing
     .... ")
3250 :
3260 NEXT
3270 :
3280 END
3290 :
3291 :
3300 DEF PROCtext (N,A$)
3310 msg(N) = P%
3320 $msg(N) = A$
3330 P% = P% + LEN(A$) + 1
3340 P%?-1 = 0
3350 ENDPROC
```

RAM TEST

The program is written in machine-code.

It is less than &100 bytes long and so there is room for it in the stack! Although that is a most unusual place to store a program, we have done so in order that the program can test as much RAM as possible without over-writing itself.

Call the program either by ***LOAD RAMTEST** followed by **CALL &100** or by ***RUN RAMTEST**. On disk-based systems, you can type ***RAMTEST**.

Mode 7 will be set and you will see an asterisk progressing across the screen. In a short while, small coloured squares will appear and then the whole screen will fill with changing characters in ASCII order. The display will continue to cycle in this manner until you stop the program by pressing Break (Escape won't work).

What is happening is that the entire RAM from &900 to &7FFF is being filled by bytes from 0 to &FF. After each new number is used, the entire memory is checked to see whether the contents of any byte have changed. What you are seeing on the screen is the Mode 7 display area (between &7C00 and &7FFF) being filled with these values, some of which correspond to control codes and some to ASCII characters.

At the end of every complete cycle, the speaker will beep to indicate that all 256 different bit patterns have been successfully tested.

Should a failure occur, the program will stop with a beep and the prompt will return. The address of the bad byte is in locations &70 and &71.
PRINT ~&70 + ?&71 * &100
will put it on to the screen. The value which caused the failure is held in location &72.

It is not possible to test the memory below &900 with this program, since that area is used by the Operating System to store its control variables, vectoring addresses and the like. If you fill it with different byte values, the computer tends to crash to a sudden halt!

Since various buffers are tested, it is best to clear the system completely with a CNTRL BREAK when you have finished with the test.

```
 10 REM **** RAM TEST ****
 20 :
 30 REM (c) Ian Trackman 1983
 40 :
 50 loc  = &70 : REM + &71
 60 byte = &72
 70 :
 80 oswrch = &FFEE
 90 osbyte = &FFF4
100 :
110 start = 9 : REM Start at &900
120 pages = &80 - start
130 :
140 opt = 2
150 :
160 FOR I% = 0 TO opt STEP opt
170 P% = &100
180 [
190 OPT I%
200 :
210  LDX #&FF
220  TXS \ Reset stack
230 :
240 \ Flush function key buffer
250  LDA #18
260  JSR osbyte
270 :
280 \ Mode 7
290  LDA #&16
300  JSR oswrch
310  LDA #7
320  JSR oswrch
330 :
340 \ Cursor off
350  LDA #23
360  JSR oswrch
370  LDA #1
380  JSR oswrch
390  LDA #0
400  LDX #8
```

```
410 :
420 .zero JSR oswrch
430  DEX
440  BNE zero
450 :
460 \ Set up to start
470  LDA #0
480  STA loc
490  STA byte \ Test byte
500  TAY
510 :
520 .begin LDA #start
530  STA loc+1
540  LDX #pages
550 :
560 \ Store a byte throughout memory
570 .loop1 LDA byte
580  STA (loc),Y
590  INY
600  BNE loop1
610 :
620  INC loc+1
630  DEX \ Decrement page counter
640  BNE loop1 \ All pages done ?
650 :
660 \ Now check the byte
670  LDA #start
680  STA loc+1
690  LDX #pages
700 :
710  LDA byte
720 .loop2 CMP (loc),Y
730  BNE error
740  INY
750  BNE loop2
760 :
770  INC loc+1
780  DEX \ Decrement page counter
790  BNE loop2 \ All pages done ?
800 :
```

```
810\ End of one pass
820 LDA #ASC "*"
830 JSR oswrch
840 INC byte
850 BNE begin \ Unless byte = 0
860:
870 LDA #7 \ Beep
880 JSR oswrch
890 JMP begin
900:
910.error STY loc \ Error address now
    in loc, loc+1
920 LDA #7 \ Bell and exit
930 JMP oswrch
940]
950NEXT
960:
970END
```

REM STRIPPER

This is one of the three 'squeeze' utilities which will help to shorten a BASIC program and so make it run faster. It removes **REM** statements and, optionally, lines containing only colons and spaces (used, for example, to highlight the boundaries of block structures).

There are two versions of the program on the tape. REMSTRP is the version for use with tape-based computers and resides between &E00 and &10FF. RMSTRPDISK is for use with disks and is loaded between &1600 and &18FF. Please refer to 'Using the Programming Utilities' for installation instructions. Other than the addresses at which they start, the two programs operate identically.

As the utility is co-resident, you can load it before or after you get your BASIC program into memory. Once the utility is in memory, start it working with **CALL &E00** (tape version) or **CALL &1600** (disk version). On disk-based systems, it is the first of the utilities used in the ***EXEC SQUEEZE** routine.

The routine begins by setting Mode 7 and displaying the message
Do you want to remove single-colon and single-space lines ?
If you type **Y**, all such lines will be removed. Otherwise, type **N**. If, instead, you want to stop the routine at this point, press Escape.

The next message to be displayed is **Stripping in progress** . . . After a short while - just how long depends on the length of your program and how much stripping needs to be done - the prompt will return, leaving the stripped BASIC program in memory.

If you are squeezing a program to its smallest size, REMSTRP should be the first utility that you use, before CRUNCH and PACKER.

In case your program contains a reference to a line which is actually a **REM** statement line, such as:

```
100 GOSUB 1000


1000 REM Print-out subroutine
1010 :
1020 PRINT . . .
```

REMSTRP will automatically adjust the line reference to the next 'live' line. In this case, line 100 will be changed to **GOSUB 1010** - if you have not opted to remove single-colon lines - or to **GOSUB 1020** if you have.

If you want to keep one or two **REM**s (the program title for instance), you can either re-enter them after using REMSTRP or you can temporarily alter the **REM** to, say, **RME** so that it will not be recognised by the utility.

REMSTRP will not remove comments from an assembly language listing (i.e. text preceded by a reverse oblique).

Please refer to 'Using the Programming Utilities' for notes on tacked-on bytes, embedded control codes and other general hints.

```
  10 REM ×××× REM STRIPPER ××××
  20 :
  30 REM (c) Ian Trackman 1982
  40 :
  50 REM Re-referencing fails if
     reference to REM etc. as last line
     of program
  60 REM Assumes no hidden &8Ds embedded
     in text
  70 :
  80 DIM msg(2)
  90 :
 100 REM Basic pointers
 110 lomem   = &0
 120 himem   = &6
 130 vartop  = &2
 140 top     = &12
 150 page    = &18
 160 :
 170 memloc  = &70 : REM + &71
 180 linenum = &72 : REM + &73
 190 srchloc = &74 : REM + &75
 200 binnum  = &76 : REM + &77, &78
 210 temp    = &79 : REM + &7A, &7B
 220 source  = &7C : REM + &7D
 230 destin  = &7E : REM + &7F
 240 colonflag = &80
 250 flag    = &81
 260 length  = &82
 270 srchlen = &83
```

```
280 offset   = &84
290 ysave    = &85
300 count    = &86
310 :
320 osrdch = &FFE0
330 oswrch = &FFEE
340 osnewl = &FFE7
350 :
360 REM Constants
370 eol    = &0D
380 esc    = &1B
390 space = ASC " "
400 colon = ASC ":"
410 rem    = &F4 : REM Basic token
420 maxsize = &7C : REM for Mode 7
430 :
440 org = &E00
450 :
460 opt = 2
470 :
480 FOR I% = 0 TO opt STEP opt
490 P% = org
500 [
510 OPT I%
520 :
530  LDY #0
540 .msg1loop LDA msg(1),Y \ Mode 7
     and title
550  BEQ msg1done
560  JSR oswrch
570  INY
580  BNE msg1loop
590 :
600 \ Himem under Mode 7
610 .msg1done LDA #maxsize
620  STA himem+1
630  LDX #0
640  STX himem
650 :
```

```
 660 \ Strip single colons ?
 670 .ask JSR osrdch
 680  AND #&DF \ Mask to upper-case
 690  CMP #ASC"N"
 700  BEQ no
 710  CMP #ASC"Y"
 720  BEQ yes
 730  CMP #esc
 740  BNE ask
 750  JMP finish
 760 :
 770 .yes DEX \ To &FF
 780 .no STX colonflag
 790  JSR oswrch \ Print Y or N
 800 :
 810  LDY #0
 820 .msg2loop LDA msg(2),Y \
      Stripping
 830  BEQ msg2done
 840  JSR oswrch
 850  INY
 860  BNE msg2loop
 870 :
 880 \ Adjust cross-references
 890 .msg2done LDA #1 \ Start at PAGE +
      1
 900  STA memloc
 910  LDA page
 920  STA memloc+1
 930 :
 940 \ Find an opening REM etc.
 950 .nextline LDY #0
 960  LDA (memloc),Y
 970  CMP #&FF \ End of program flag
 980  BNE morelines
 990  JMP strip \ Re-referencing
      completed
1000 :
```

```
1010 .morelines INY
1020  INY
1030  LDA (memloc),Y
1040  STA length \ Offset to start of
      next line
1050 :
1060 .nextbyte INY
1070  LDA (memloc),Y
1080  CMP #rem
1090  BEQ refchek
1100 :
1110  CMP #eol
1120  BEQ flagtest
1130 :
1140  CMP #colon
1150  BEQ nextbyte
1160 :
1170  CMP #space
1180  BEQ nextbyte
1190 .notcolon JMP endline
1200 :
1210 .flagtest BIT colonflag
1220  BPL notcolon
1230 :
1240 \ Is this line referred to ?
1250 .refchek LDY #0
1260  LDA (memloc),Y
1270  STA linenum+1
1280  INY
1290  LDA (memloc),Y
1300  STA linenum
1310  JSR convert \ Convert linenum to
      3-byte code
1320 :
1330 \ 'temp' will be re-used, so -
1340  LDX #2
1350 .transfer LDA temp,X
1360  STA binnum,X
1370  DEX
1380  BPL transfer
```

```
1390 :
1400 \ Look for that number in program
1410  LDA #1
1420  STA srchloc
1430  LDA page
1440  STA srchloc+1
1450 :
1460 .nextsrch LDY #0
1470  LDA (srchloc),Y
1480  CMP #&FF
1490  BEQ endline \ End of this pass
1500 :
1510  INY
1520  INY
1530  LDA (srchloc),Y
1540  STA srchlen
1550 :
1560 .moresrch INY
1570  LDA (srchloc),Y
1580  CMP #eol
1590  BEQ srchline
1600 :
1610  CMP #&8D \ Line number token
1620  BNE moresrch
1630 :
1640 \ Compare with 3 coded bytes
1650  LDX #0
1660  STX flag
1670 :
1680 .trymatch INY
1690  LDA (srchloc),Y
1700  CMP binnum,X
1710  BEQ morematch
1720  DEC flag
1730 :
1740 .morematch INX
1750  CPX #3
1760  BNE trymatch
1770 :
```

```
1780  LDA flag
1790  BNE moresrch \ No match if < 0
1800 :
1810 \ Alter this reference to coded
       next line
1820 \ Code up next line number
1830  LDA memloc
1840  CLC
1850  ADC length
1860  STA source
1870  LDA memloc+1
1880  ADC #0
1890  STA source+1
1900  STY ysave
1910  LDY #0
1920  LDA (source),Y
1930  STA linenum+1
1940  INY
1950  LDA (source),Y
1960  STA linenum
1970  JSR convert
1980  LDY ysave
1990 :
2000 \ Alter the &8D reference
2010  LDX #2
2020 .swap LDA temp,X
2030  STA (srchloc),Y
2040  DEY
2050  DEX
2060  BPL swap
2070 :
2080  LDY ysave \ Where we were looking
2090  BNE moresrch \ Always
2100 :
2110 .srchline LDA srchloc
2120  CLC
2130  ADC srchlen
2140  STA srchloc
2150  LDA srchloc+1
2160  ADC #0
```

```
2170  STA srchloc+1
2180  BNE nextsrch \ Always
2190 :
2200 \ Set up for next line
2210 .endline LDA memloc
2220  CLC
2230  ADC length
2240  STA memloc
2250  LDA memloc+1
2260  ADC #0
2270  STA memloc+1
2280  JMP nextline
2290 :
2300 :
2310 \ Convert linenum to 3-byte code
2320 .convert LDA linenum+1
2330  ORA #&40
2340  STA temp+2
2350  LDA linenum
2360  AND #&3F
2370  ORA #&40
2380  STA temp+1
2390  LDA linenum
2400  AND #&C0
2410  STA linenum
2420  LDA linenum+1
2430  AND #&C0
2440  LSR A
2450  LSR A
2460  ORA linenum
2470  LSR A
2480  LSR A
2490  EOR #&54
2500  STA temp
2510  RTS
2520 :
2530 :
2540 \ Actual stripping starts here
2550 :
```

```
2560 .strip LDA #1
2570  STA memloc
2580  LDA page
2590  STA memloc+1
2600 :
2610 .lookrem LDY #0
2620  LDA (memloc),Y
2630  CMP #&FF
2640  BNE lookrem2
2650  JMP finish
2660 :
2670 .lookrem2 INY \ Skip line-number
      byte 2
2680  INY
2690  LDA (memloc),Y
2700  STA length
2710  INY
2720  LDA (memloc),Y
2730  CMP #rem \ Opening REM ?
2740  BEQ lineout
2750 :
2760  BIT colonflag \ Leave colons ?
2770  BPL midlook
2780 :
2790  CMP #colon
2800  BEQ opener
2810 :
2820  CMP #space
2830  BNE midlook
2840 :
2850 \ Test for one or more opening
       colons or spaces
2860 .opener INY
2870  LDA (memloc),Y
2880  CMP #space
2890  BEQ opener \ Still maybe
2900 :
2910  CMP #colon
2920  BEQ opener \ Still maybe
2930 :
```

```
2940  CMP #eol \ A whole line of them ?
2950  BNE midlook2
2960 :
2970 \ Remove entire line
2980 .lineout LDY #0 \ Nothing to keep
2990  JSR pack
3000  JMP lookrem \ Next line now in
      position of lost line
3010 :
3020 .midlook INY
3030 .midlook2 LDA (memloc),Y
3040  CMP #eol
3050  BEQ lookline \ Onto next line
3060 :
3070  CMP #rem
3080  BNE midlook
3090 :
3100 \ Test for preceding colons or
       spaces
3110 .back DEY
3120  CPY #2 \ At start of line ?
3130  BEQ lineout
3140 :
3150  LDA (memloc),Y
3160  CMP #space
3170  BEQ back
3180 :
3190  CMP #colon
3200  BEQ back
3210 :
3220  INY \ Cancel last INY
3230  DEC length \ Overlay &D before
      following line
3240  JSR pack \ Found mid-line REM
3250  LDY ysave \ Saved by 'pack' but
      points at eol, so -
3260  INY
3270  TYA
3280  STA length
3290  LDY #2
```

```
3300  STA (memloc),Y \ Over-write
      current length
3310 :
3320 .lookline LDA memloc
3330  CLC
3340  ADC length
3350  STA memloc
3360  LDA memloc+1
3370  ADC #0
3380  STA memloc+1
3390  JMP lookrem
3400 :
3410 :
3420 \ Packing sub-routine
3430 \ Enter with (memloc),Y on split
       point
3440 .pack STY ysave \ Set up move
      pointers
3450  LDA memloc
3460  CLC
3470  ADC ysave
3480  STA destin
3490  LDA memloc+1
3500  ADC #0
3510  STA destin+1
3520 :
3530  LDA memloc
3540  CLC
3550  ADC length
3560  STA source
3570  LDA memloc+1
3580  ADC #0
3590  STA source+1
3600 :
3610  LDA top
3620  SEC
3630  SBC source
3640  STA count
3650  LDA top+1
3660  SEC
```

```
3670  SBC source+1
3680  LDY #0
3690  TAX \ Pages to move
3700  BEQ shift2
3710 :
3720 .shift1 LDA (source),Y
3730  STA (destin),Y
3740  INY
3750  BNE shift1
3760  INC source+1
3770  INC destin+1
3780  DEX
3790  BNE shift1
3800 :
3810 .shift2 LDX count \ Move odd
      bytes
3820  BEQ shiftdone
3830 :
3840 .shift3 LDA (source),Y
3850  STA (destin),Y
3860  INY
3870  DEX
3880  BNE shift3
3890 :
3900 \ Reset Basic pointers
3910 .shiftdone LDA length
3920  SEC
3930  SBC ysave
3940  STA offset
3950  LDA top
3960  SEC
3970  SBC offset
3980  STA top
3990  STA lomem
4000  STA vartop
4010  LDA top+1
4020  SBC #0
4030  STA top+1
4040  STA lomem+1
4050  STA vartop+1
```

```
4060  RTS
4070 :
4080 .finish JSR osnewl
4090  JMP osnewl \ & exit to caller
4100 ]
4110 :
4120 PROCtext (1, CHR$22 + CHR$7 +
     CHR$31 + CHR$12 + CHR$2 + "REM
     STRIPPER" + CHR$31 + CHR$1 + CHR$8
     + "Do you want to remove
     single-colon and" + CHR$13 + CHR$10
     + CHR$10 + " single-space lines ?
     ")
4130 PROCtext (2, CHR$31 + CHR$7 +
     CHR$16 + "Stripping in progress
     .... ")
4140 :
4150 NEXT
4160 :
4170 END
4180 :
4190 :
4200 DEF PROCtext (N,A$)
4210 msg(N) = P%
4220 $msg(N) = A$
4230 P% = P% + LEN(A$) + 1
4240 P%?-1 = 0
4250 ENDPROC
```

GLOBAL REPLACEMENT

REPLACE is a utility written in machine-code, which will change almost anything - commands, variables, strings or text - into anything else within a BASIC program, irrespective of their relative sizes. You can use it, for example, to change long variable names to short ones or vice versa, to change real arrays to integer arrays or subroutine calls into calls to named procedures.

There are two versions of the program on the tape. REPLACE is the version for use with tape-based computers and resides between &E00 and &FFF. REPLACEDSK is for use with disks and is loaded between &1700 and &18FF. Please refer to 'Using the Programming Utilities' for installation instructions. Other than their starting addresses, the two programs operate identically.

The utility is co-resident, that is, it will remain in the computer's memory whilst you load, run and save BASIC programs until you over-write it.

Before you use the utility, you'll need to set up two function keys - any keys will do. Program the first key
***KEY 1 0 |H Replace ?**
and the second key
***KEY 2CA.&E00|K|M**
If you like, you can add a space after the question mark at the end of the first key's string. The second key's string must be typed exactly as printed above, with *no* spaces between the characters. If you want to use keys other than 1 and 2, you'll obviously use different key numbers when setting them up. If you are using the disk version of the program, the second key string will be
*** KEY 2CA.&1700|K|M**

To use the utility, press the first function key and the message **Replace ?** will appear on the screen. Type in what you want to replace (let's call it the 'target' from now on) followed by a reverse oblique('\'), then whatever you want to substitute for the target (we'll call it the 'replacement'), ending with a Return. If you are in Mode 7, the oblique will appear on the screen as ½. The oblique acts as a 'delimiter', that is, it tells REPLACE where your target ends and your replacement begins.

As an example, let us say that your program contains an integer array **A%()**. You now realise that it has to hold decimal numbers and you have to alter every reference to it into **A()**. After pressing the first function key (**Replace ?** appears), you would type:
A%(\A(
followed by a Return.

What you have done at this point is to cause a new line, numbered 0, to be added to the program in memory, containing the target and the replacement separated by a delimiting oblique. (If you wonder why we add a new line to your BASIC program, the reason is so that we can use the parser in ROM to create tokens from BASIC keywords.)

There are a number of points to bear in mind from this process.

Since REPLACE temporarily adds a new line 0 to your BASIC program (which it subsequently deletes), your program must not already contain a line 0. If it does, it will be lost.

Be careful if your target might appear in different contexts in the program. For example, if you want to change the variable **A** to **B**, and you give just the letter **A** as your target, every occurrence of that character in your program will be altered. To prevent unwanted changes, you could temporarily change other occurrences to unused names (e.g. **A$** into **ZZZ$**). Then alter **A** to **B** and finally reset **ZZZ$** to **A$**.

If you do not proceed to the second stage of the utility (by pressing the second function key), you will be left with an unwanted line 0, which will probably cause a syntax error unless you delete it before running the program.

Since the reverse oblique is reserved for use as a delimiter, you can't include it as part of your target or replacement.

The utility will search for the target in the exact form in which you have typed it and will change it to the replacement, again exactly as you have typed it. Therefore, be accurate - particularly with spaces.

One advantage of using the parser is that you can type in the truncated form of BASIC keywords. If you are changing, say, **MOVE XPOS** to **DRAW XPOS**, you can type:
MOV. XPOS\DR. XPOS

The drawback of using the parser is that it will not tokenise anything after a **REM** or **DATA** command. You cannot therefore specify **REM** or **DATA** as part of your target, because the second **REM** or **DATA** will be retained as ASCII characters by the parser. Although it will look correct when you list the program line, it will generate a syntax error when the program is run. If this happens, re-copy the line with the cursor edit keys and the line will be re-parsed and re-inserted into your program in its correct form. The same principle explains why you should be careful about including both opening and closing quotation marks in a target. For instance,

PRINT "Hello\PRINT"Goodbye
will not work, whereas you will succeed with
PRINT "Hello"\PRINT "Goodbye"

You can mix keywords, variables and text as you wish. For example, you can change
A$ = "APPLES"
to
PRINT TAB(13) "ORANGES"

Because your target and replacement are tokenised, you must include the full word (don't type **SUB 1000** instead of **GOSUB 1000**) and all necessary brackets in accordance with the list of tokens on pages 483–4 of the User Guide. For instance, if you want to replace **LEFT$** with **RIGHT$**, you must type:
LEFT$(RIGHT$(

The final point is that you must not begin your target with a number, since it will be parsed as part of the line-number and you will add a new and unwanted line somewhere else in the program! Since numbers within a line can almost always be related to another command, e.g. **GOSUB**, or a mathematical symbol, include that as the start of the target and replacement.

Having entered your target and replacement, press the second function key.

If you have made an error in the form of your command line, you'll get the message **Bad command.** This will happen if there is no target before the oblique, no replacement after it, no oblique or more than one oblique. Re-enter the command line by pressing the first function key.

If the utility cannot find your target anywhere in the BASIC program, it will respond with **Not found**. Otherwise it will replace the target with the replacement and indicate, with the message **Replaced**, that it has finished.

There are two situations when replacement will not take place. The maximum length of a BASIC program line is 237 characters. If you try to change a target in a line with a longer replacement and the effect would be to exceed the maximum line length, replacement will stop with the message **Too long.** Since some replacement may already have taken place, you should now list the program to find the problem line and break it up into shorter parts. Remember to re-use the utility, since otherwise you will have a partially altered program.

The other case when replacement will stop is if, as before, you are enlarging the program by making the replacement larger than the target. If the size of the program grows until **TOP** reaches **HIMEM** and it runs out of expansion space, you'll again get a **Too long** error. One solution is to change to Mode 7 and repeat the command, unless you're already in Mode 7, in which case your program is just too big!

As with all of these types of utility, 'tacked-on' bytes will cause problems. (See 'Using the Programming Utilities'.)

The utility contains useful routines for string searching and for moving blocks of bytes upwards and downwards through the memory.

```
   10 REM **** GLOBAL REPLACEMENT ****
   20 :
   30 REM (c) Ian Trackman 1983
   40 :
   50 DIM msg(4),OSCL 20
   60 find$ = "Replace ? "
   70 PROCoscli ("KEY 1 0|H" + find$) :
      REM Put user input into line 0
   80 :
   90 himem = &06 : REM &07
  100 top    = &12 : REM &13 Top of
      program
  110 page   = &18 : REM Holds msb of
      start of program
  120 :
  130 loc      = &70 : REM + &71
  140 from     = &72 : REM + &73
  150 dest     = &74 : REM + &75
  160 gap      = &76 : REM + &77
  170 length   = &80 : REM ... of line
  180 status   = &81 : REM Exit message
      index
  190 size     = &82 : REM Relative size
      flag
  200 len_one = &83 : REM Size of target
  210 len_two = &84 : REM Size of
      replacement
  220 offset   = &85 : REM Difference in
      size
```

```
230 ysave   = &86
240 ysave2  = &87
250 :
260 REM User's data buffers
270 first   = &710 : REM After start of
    keyboard buffer
280 second  = &780
290 :
300 oswrch = &FFEE
310 osbyte = &FFF4
320 :
330 eol   = &0D
340 slash = ASC "\"
350 :
360 org = &E00
370 PROCoscli ("KEY 2CA.&" + STR$~org +
    "|K|M")
380 :
390 opt = 2
400 :
410 FOR I% = 0 TO opt STEP opt
420 P% = org
430 [
440 OPT I%
450 :
460 \ Find user's input in line 0
470 :
480  LDA #3 + LEN find$ \ Skip prompt
490  STA loc
500  LDA page
510  STA loc+1
520 :
530  LDY #&FF
540 .loop1 INY
550  LDA (loc),Y
560  CMP #eol
570  BEQ errjmp \ Before slash ?
580  CMP #slash
590  BNE loop1
600 :
```

```
610  STY len_one
620  TYA \ For Z-flag
630  BEQ errjmp \ Null string
640  BMI errjmp \ Over &7F characters
650  DEC len_one
660  DEC len_one
670 :
680  LDX #&FE \ For slash and eol
690 .loop2 INX
700  INY
710  LDA (loc),Y
720  CMP #eol
730  BNE loop2
740 :
750  CPX #&FF
760  BEQ errjmp \ Nothing after "\"
770  CPX #&7F
780  BCS errjmp \ Too big
790  STX len_two
800 :
810 \ Reverse makes faster search
820  LDX #0
830  DEY
840 .rev1 LDA (loc),Y
850  CMP #slash
860  BEQ rev2
870  STA second,X
880  INX
890  DEY
900  BNE rev1
910 :
920 .rev2 LDX #0
930  DEY \ skip slash
940 .rev3 LDA (loc),Y
950  STA first,X
960  INX
970  DEY
980  BNE rev3
990 :
```

```
1000 \ Difference in size ?
1010  LDX #0 \ For size flag
1020  LDA len_one
1030  SEC
1040  SBC len_two
1050  BEQ search \ Size remains 0
1060 :
1070  LDX #&40 \ Bit 6 set
1080  BCS search
1090 :
1100  LDA len_two
1110  SEC
1120  SBC len_one
1130  LDX #&80 \ Bit 7 set
1140  BNE search \ always
1150 :
1160 :
1170 .errjmp LDA #msg(1) \ "Bad
      command"
1180  STA status
1190  JMP finish
1200 :
1210 :
1220 \ Start of search
1230 :
1240 .search STA offset
1250  STX size
1260  LDA #msg(2) \ "Not found"
1270  STA status
1280 :
1290  LDY #1 \ Start of program
1300  STY loc
1310 :
1320 \ Skip fake line 0
1330  INY
1340  LDA (loc),Y
1350  STA length
1360  BNE endline \ Always
1370 :
```

```
1380 .nextline LDY #0
1390  LDA (loc),Y
1400  BMI finish \ &FF end of program
1410 :
1420  INY
1430  INY
1440  LDA (loc),Y
1450  STA length \ Offset to start of
      next line
1460 :
1470 .lookmore LDX len_one
1480  STY ysave \ For moves
1490 .nextbyte INY
1500  LDA (loc),Y
1510  CMP #eol
1520  BEQ endline
1530 :
1540 .compare CMP first,X
1550  BNE lookmore
1560  DEX
1570  BPL nextbyte
1580 :
1590 :
1600 \ Match found
1610  STY ysave2 \ End of match
1620  BIT size
1630  BMI larger
1640  BVC swap \ Same size
1650 :
1660  JSR movedown
1670  JMP swap
1680 :
1690 .larger JSR moveup
1700  BCS finish \ Error in move
1710 :
1720 \ Insert replacement
1730 :
1740 .swap LDX len_two
1750  LDY ysave
1760  INY
```

```
1770 .swap2 LDA second,X
1780  STA (loc),Y
1790  INY
1800  DEX
1810  BPL swap2
1820 :
1830  LDA #msg(3) \ "Completed"
1840  STA status
1850  LDY ysave
1860  BIT size
1870  BPL lookmore \ Same size or less
1880 :
1890 \ Skip re-search of replacement
1900  LDA ysave
1910  SEC \ Plus 1
1920  ADC offset
1930  TAY
1940  BNE lookmore \ Always
1950 :
1960 .endline LDA loc
1970  CLC
1980  ADC length
1990  STA loc
2000  BCC nextline
2010  INC loc+1
2020  BNE nextline \ Always
2030 :
2040 :
2050 .finish LDX status
2060 .msgloop LDA msgstart,X
2070  BEQ exit
2080  JSR oswrch
2090  INX
2100  BNE msgloop
2110 :
2120 \ Delete line 0
2130 .exit LDA #&15 \ VDU off
2140  JSR oswrch
2150  LDY #ASC "0"
2160  JSR bufchar
```

```
2170  LDY #6 \ VDU on
2180  JSR bufchar
2190  LDY #eol \ Fall through & exit to Basic
2200 :
2210 \ Puts Y into keyboard buffer
2220 .bufchar LDA #&8A
2230  LDX #0
2240  JMP osbyte
2250 :
2260 :
2270 \ **** Subroutines ****
2280 :
2290 \ **** Move memory block right
2300 :
2310 .moveup LDA length
2320  CLC
2330  ADC offset
2340  CMP #&ED
2350  BCS too_long \ Over 237 characters
2360 :
2370 \ Store new length
2380  STA length
2390  LDY #2
2400  STA (loc),Y
2410 :
2420  LDA top
2430  STA from
2440  CLC
2450  ADC offset
2460  STA dest
2470  TAX \ Temp save
2480  LDA top+1
2490  STA from+1
2500  ADC #0
2510  STA dest+1
2520  CMP himem+1
2530  BCS too_long
2540 :
```

```
2550  STX top \ OK to alter
2560  STA top+1
2570 :
2580  LDA from
2590  SEC
2600  SBC loc
2610  STA gap
2620  LDA from+1
2630  SBC loc+1
2640  STA gap+1
2650 :
2660  LDA gap
2670  SEC
2680  SBC ysave2
2690  STA gap
2700  BCS rmove
2710  DEC gap+1
2720 :
2730 .rmove LDY #0
2740  LDX gap+1
2750  BEQ rmvpart
2760 :
2770 .rmv2 DEC from+1
2780  DEC dest+1
2790 .rmv3 DEY
2800  LDA (from),Y
2810  STA (dest),Y
2820  CPY #0
2830  BNE rmv3
2840 :
2850  DEX
2860  BNE rmv2
2870 :
2880 .rmvpart LDX gap
2890  BEQ rmvdone
2900 :
2910  DEC from+1
2920  DEC dest+1
2930 .rmv4 DEY
2940  LDA (from),Y
```

```
2950  STA (dest),Y
2960  DEX
2970  BNE rmv4
2980 :
2990 .rmvdone CLC
3000  RTS
3010 :
3020 .too_long LDA #msg(4)
3030  STA status
3040  RTS \ With carry set
3050 :
3060 :
3070 \ **** Move memory block left
3080 :
3090 .movedown LDA loc
3100  CLC
3110  ADC ysave2
3120  STA from
3130  LDA loc+1
3140  ADC #0
3150  STA from+1
3160 :
3170  LDA from
3180  SEC
3190  SBC offset
3200  STA dest
3210  LDA from+1
3220  SBC #0
3230  STA dest+1
3240 :
3250  LDA top
3260  SEC
3270  SBC from
3280  STA gap
3290  LDA top+1
3300  SBC from+1
3310  STA gap+1
3320 :
3330  LDA top
3340  SEC
```

```
3350  SBC offset
3360  STA top
3370  LDA top+1
3380  SBC #0
3390  STA top+1
3400 :
3410 .lmove LDY #0
3420  LDX gap+1
3430  BEQ lmvpart
3440 :
3450 .lmvpage LDA (from),Y
3460  STA (dest),Y
3470  INY
3480  BNE lmvpage
3490  INC from+1
3500  INC dest+1
3510  DEX
3520  BNE lmvpage
3530 :
3540 .lmvpart LDX gap
3550  BEQ lmvdone
3560 :
3570 .lmvlast LDA (from),Y
3580  STA (dest),Y
3590  INY
3600  DEX
3610  BNE lmvlast
3620 :
3630 .lmvdone LDA length
3640  SEC
3650  SBC offset
3660  STA length
3670  LDY #2
3680  STA (loc),Y
3690  RTS
3700 :
3710 .msgstart \ Used for messages
3720 ]
3730 :
```

```
3740 REM Cover "CA.&aaaa"
3750 PROCtext (1,CHR$7 + "Bad command")
3760 PROCtext (2,"Not found")
3770 PROCtext (3,"Replaced ")
3780 PROCtext (4,CHR$7 + "Too long ")
3790 NEXT
3800 :
3810 END
3820 :
3830 :
3840 DEF PROCoscli (A$)
3850 X% = OSCL MOD &100
3860 Y% = OSCL DIV &100
3870 $OSCL = A$
3880 CALL &FFF7
3890 ENDPROC
3900 :
3910 :
3920 DEF PROCtext (N,A$)
3930 msgaddr = P%
3940 $msgaddr = A$
3950 msg(N) = P% - msgstart : REM Offset
     to message
3960 P% = P% + LEN(A$) + 1
3970 P%?-1 = 0
3980 ENDPROC
```

PROGRAM RESEQUENCER

RESEQ is a utility written in machine-code, which moves one or more lines of a BASIC program to another place within the program. Having written part of a program, you might decide that you need to re-use a section of code. Instead of copying it out again, you can move it down the program, turn it into a procedure and call on it as needed. Even if your program uses a block of the code only once, it often helps to establish the structure of a program by creating a number of smaller, self-contained procedures out of a large section of code. You might also want to use RESEQ to sort all of your procedures into alphabetical order so as to make them easier to find in a listing.

There are two versions of the program on the tape. RESEQ is the version for use with tape-based computers and resides between &E00 and &10FF. RESEQDISK is for use with disks and is loaded between &1600 and &18FF. Please refer to 'Using the Programming Utilities' for installation instructions. Other than the addresses at which they start, the two programs operate identically.

As the utility is co-resident, you can load it before or after you get your BASIC program into memory. Once the utility is in memory, start it working with **CALL &E00** (tape version) or **CALL &1600** (disk version). If you are going to make repeated use of the utility, it might be worth setting up a function key to make the call.

When you invoke the routine, it will respond with a hash sign ('#') to indicate that it is waiting for your command line.

The form of the command line is:
FLN/LLN:TLN
FLN is the line-number of the first line of the block to be moved. LLN is the line-number of the last line of the block to be moved. TLN is the 'target' line-number which tells RESEQ where to move the block. Don't include any spaces.

As an example, if you want to move lines 550 to 740 to a point just after line 1250, you would type:
550/740:1251
followed, of course, by Return.

Since the maximum number of digits in FLN, LLN and TLN is five, you will be allowed to enter up to 17 characters. Anything other than a numeral, an oblique or a colon will be ignored.

FLN and LLN are interpreted in the same way as line-numbers that you would give to the **LIST** command. Assuming that your program is numbered in normal incremental steps of 10, you could have given any number between 541 and 550 as FLN and any number between 740 and 749 as LLN. In other words, RESEQ uses the closest actual line-number.

As to TLN, you can give any number up to and including the line-number of the line before which the insertion is to be made. In other words, in the above example you could have given any number between 1251 and 1260. However, we suggest that to avoid mistakes, you forget about the ability to use an existing line-number (i.e. 1260).

The utility will also accept default values. Using our symbols to represent real numbers, if you type:
/LLN:TLN
RESEQ will assume that FLN is the first line of the program. If you type:
FLN/:TLN
it will assume that LLN is the last line of the program; and if you type:
FLN/LLN:
it will assume that you want to move the defined block to the end of the program. You can use the default values of FLN and TLN in combination, as in:
/50:
which means that you want to move the lines from the start of the program up to line 50 to the end of the program.

RESEQ will first test the validity of your command line. It will reject it:
- if the command line does not contain one and only one oblique
- if the command line does not contain one and only one colon
- if the colon precedes the oblique
- if FLN, LLN or TLN are not integer decimal numbers in the range 0 to 32767
- if FLN is greater than LLN
- if LLN is less than FLN
- if TLN is between FLN and LLN.

Notice that there is nothing to prevent FLN and LLN from being the same number. Indeed, that is how you would tell RESEQ to move a single line.

Remember that if your command line is rejected, you must re-invoke the utility and produce a new hash symbol. Don't just copy over the old command line, as you'll either create a syntax error or, worse, accidentally over-write the original line at FLN if you miss out the hash!

Because the block of code has to be moved around the computer's memory, additional RAM space is needed. If there is insufficient room, RESEQ will fail with a **No room** error. If you are in a high-resolution Mode, try switching to Mode 7.

Once your command line is accepted, RESEQ will take a fraction of a second to make the move. It then has to change the line-numbers. Rather than making the utility even larger by including our own renumberer, we decided to make RESEQ call up BASIC's own RENUMBER command and so you'll see **REN.** appear on the screen just before control is handed back to you.

Renumbering can cause problems if the block being moved contains a line which is referred to elsewhere in the program. You'll know about it when you get the **Failed at** . . . error message. If you write well-structured programs with named procedures, this should cause only minor difficulties with **RESTORE** and **ON ERROR** commands containing line references. You have to edit those lines by hand after the renumber in order to reinstate the correct line references. However, if you write unstructured, 'spaghetti' code full of **GOTO**s, we regret that we have little sympathy for you.

Referring back to the opening paragraph, we would make one suggestion as to the creation of new procedures out of main-line code. Enter the **DEF** and **ENDPROC** lines and the **PROC** call itself *before* you invoke RESEQ. In that way, your program will be correct immediately after using the utility. If you insert the procedure commands after moving the block of code, there is a danger that the renumbering will confuse you as to the correct place in the program to add the new procedure call.

The utility contains some useful subroutines for text input verification, block memory moves and decimal to hexadecimal number conversion.

Please refer to 'Using the Programming Utilities' for notes on tacked-on bytes and other general hints.

```
10 REM **** RESEQUENCER ****
20 :
30 REM (c) Ian Trackman 1983
40 :
50 *KEY 4 CALL &1600|M
60 :
70 himem = &06 : REM + &7
80 page = &18 : REM Hi-byte of PAGE
90 :
```

```
100 loc     = &70 : REM + &71
110 final   = &72 : REM + &73
120 line1   = &74 : REM + &75
130 line2   = &76 : REM + &77
140 target  = &78 : REM + &79
150 top     = &7A : REM + &7B
160 addr1   = &7C : REM + &7D
170 addr2   = &7E : REM + &7F
180 tgtadr  = &80 : REM + &81
190 temp    = &82 : REM + &83
200 dest    = &84 : REM + &85
210 size    = &86 : REM + &87
220 gap     = &88 : REM + &89
230 mark    = &8A
240 flag    = &8B
250 hi_tmp  = &8C
260 :
270 param   = &600 : REM 5 bytes
280 buffer  = &605 : REM User-defined
    keyboard input buffer
290 :
300 osnewl  = &FFE7
310 oswrch  = &FFEE
320 osword  = &FFF1
330 osbyte  = &FFF4
340 :
350 return  = 13
360 slash = ASC "/"
370 colon = ASC ":"
380 :
390 org = &E00
400 :
410 opt = 2
420 :
430 FOR I% = 0 TO opt STEP opt
440 P% = org
450 [
460 OPT I%
470 :
480 :
```

```
490 \ **** Input a line
500 :
510 \ Set up parameter block
520  LDA #buffer MOD &100
530  STA param
540  LDA #buffer DIV &100
550  STA param+1
560  LDA #17 \ Maximum line length
570  STA param+2
580  LDA #slash \ Minimum ASCII
590  STA param+3
600  LDA #colon \ Maximum ASCII
610  STA param+4
620 :
630  LDA #ASC "#" \ As prompt
640  JSR oswrch
650  LDA #15 \ Flush input buffer
660  LDX #1
670  JSR osbyte
680 :
690 \ Call OS input routine
700  LDA #0
710  LDX #param MOD &100
720  LDY #param DIV &100
730  JSR osword
740  BCC not_esc \ Carry set if escape
750  JMP osnewl \ so RTS to BASIC
760 :
770 :
780 \ **** Get first line number
790 :
800 .not_esc LDY #1 \ Skip 'return'
810  STY loc
820  LDA page
830  STA loc+1
840  DEY \ To 0
850  LDA (loc),Y
860  STA line1+1 \ Hi-byte of line no.
870  INY
880  LDA (loc),Y
```

```
 890  STA line1 \ Lo-byte of line no.
 900  INY
 910 :
 920 :
 930 \ **** Get last line number
 940 :
 950 .getlast LDA (loc),Y \ Line
      length
 960  CLC
 970  ADC loc
 980  STA loc
 990  BCC getlst2
1000  INC loc+1
1010 :
1020 .getlst2 LDY #0
1030  LDA (loc),Y
1040  BMI save_top \ &FF = end of
      program
1050 :
1060 \ Save latest line no.
1070  STA final+1 \ Hi-byte of last line
      no.
1080  INY
1090  LDA (loc),Y
1100  STA final \ Lo-byte of last line
      no.
1110  INY
1120  BNE getlast \ Always
1130 :
1140 \ Save top of program address
1150 .save_top LDA loc
1160  STA top
1170  LDA loc+1
1180  STA top+1
1190 :
1200 :
1210 \ **** Get first command line
       no.
1220 :
```

```
1230  LDA #buffer MOD &100 \ Set up for
      'convert'
1240  STA loc
1250  LDA #buffer DIV &100
1260  STA loc+1
1270 :
1280  LDA line1 \ Parameters
1290  LDX line1+1
1300  LDY #slash
1310  JSR convert
1320  BCS jmperr
1330 :
1340 \ Is new start line after real
       first line ?
1350  LDA temp
1360  CMP line1
1370  LDA temp+1
1380  SBC line1+1
1390  BCC hexscnd
1400 :
1410 \ If so, use new start line
       number
1420  LDA temp
1430  STA line1
1440  LDA temp+1
1450  STA line1+1
1460 :
1470 :
1480 \ **** Convert second number to
       hex
1490 :
1500 .hexscnd LDA final
1510  STA line2 \ Can never be > final
1520  LDX final+1
1530  STX line2+1
1540  LDY #colon
1550  JSR convert
1560  BCS jmperr
1570 :
```

```
1580 \ Is new start line before real
      last line ?
1590  LDA temp
1600  CMP final
1610  LDA temp+1
1620  SBC final+1
1630  BCS order
1640 :
1650 \ If so, use new end line number
1660  LDA temp
1670  STA line2
1680  LDA temp+1
1690  STA line2+1
1700 :
1710 :
1720 \ **** Is 'last' before 'first'
      ?
1730 :
1740 .order LDA line2
1750  CMP line1
1760  LDA line2+1
1770  SBC line1+1
1780  BCC jmperr
1790 :
1800 :
1810 \ **** Where to move it ?
1820 :
1830  LDA final
1840  LDX final+1
1850  LDY #return
1860  JSR convert
1870  BCS jmperr
1880 :
1890 \ Save it
1900  LDA temp
1910  STA target
1920  LDA temp+1
1930  STA target+1
1940 :
1950 :
```

```
1960 \ **** Is target above first line ?
1970 :
1980  LDA target
1990  CMP line1
2000  LDA target+1
2010  SBC line1+1
2020  BCC find \ Below
2030 :
2040 \ ... and below end line ?
2050  LDA line2
2060  CMP target
2070  LDA line2+1
2080  SBC target+1
2090  BCC find \ Line no. is above
2100 :
2110 .jmperr JMP synterr
2120 :
2130 :
2140 \ **** Find 2 line no. addresses
2150 \ Exit with closest line addresses in addr1, addr2
2160 :
2170 .find LDX #0
2180  STX flag
2190  INX \ Skip opening 'return'
2200  STX loc
2210  LDA page
2220  STA loc+1
2230 :
2240 .find2 LDY #0
2250  LDA (loc),Y \ Hi-byte line no.
2260  STA hi_tmp \ Hold it
2270  INY \ Ready for lo-byte
2280  BIT flag
2290  BMI test2
2300 :
```

```
2310 \ Compare with first no.
2320  CMP #&FF
2330  BEQ jmperr \ First no. > last line
      no.
2340 :
2350  LDA (loc),Y \ Lo-byte line no.
2360  CMP line1
2370  LDA hi_tmp
2380  SBC line1+1
2390  BCC newline \ Not yet passed
2400 :
2410 \ Save details of this line
2420  LDA loc
2430  STA addr1
2440  LDA loc+1
2450  STA addr1+1
2460  DEC flag
2470  BNE newline \ Always
2480 :
2490 .test2 CMP #&FF
2500  BEQ overtop \ &FF end of program
2510 :
2520 \ Compare with second no.
2530  LDA line2
2540  CMP (loc),Y \ Lo-byte
2550  LDA line2+1
2560  SBC hi_tmp \ Hi-byte
2570  BCC overtop
2580 :
2590 .newline LDA loc
2600  CLC
2610  INY
2620  ADC (loc),Y \ line length
2630  STA loc
2640  LDA loc+1
2650  ADC #0
2660  STA loc+1
2670  BNE find2 \ Always
2680 :
```

```
2690 .overtop LDA loc
2700  STA addr2
2710  LDA loc+1
2720  STA addr2+1
2730 :
2740 :
2750 \ **** Test for move space above
       TOP
2760 :
2770  LDA addr2
2780  SEC
2790  SBC addr1 \ Difference ...
2800  PHP
2810  CLC
2820  ADC top \ ... added to top
2830  STA temp
2840  LDA addr2+1
2850  PLP
2860  SBC addr1+1
2870  CLC
2880  ADC top+1
2890  STA temp+1
2900 :
2910 \ Enough room below HIMEM ?
2920  LDA himem
2930  CMP temp
2940  LDA himem+1
2950  SBC temp+1
2960  BCS shift
2970  JMP noroom
2980 :
2990 :
3000 \ **** Move block to end
3010 :
3020 .shift LDA addr2
3030  SEC
3040  SBC addr1
3050  STA size
3060  STA gap
3070  LDA addr2+1
```

```
3080  SBC addr1+1
3090  STA size+1
3100  STA gap+1
3110  LDA addr1
3120  STA loc
3130  LDA addr1+1
3140  STA loc+1
3150  LDA top
3160  STA dest
3170  LDA top+1
3180  STA dest+1
3190  JSR lmove
3200 :
3210 :
3220 \ **** Close gap
3230 :
3240 \ Include end block in shift
       ready for next exit test
3250  LDA top
3260  SEC
3270  SBC addr1
3280  STA size
3290  LDA top+1
3300  SBC addr1+1
3310  STA size+1
3320  LDA addr2
3330  STA loc
3340  LDA addr2+1
3350  STA loc+1
3360  LDA addr1
3370  STA dest
3380  LDA addr1+1
3390  STA dest+1
3400  JSR lmove
3410 :
3420 \ Is shift to end of program ?
       (Target => last line no.)
3430  LDA target
3440  CMP final
3450  LDA target+1
```

```
3460  SBC final+1
3470  BCC gettgt
3480  JMP exit \ No more moves needed
3490 :
3500 :
3510 \ **** Find address of line above target
3520 :
3530 .gettgt LDY #1 \ Skip 'return'
3540  STY loc
3550  LDA page
3560  STA loc+1
3570  INY \ To 2
3580 :
3590 .gettgt2 DEY \ To 1
3600  LDA (loc),Y
3610  CMP target
3620  DEY \ To 0
3630  LDA (loc),Y
3640  SBC target+1
3650  BCS gettgt3 \ Passed it
3660 :
3670  INY
3680  INY \ To 2
3690  LDA (loc),Y \ Line length
3700  CLC
3710  ADC loc
3720  STA loc
3730  BCC gettgt2
3740  INC loc+1
3750  BNE gettgt2 \ Always
3760 :
3770 .gettgt3 LDA loc
3780  STA tgtadr
3790  LDA loc+1
3800  STA tgtadr+1
3810 :
3820 :
3830 \ **** Open new gap
3840 :
```

```
3850  LDA top
3860  SEC
3870  SBC tgtadr
3880  STA size
3890  LDA top+1
3900  SBC tgtadr+1
3910  STA size+1
3920 :
3930  LDA top
3940  STA dest
3950  SEC
3960  SBC gap
3970  STA loc
3980  LDA top+1
3990  STA dest+1
4000  SBC gap+1
4010  STA loc+1
4020 :
4030 :
4040 \ **** Move memory block right
4050 :
4060 .rmove LDY #0
4070  LDX size+1
4080  BEQ rmvpart
4090 :
4100 .rmv2 DEC loc+1
4110  DEC dest+1
4120 .rmv3 DEY
4130  LDA (loc),Y
4140  STA (dest),Y
4150  CPY #0
4160  BNE rmv3
4170 :
4180  DEX
4190  BNE rmv2
4200 :
4210 .rmvpart LDX size
4220  BEQ rmvdone
4230 :
```

```
4240  DEC loc+1
4250  DEC dest+1
4260 .rmv4 DEY
4270  LDA (loc),Y
4280  STA (dest),Y
4290  DEX
4300  BNE rmv4
4310 :
4320 :
4330 \ **** Move block to gap
4340 :
4350 .rmvdone LDA gap
4360  STA size
4370  LDA gap+1
4380  STA size+1
4390  LDA top
4400  STA loc
4410  LDA top+1
4420  STA loc+1
4430  LDA tgtadr
4440  STA dest
4450  LDA tgtadr+1
4460  STA dest+1
4470  JSR lmove
4480 :
4490 .exit LDA #&FF
4500  LDY #0
4510  STA (top),Y
4520 :
4530  LDA #&8A \ Stuff buffer
4540  LDX #0
4550  LDY #ASC "R"
4560  JSR osbyte
4570  LDY #ASC "E"
4580  JSR osbyte
4590  LDY #ASC "N"
4600  JSR osbyte
4610  LDY #ASC "."
4620  JSR osbyte
4630  LDY #return
```

```
4640  JMP osbyte \ RTS to BASIC
4650 :
4660 :
4670 \ **** Subroutines ****
4680 :
4690 \ Convert decimal to hex
4700 \ 'loc' contains correct address
      in buffer
4710 \ Enter with default line no. in
      AX and delimiter in Y
4720 :
4730 .convert STA temp
4740  STX temp+1
4750  STY mark
4760 :
4770 \ Is delimiter first character ?
4780  LDY #0
4790  LDA (loc),Y
4800  CMP mark
4810  BEQ convdone \ No digits to
     convert
4820 :
4830 \ Else clear temp
4840  STY temp \ 0
4850  STY temp+1
4860 :
4870 \ Top of loop
4880 .conv2 LDA (loc),Y
4890  CMP mark \ Delimited ?
4900  BEQ convdone
4910 :
4920  CMP #ASC "0" \ For what's below
4930  BCC converr \ Wrong delimiter
4940 :
4950  ASL temp \ times 2
4960  ROL temp+1
4970  LDA temp \ Save in AX
4980  LDX temp+1
4990  ASL temp \ times 8
5000  ROL temp+1
```

```
5010  ASL temp
5020  ROL temp+1
5030  CLC
5040  ADC temp \ add times 2
5050  STA temp
5060  TXA
5070  ADC temp+1
5080  STA temp+1
5090 :
5100  LDA (loc),Y \ Next ASCII
5110  SEC
5120  SBC #ASC "0" \ To hex
5130  CLC
5140  ADC temp \ Add it on
5150  STA temp
5160  BCC conv3
5170  INC temp+1
5180 :
5190 .conv3 BIT temp+1
5200  BMI converr \ line no. > 32767 ?
5210 :
5220  INY
5230  CPY #6 \ Too long ?
5240  BNE conv2 \ If so, fall through
5250 :
5260 .converr SEC
5270  RTS
5280 :
5290 .convdone INY \ Point to next
      byte
5300  TYA \ Ready for re-entry
5310  CLC
5320  ADC loc
5330  STA loc \ Hi-byte must stay on
      same page
5340  RTS \ With carry clear as OK flag
5350 :
5360 :
5370 \ **** Move memory block left
5380 :
```

```
5390 .lmove LDY #0
5400  LDX size+1
5410  BEQ lmvpart
5420 :
5430 .lmvpage LDA (loc),Y
5440  STA (dest),Y
5450  INY
5460  BNE lmvpage
5470  INC loc+1
5480  INC dest+1
5490  DEX
5500  BNE lmvpage
5510 :
5520 .lmvpart LDX size
5530  BEQ lmvdone
5540 :
5550 .lmvlast LDA (loc),Y
5560  STA (dest),Y
5570  INY
5580  DEX
5590  BNE lmvlast
5600 :
5610 .lmvdone RTS
5620 :
5630 :
5640 :
5650 ]
5660 :
5670 synterr = P%
5680 M$ = CHR$0 + CHR$4 + CHR$7 +
     "Silly"
5690 $synterr = M$
5700 P% = P% + LEN M$
5710 ?P% = 0
5720 :
5730 noroom = P%
5740 M$ = CHR$0 + CHR$0 + CHR$7 + "No
     room"
5750 $noroom = M$
5760 P% = P% + LEN M$
```

```
5770 ?P% = 0
5780 P% = P% + 1
5790 :
5800 NEXT
5810 :
5820 END
```

SHAPE MAKER

SHAPER is a BASIC program which analyses a shape drawn on the screen and converts it into a string of user-defined characters, so that you can re-display it at different places over the screen. The program is written as a BASIC procedure (**PROCfill**) and is embodied within a larger demonstration program.

The shape that we are using for our demonstration is a circle with a thick circumference line. It will take up 36 character spaces and is set in a box to define its boundaries.

Since we will exceed the normal 32 characters available for user-defined characters, we have to 'explode' the character set. Please refer to the notes on the Character Generator for a full description of this process. The program starts by checking whether PAGE has been reset in order to make room for the extra characters. If your shape is of a different size, you'll need to make the appropriate adjustments.

At the beginning of the program there are two variables, **HZ** and **VT**, which specify the horizontal and vertical dimensions of our shape. If you look at the source listing, you'll see how the program uses these two variables to work out whether we need to explode the character set and, if so, where to set PAGE. We call on OSBYTE routine 131 (see page 431 of the User Guide for details) to establish whether you have a disk- or tape-based system.

Before we draw the circle itself, we set up an array of sine and cosine values. Although this takes up extra space in the computer's memory, it is faster to look up the array during run-time than to carry out the same calculations of trigonometric values over and over again. The circle itself is drawn in quadrants, using the same routine as appears in the Circle Draw program.

The shape is then analysed with the POINT command and the individual pixels are transformed into 8 × 8 character cells. They are re-plotted in red simply to demonstrate the progress of the scan.

The scan itself is carried out in the form of a vertical boustrophedon, that is, in columns moving alternately up and down the screen, rather than in a more usual left-to-right row scan. The reason for this is so that the final shape can be sliced up with LEFT$ and RIGHT$ commands. In that way, you can 'pull' it on from the left of the screen and 'push' it off at the other side without wrap-around.

Although the demonstration uses a circle, you can obviously create your own shapes, with mathematical routines or from the keyboard (perhaps using the cursor move keys) or even with a joystick.

```
10 REM **** SHAPE-MAKER ****
  20 :
  30 REM (c) Ian Trackman 1983
  40 :
  50 REM Draws a shape on the screen
     then converts it into a
     user-defined shape for printing
     anywhere else
  60 :
  70 HZ = 6 : REM Horizontal size
  80 VT = 6 : REM Vertical size
  90 :
 100 MEM = ((HZ * VT) DIV 32) * &100 :
     REM Every 32 characters above the
     first needs an extra &100 bytes of
     memory
 110 :
 120 REM Find lowest free RAM address
 130 A% = &83
 140 BASE = (USR(&FFF4) AND &FFFF00) DIV
     &100
 150 :
 160 REM Test that the character memory
     is sufficiently "exploded"
 170 IF PAGE <> BASE + MEM THEN PRINT
     "Reset PAGE to "; ~BASE + MEM " and
     re-run" : STOP
 180 :
 190 MODE 1
 200 VDU 23,1,0;0;0;0; : REM Cursor off
 210 :
 220 MEM = MEM DIV &100
 230 IF MEM > 0 THEN PROCoscli
 240 :
 250 PROCbox (HZ * &20,VT * &20) : REM
     Draw a box to show shape's area
 260 :
```

```
270 SZ = 2 : REM Drawing step size
280 PROCtrig
290 PROCcircle (&50,&60) : REM As a
    demo
300 GCOL 0,1
310 PROCfill (6,6)
320 PROCsetup
330 PRINT A$
340 PRINT TAB(0,30)
350 END
360 :
370 :
380 DEF PROCbox (X,Y)
390 VDU 29,&280;&200;
400 X = X + &20
410 Y = Y + &20
420 MOVE -X DIV 2,Y DIV 2
430 PLOT 1,X,0
440 PLOT 1,0,-Y
450 PLOT 1,-X,0
460 PLOT 1,0,Y
470 ENDPROC
480 :
490 :
500 DEF PROCcircle (R1,R2)
510 :
520 FOR R% = R1 TO R2 STEP 2
530   x% = 0
540   y% = R%
550   :
560   FOR A = 0 TO RAD 90 STEP RAD SZ
570     X% = R% * S((DEG A) / 2) + .5
580     Y% = R% * C((DEG A) / 2) + .5
590     FOR QX% = -1 TO 1 STEP 2
600       FOR QY% = -1 TO 1 STEP 2
610         MOVE x% * QX%,y% * QY%
620         DRAW X% * QX%,Y% * QY%
630         NEXT
640       NEXT
650     x% = X%
```

```
 660       y% = Y%
 670       NEXT
 680     :
 690     NEXT
 700  :
 710  ENDPROC
 720  :
 730  :
 740  DEF PROCfill (X,Y)
 750  DIM B(7)
 760  VDU 29,&260 - X*&10;&21C + Y*&10;
 770  N = 128
 780  :
 790  FOR I = 1 TO X
 800    :
 810    IF I MOD 2 THEN FOR J = 1 TO Y
 820      IF I MOD 2 = 0 THEN FOR J = Y
          TO 1 STEP -1
 830        :
 840        FOR K = 0 TO 7 STEP 2
 850          FOR L = 1 TO 0 STEP -1
 860            :
 870            B = 0
 880            :
 890            FOR M = 0 TO 7
 900              XT = I*&20 + M*4
 910              YT = -J*&20 - (K +
                  L)*4
 920              IF POINT(XT,YT) THEN
                  PLOT 69,XT,YT
 930              IF POINT(XT,YT) THEN B
                  = B + 2 ^ (7 - M)
 940              NEXT M
 950            :
 960            B(K + L) = B
 970            :
 980            NEXT L
 990          NEXT K
1000        :
```

```
1010        VDU 23,N,B(0),B(1),B(2),B(3),
            B(4),B(5),B(6),B(7)
1020        N = N + 1
1030        :
1040        NEXT J
1050      NEXT I
1060    :
1070    ENDPROC
1080    :
1090    :
1100    DEF PROCoscli
1110    REM Explode the character set
1120    DIM OSCL 10
1130    $OSCL = "FX 20," + STR$ MEM
1140    X% = OSCL
1150    Y% = OSCL DIV &100
1160    CALL &FFF7
1170    ENDPROC
1180    :
1190    :
1200    DEF PROCsetup
1210    A$ = STRING$(96," ") : REM
        "Declare" the string size -
        prevents string garbage
1220    A$ = "" : REM Now clear it
1230    N = 128 : REM CHR$ counter
1240    :
1250    REM Connect defined shapes in a
        vertical "boustrephon" with
        cursor move keys
1260    REM This allows LEFT$ and RIGHT$
        sectioning of the shape
1270    FOR X = 1 TO 6
1280      :
1290      IF X MOD 2 THEN FOR Y = 1 TO 5
1300        IF X MOD 2 = 0 THEN FOR Y = 5
            TO 1 STEP -1
1310          A$ = A$ + CHR$N + CHR$8
```

```
1320            IF X MOD 2 THEN A$ = A$ +
                CHR$10 ELSE A$ = A$ +
                CHR$11 : REM Up and down on
                alternate passes
1330            N = N + 1
1340            NEXT
1350          :
1360          A$ = A$ + CHR$N
1370          N = N + 1
1380          NEXT
1390        :
1400        ENDPROC
1410        :
1420        :
1430        DEF PROCtrig
1440        REM Create array of trig
            values.
1450        REM Faster than calculating
            during run-time
1460        DIM S(90 DIV SZ),C(90 DIV SZ)
1470        :
1480        FOR A% = 0 TO 90 DIV SZ
1490          S(A%) = SIN (RAD A%*2)
1500          C(A%) = COS (RAD A%*2)
1510          NEXT
1520        :
1530        ENDPROC
```

SIDEWAYS CHARACTERS

The routine which forms the basis of this program was originally part of a program called 'Sideways', which is included in the 'Programs from the Computer Programme' Pack. The routine has since been enhanced to make it more versatile.

The program is called TWIST. It lets you display any character sideways (left or right) or upside-down. The routine itself is written in assembly language, but the demonstration is in BASIC, so **CHAIN** or **LOAD**-and-**RUN** it.

First, a business letter is typed on the screen the right way up. Then, when a key is pressed, the first part of the letter is reprinted sideways, waiting for you to finish it off. Characters typed in from the keyboard will be displayed sideways.

The machine-code routine takes up 67 bytes and makes use of CHR$224. You can substitute any free user-definable character in line 1020 if your program is already using CHR$224. The routine is **CALL**ed by **PROCtwist**, which simply 'pokes' the required ASCII character into a location in zero-page, where it is collected on entry to the routine.

Another location, labelled **type**, controls the orientation of the output character. If it contains 0, characters will appear twisted to the right. With &80 (decimal 128) in it, the characters will be produced upside-down and &40 (decimal 64) will cause a twist to the left. In the demonstration, it is set to 0 in line 400. Of course, altering the value of **type** merely changes the orientation of the characters and not of the entire screen display, so that if you change the value of **type** and re-run the program, you'll either have to use a mirror or stand on your head to make sense of what appears on the screen!

When displaying a screenful of characters, the trick effect is clearly enhanced if the layout of the page is also sideways. You will see from the demonstration that we have had to use joined cursors **(VDU 5)** and **MOVE** commands to perform this trick. We also needed to write our own rudimentary line-feed and carriage-return routines.

Since the BASIC part of the program is only a demonstration for fun and to illustrate the speed of the machine-code, we haven't added complete routines to trap and handle the cursor-move and edit keys; nor, indeed, for scrolling.

Where we think that you will actually find the routine most useful is in diagrams and games. You can label graphs down the edges without having to make up an entire set of user-defined characters. You can also give the idea of turning movement in a game or moving diagram by making up your own symbols and then rotating them as you go round corners or change direction. (You're obviously not limited to the letters of the alphabet - any character, keyboard or user-defined, will work.)

If you are interested in understanding how the matrix inversion routines work, you may care to study the BASIC translation of the right-twist routine which we've added at the end of the program. It's only there for information. As it is never used, you don't need to copy it when you use the routine in your own programs.

```
  10 REM **** TWIST ****
  20 :
  30 REM (c) Ian Trackman 1982, 1983
  40 :
  50 REM This program demonstrates how
     the standard character set can be
     re-defined in a different rotation
  60 :
  70 ON ERROR GOTO 800
  80 :
  90 MODE 4
 100 VDU 23,1,0;0;0;0; : REM No cursor
 110 VDU 28,0,29,39,1 : REM New screen
     size
 120 :
 130 PROCassemble
 140 :
 150 COLOUR 0
 160 COLOUR 129
 170 CLS
 180 PRINT ''
 190 :
 200 REM Print letter normally
 210 REPEAT
 220   READ LIN$
 230   PRINT TAB(7) LIN$
 240   UNTIL LIN$ = "    "
 250 :
```

```
260 *FX 15,0
270 K = GET
280 RESTORE
290 :
300 REM Print letter sideways
310 CLS
320 VDU 5 : REM Join cursors
330 GCOL 4,0 : REM Inverting
340 GCOL 0,129
350 GAP = &1C : REM Gap between
    characters
360 MARGIN = 5 * GAP
370 X = &488 : REM "Top of page" gap
380 LMARG = &3FF - MARGIN
390 Y = LMARG
400 ?type = 0 : REM &80 for
    upside-down, &40 for
    anti-clockwise
410 :
420 REPEAT
430   READ LIN$
440   :
450   FOR I% = 1 TO LEN LIN$
460     MOVE X, Y
470     PROCtwist (MID$(LIN$,I%,1))
480     Y = Y - GAP
490     NEXT
500   :
510   Y = LMARG
520   X = X - &28
530   :
540   UNTIL LIN$ = "  "
550 :
560 REM Print keyboard characters
    sideways
570 *FX 15,0
580 X = X - &28
590 :
```

```
600 REPEAT
610   MOVE X,Y
620   PROCtwist ("_") : REM Print
      "cursor"
630   A$ = GET$
640   MOVE X,Y
650   PROCtwist ("_") : REM Remove
      cursor
660   MOVE X,Y
670   IF A$ = CHR$13 THEN Y = LMARG : X
      = X - &28 : UNTIL FALSE : REM If
      'Return', back for another
      character else ...
680 :
690 MOVE X,Y
700 PROCtwist (A$)
710 Y = Y - GAP
720 IF Y < MARGIN + GAP THEN Y = LMARG
    : X = X - &28 : REM New line
730 :
740 UNTIL FALSE
750 :
760 END
770 :
780 :
790 REM Error trap
800 MODE 6
810 IF ERR <> 17 THEN REPORT : PRINT "
    at line "; ERL
820 END
830 :
840 :
850 DEF PROCtwist (L$)
860 ?char = ASC L$
870 CALL Mcode
880 ENDPROC
890 :
900 :
```

```
 910 DEF PROCassemble
 920 REM Creates a machine-code program
     for matrix inversion
 930 :
 940 DIM Mcode 100 : REM Space for
     machine code program
 950 :
 960 char = &70 : REM Use 9 bytes on
     zero page for speed
 970 type = &80
 980 :
 990 oswrch = &FFEE
1000 osword = &FFF1
1010 :
1020 vdu_char = 224 : REM Use any spare
     ASCII character
1030 :
1040 opt = 2 : REM No display
1050 :
1060 FOR I% = 0 TO opt STEP opt
1070   P% = Mcode
1080   [OPT I%
1090   :
1100   \ Call OS 'read character
         definition' routine
1110    LDA #&A
1120    LDX #char
1130    LDY #char DIV &100
1140    JSR osword
1150   :
1160   \ Create new character with VDU
         23
1170    LDA #23 \ Start VDU 23 command
1180    JSR oswrch
1190    LDA #vdu_char
1200    JSR oswrch
1210    LDY #8 \ Twist 8 bytes
1220   :
```

```
1230   .test LDX #8 \ Bit counter
1240    BIT type
1250    BMI reverse \ Bit 7 set
1260    BVS anti \ Bit 6 set
1270   :
1280   \ Twist 8 bits clockwise
1290   .loop1 ROL char,X \ Rotate one
        bit from each byte into carry
        ...
1300    ROL A \ ... and collect it
1310    DEX
1320    BNE loop1 \ Done 8 bits ?
1330    BEQ print \ always
1340   :
1350   \ To twist upside-down -
1360   .reverse LDA char,Y \ Take each
        byte ...
1370    STA char \ ... hold it
1380   .loop2 ASL char \ Reverse each
        bit ...
1390    ROR A \ ... and save it
1400    DEX
1410    BNE loop2
1420    BEQ print \ Always
1430   :
1440   \ To twist anti-clockwise -
1450   .anti ROR char,X
1460    ROR A
1470    DEX
1480    BNE anti
1490   :
1500   .print JSR oswrch \ Print a
        byte
1510    DEY
1520    BNE test \ Done 8 bytes ?
1530   :
1540    LDA #vdu_char \ Print new
        character ...
1550    JMP oswrch \ ... and RTS to
        BASIC
```

```
1560    ]
1570    NEXT
1580 :
1590 ENDPROC
1600 :
1610 :
1620 REM Here's the same routine in
     Basic
1630 :
1640 DIM X% 8, B%(8)
1650 Y% = X% DIV &100
1660 A% = 10
1670 :
1680 ?X% = ASC L$
1690 CALL &FFF1
1700 :
1710 H% = &80
1720 :
1730 FOR I% = 1 TO 8
1740   B%(I%) = 0
1750   H2% = 1
1760   :
1770   FOR J% = 1 TO 8
1780     IF (X%?J% AND H%) THEN B%(I%) =
         B%(I%) + H2%
1790     H2% = H2% * 2
1800     NEXT
1810   :
1820   H% = H% DIV 2
1830   NEXT
1840 :
1850 VDU 23,224,B%(1),B%(2),B%(3),
     B%(4),B%(5),B%(6),B%(7),B%(8)
1860 VDU 224
1870 :
1880 :
1890 REM Demo data
1900 DATA "        15, HIGH STREET"
1910 DATA "             ANYTOWN"
1920 DATA " "
```

```
1930 DATA " "
1940 DATA "J. Smith Esq.     1st March"
1950 DATA "12, The Avenue"
1960 DATA "London N.W.18"
1970 DATA " "
1980 DATA "Dear Mr Smith,"
1990 DATA " "
2000 DATA "Thank you for your letter"
2010 DATA "of 18th February."
2020 DATA " "
2030 DATA "I confirm that next Monday"
2040 DATA "is a suitable date for our"
2050 DATA "meeting and I look forward"
2060 DATA "to seeing you."
2070 DATA "  "
2080 DATA "Yours sincerely,"
2090 DATA " "
2100 DATA "Joe Bloggs"
2110 DATA "   "
```

SORTING IT ALL OUT

Three factors control how well a sort works. The first is, naturally, how good the sorting algorithm is and how well it has been translated into the particular computer language (BASIC, in our case). The second factor is the total number of items in the list to be sorted. Some sorts work well with a small list, but work much more slowly as the size of the list grows (the so-called 'exponential explosion'). The third factor is how badly the list is out of order. Some sorts work very fast with a list which is nearly in order; others work at the same rate, whatever the order of the list. The last two factors mean that there isn't really a 'best' sorting method - it depends on your requirements.

The Demonstration Programs

There are seven demonstration programs on your tape. They are named:
BUBLSRT
SLCTSRT
INDXSRT
SHELSRT
SHL2SRT
QUIKSRT
HEAPSRT

Each of the programs follows the same format. There is a main program which:
a) sets up an array of 100 random numbers
b) calls PROCsort (the actual sorting procedure)
c) displays the sorted array and various information about the way in which the sort has worked.

Here is the main routine from BUBLSRT - the Bubble Sort demonstration:

```
 10 REM **** BUBBLE SORT ****
 20 :
 30 CLS
 40 @% = 4 : REM Print formatting
 50 N% = 100 : REM No. of items
 60 DIM A(N%)
 70 I% = RND(-1) : REM Reset
    randomizer
 80 :
 90 FOR I% = 1 TO N%
100   A(I%) = RND(N%)
110   PRINT A(I%);
```

```
120     NEXT
130  :
140  PRINT
150  COMP = 0
160  SWAP = 0
170  TIME = 0 : REM Reset timer
180  :
190  PROCsort
200  :
210  NOW = TIME
220  :
230  FOR I% = 1 TO N%
240     PRINT A(I%);
250     NEXT
260  :
270  @% = &90A : REM Reset print format
280  PRINT ' "Time "; NOW/100 "
     Seconds"
290  PRINT; N% " Numbers"
300  PRINT; COMP " Comparisons"
310  PRINT; SWAP " Swaps" '
320  END
```

Line 40 (@ % =4) simply formats the colums of numbers so that they fit properly onto the screen. Normal formatting is reset in line 270 before the program ends.

In line 50, the variable N% sets up the number of items to be sorted. As we mentioned above, it doesn't necessarily follow that if you double the number of items, each of the various sorts will take twice as long. Some are comparatively faster, but some take more than double the time. We'll come back to this point shortly.

Line 70 resets BASIC's randomiser. Each of the series of random numbers generated by the different routines will be the same, so that valid comparisons can be made.

A loop then creates an array of N% random numbers. An interesting experiment is to re-run the sorts with the numbers already in order by changing line 100 to:

A(I%)=101–I%

Then run the sort again with the numbers in reverse order altering line 100 to:

A(I%)=101–I%

The computer's internal timer is set to zero just before the test starts and then read as soon as the sort is completed.

The variables COMP and SWAP count how many comparisons and swaps have been made. Although time is usually the most important factor, these variables will give you some indication of the efficiency of the sort routine, particularly if you change the size of the array or the order of its contents.

Using the Routines in Your Own Programs

The procedures containing the sorting routines are almost completely self-contained. The only things that need to be done beforehand are to dimension the array(s) that will be needed and, of course, to put some data into the main array. All the variables used by the procedures have been declared as 'local' variables so that they will not conflict with any variables with the same names in your main program.

The COMP and SWAP variables are included in the procedures only because we want their values afterwards for display. They don't affect the actual sorting process and all references to them should be deleted.

If you delete the main program (lines 10-340), you can append the sort procedure to your own program, using one of the methods described on pages 402-3 of the User Manual.

You can use the routines to sort string arrays simply by changing the variables A() to A$() and T (whenever it is used) to T$, in which case the array variables will be sorted according to their ASCII values.

As written, the routines always sort the entire array - the normal case. If you want to sort only a part of the array, you can send the start and end 'pointers' to the sort procedure as parameters, e.g. PROCsort (FIRST, LAST). You'll then have to re-define the loop counters (I and N% in the demonstration programs) to correspond to the new values.

Which Sort for You?

The table below sets out the sorting times which we obtained from the six routines, using three different array sizes (25, 100 and 150 items) in random order, correct order and reverse order. We've also calculated the average times for each routine.

Before we ran the timing tests, we removed the COMP and SWAP variables from the procedures. If you want to make the routine work even faster, you can remove the blank spaces, which we put in to make the listings more legible.

You can also put several statements on to one line. However, if you are that anxious to save a few more micro-seconds, perhaps you should be thinking of using a sort written in machine-code. A well-designed machine-code sort could be over 150 times as fast as its BASIC equivalent!

TABLE OF SORTING-ROUTINE SPEEDS

Number of items		25	100	150	Average
BUBBLE	Random	1·34	21·00	49·44	29·93
	1 – N%	**0·01**	**0·24**	**0·37**	**0·21**
	N% – 1	1·95	31·00	69·75	34·23
	Average	1·10	17·41	39·85	19·46
HEAP	Random	0·88	5·27	**8·65**	4·93
	1 – N%	1·01	5·71	9·28	5·33
	N% – 1	0·86	4·98	8·13	4·66
	Average	0·92	5·32	8·69	4·97
INDEX	Random	1·48	21·98	49·02	24·16
	1 – N%	1·47	21·88	48·84	24·06
	N% – 1	1·47	21·87	48·84	24·06
	Average	1·47	21·91	48·90	24·09
QUICK	Random	0·83	**4·97**	8·76	**4·85**
	1 – N%	1·73	24·99	55·53	27·42
	N% – 1	1·69	24·73	55·06	27·16
	Average	1·42	18·23	39·78	19·81
SELECT	Random	**0·70**	8·86	19·31	9·62
	1 – N%	0·64	8·51	18·69	9·28
	N% – 1	0·85	11·81	26·10	12·92
	Average	0·73	9·73	21·37	10·61
SHELL	Random	1·03	9·24	13·14	7·80
	1 – N%	0·26	1·51	2·68	1·48
	N% – 1	**0·66**	**4·08**	**7·01**	**3·92**
	Average	**0·65**	**4·94**	**7·61**	**4·40**

Best results are shown in **bold type.**

How Do They Do It?

The next section attempts to explain the mechanics of each type of sort. You don't need to understand how the routine works in order to use it, but if you do, you might like to play around with the routine to try to improve its speed or, say, to sort the array into descending, rather than ascending, order.

To assist in the explanations, we're going to use a pack of playing cards to represent our array. Let's arbitrarily decide that, for our purposes, a pack of cards in correct order will start with the Ace of Clubs as the first card, going through the Aces in the order Clubs, Diamonds, Hearts and Spades, and then repeat the suit order for the Twos, Threes, etc. The last card will therefore be the King of Spades. The only problem with using a pack of cards is that there are no duplicates, whereas in an array there may well be. So, if you really want to be realistic, mix up two or more packs and then deal yourself 52 cards at random.

The Bubble Sort

The Bubble Sort is probably the easiest routine to understand.

Starting at the beginning of the array, adjacent items are compared. If they are out of order, they are swapped. In that way, the largest item will 'bubble' through to the end of the array. The same process is repeated to bring the next largest number to one position before the end. Every time that a swap is made, a 'flag' (the variable F%) is set to FALSE. The swapping continues until no more swaps are needed, that is, until the flag remains TRUE.

Take your shuffled pack of cards and a coin. Turn the coin heads up. Look at the first two cards. If the first card should be lower in the pack than the second, swap them over. Look at the second and third cards. Again, swap them if necessary. Repeat this process until you reach the end of the pack. The first time that you make a swap on each 'pass' through the pack, turn the coin over and then leave it tails up until you reach the end of the pack. At the end of the first pass, you should be left with the King of Spades (the 'largest' card) at the bottom of the pack.

If the coin is heads up, you didn't make any swaps. This means that the pack is in order and you have finished. However, if the coin is tails up, there is more work to be done. Turn the coin back to heads and restart the 'compare and swap' process from the beginning of the pack. Since the King of Spades is already at the bottom of the pack, you only have to compare 51 cards this time. Now you'll get the King of Hearts to one before the end. Each time you go through the pack, stop one card sooner, since you know that the cards below are in order.

In the computer program, the coin is represented by the variable F% (for 'flag') and the stopping point is held in the variable P% (for 'pointer'). Notice how we need an extra variable T to help in the swap. If you don't understand why, try swapping the contents of a glass of water with the contents of a glass of milk without using a third glass! (T is a floating point variable in case the item to be swapped is a decimal number or outside the range of an integer number.)

The speed of the Bubble Sort depends on how ordered the items are to begin with. There is a significant difference between an array in order (with 100 items, the demonstration program takes about 0·2 seconds) and an array which is in reverse order (about 31 seconds).

As you will see from the table, the sorting time also increases out of direct proportion to the size of the array. A fourfold increase in the number of items causes a fifteenfold increase in the sorting time.

```
 10 REM **** BUBBLE SORT ****
 20 :
 30 CLS
 40 @% = 4 : REM Print formatting
 50 N% = 100 : REM No. of items
 60 DIM A(N%)
 70 I% = RND(-1) : REM Reset
    randomizer
 80 :
 90 FOR I% = 1 TO N%
100   A(I%) = RND(N%)
110   PRINT A(I%);
120   NEXT
130 :
140 PRINT
150 COMP = 0
160 SWAP = 0
170 TIME = 0 : REM Reset timer
180 :
190 PROCsort
200 :
210 NOW = TIME
220 :
230 FOR I% = 1 TO N%
240   PRINT A(I%);
250   NEXT
260 :
270 @% = &90A : REM Reset print format
280 PRINT ' "Time "; NOW/100 "
    Seconds"
290 PRINT; N% " Numbers"
300 PRINT; COMP " Comparisons"
310 PRINT; SWAP " Swaps" '
320 END
330 :
340 :
350 DEF PROCsort
360 LOCAL F%,I%,P%,T
370 P% = N% - 1
380 :
```

```
390 REPEAT
400   F% = TRUE
410   :
420   FOR I% = 1 TO P%
430     IF A(I%) > A(I%+1) THEN T =
        A(I%) : A(I%) = A(I%+1) :
        A(I%+1) = T : F% = FALSE : SWAP
        = SWAP + 1
440     COMP = COMP + 1
450     NEXT
460   :
470   P% = P% - 1
480   UNTIL F%
490 :
500 ENDPROC
```

The Selection Sort

In the Bubble Sort, we physically swapped adjacent cards throughout the pack. Let's imagine that before we started, the King of Spades was the 27th card. After the first pass, that card became the 52nd card but all of the other cards stayed unsorted. So why don't we just look for the King of Spades and swap it for the bottom card? Then the next time, we'll look for the King of Hearts and swap it with the 51st card and so on, throughout the pack.

That's the idea behind the Selection Sort, except that, since loops run faster in BASIC forwards than backwards, we'll begin by looking for the Ace of Clubs and bringing it to the top of the pack. When you do the sort with a pack of cards, you might not see why we have to do a swap, since you can simply extract the Ace of Clubs from somewhere in the middle of the pack and put it on the top. If you don't do a swap, you are effectively going right through the pack, putting the original first card in the second position, the second card in the third position and so on - very time-consuming for the computer.

Again, we could use our coin as a 'flag' to tell us whether a swap is needed, but, with a small array, this actually slows down the computer program. It takes longer to check the flag than to complete the entire loop. With a large array, the 'trade-off' time might warrant testing a flag.

```
 10 REM **** SELECTION SORT ****
 20 :
 30 CLS
 40 @% = 4
 50 N% = 100
 60 DIM A(N%)
 70 I% = RND(-1)
 80 :
 90 FOR I% = 1 TO N%
100   A(I%) = RND(N%)
110   PRINT A(I%);
120   NEXT
130 :
140 PRINT
150 COMP = 0
160 SWAP = 0
170 TIME = 0
180 :
190 PROCsort
200 :
```

```
210 NOW = TIME
220 :
230 FOR I% = 1 TO N%
240   PRINT A(I%);
250   NEXT
260 :
270 @% = &90A
280 PRINT ' "Time "; NOW/100 "
    Seconds"
290 PRINT; N% " Numbers"
300 PRINT; COMP " Comparisons"
310 PRINT; SWAP " Swaps" '
320 END
330 :
340 :
350 DEF PROCsort
360 LOCAL I%,J%,P%,T
370 :
380 FOR I% = 1 TO N%-1
390   T = A(I%)
400   P% = I%
410   :
420   FOR J% = I% + 1 TO N%
430     IF A(J%) < T THEN T = A(J%) :
        P% = J% : SWAP = SWAP + 1
440     NEXT
450   :
460   T = A(I%)
470   A(I%) = A(P%)
480   A(P%) = T
490   COMP = COMP + 1
500   NEXT
510 :
520 ENDPROC
```

The Index Sort

This time, take a piece of paper. Write down the numbers 1 to 52 in a column. Look at the first card. Now go through the pack, and count up the number of cards which are lower in value than that first, 'control' card. Add one. Write down that number against 1 on your paper. For example, if the first card is Six of Diamonds, you would write down the number 22 in the first row of your list of numbers. That tells us that the first card in the pack is the 22nd highest in order.

In the previous sort routines, duplicates have been taken care of automatically. In the Index Sort, we would have a problem if two or more cards had the same value. To deal with this situation, if a row already has a number against it, you would have to move down until you found an empty row.

Keep repeating the process until you have been through the entire pack.

As you'll see from the table, this is a very slow sort routine, since it goes right through the entire array for every item in it. The reason that it's sometimes used is that, as its name suggests, it creates an index to the main array, which is left alone in its original order. This could be useful to avoid wasting memory space when swapping string arrays.

```
  10 REM **** INDEX SORT ****
  20 :
  30 CLS
  40 @% = 4
  50 N% = 100
  60 DIM A(N%),X%(N%)
  70 I% = RND(-1)
  80 :
  90 FOR I% = 1 TO N%
 100   A(I%) = RND(N%)
 110   PRINT A(I%);
 120   NEXT
 130 :
 140 PRINT
 150 COMP = 0
 160 SWAP = 0
 170 TIME = 0
 180 :
 190 PROCsort
 200 :
```

```
210 NOW = TIME
220 :
230 FOR I% = 1 TO N%
240   PRINT A(X%(I%));
250   NEXT
260 :
270 @% = &90A
280 PRINT ' "Time "; NOW/100 "
    Seconds"
290 PRINT; N% " Numbers"
300 PRINT; COMP " Comparisons"
310 PRINT; SWAP " Swaps" '
320 END
330 :
340 :
350 DEF PROCsort
360 LOCAL I%,J%,P%
370 :
380 FOR I% = 1 TO N%
390   P% = 1
400   :
410   FOR J% = 1 TO N%
420     IF A(I%) > A(J%) THEN P% = P% +
        1
430     COMP = COMP + 1
440     NEXT
450   :
460   REPEAT
470     IF X%(P%) THEN P% = P% + 1
480     UNTIL X%(P%) = 0
490   :
500   X%(P%) = I%
510   SWAP = SWAP + 1
520   NEXT
530 :
540 ENDPROC
```

The Shell Sort

We can see from the table that if the items in the array are already in order, the Bubble Sort is the fastest sorting routine. The idea of the Shell Sort is to concentrate on getting a gradually increasing section of the array sorted out as early on as possible, so that it won't need re-sorting on subsequent 'passes' through the array.

Unless you have a very large card-table, you'll probably need to lay out this card demonstration on the floor. Take your shuffled pack of cards and a coin. Put the coin heads up. Set out the cards, face up, in two rows of 26 cards one above the other. (In fact, you're splitting the total number of items to be sorted into two halves.) Go through the pack, but, instead of comparing adjacent cards as in the Bubble Sort, compare pairs of cards in columns. In effect, you are comparing the first card with the 26th card, the second card with the 27th card and so on, until you have made 26 comparisons. In each comparison, if the card in the top half is larger (i.e. higher in our arbitrary card order) than the card beneath it in the second half of the pack, swap the two cards over and, if the coin is not already set to tails, turn it over.

Turn the coin heads up. (The coin is a 'flag' - tails indicates that a swap has been made during the pass.) Collect up the cards in order (i.e. the first card in the top row first, the second card in the top row second and so on). Re-deal the cards into four rows of 13 cards each. The first card goes at the left of the top row, the second card to its right and so on. Starting with the first card, compare it with the card immediately below it. Swap them if necessary and set the coin to tails up. If the coin is tails up after the last comparison, put it back to heads and repeat the 'compare and swap' process. Keep doing so until the coin stays heads up. Notice now that the cards are already in order within each column of four cards.

Collect up the cards and re-deal them into eight rows of six cards each with four cards left over in a ninth row. The size of the row - six cards - is always half of the previous row size, ignoring fractional halves. Repeat the entire 'swap and compare' process until the coin stays heads up. The columns should now be in order.

Repeat the complete process twice more, once with rows of three cards and, finally, with one long column, one card wide. The last pass is, of course, a standard Bubble Sort.

A slightly slower variation of the Shell Sort (included on the tape as SHL2SRT) carries out a Selection Sort, rather than a 'mini Bubble Sort' on each pass through the array.

```
 10 REM **** SHELL SORT ****
 20 :
 30 CLS
 40 @% = 4
 50 N% = 100
 60 DIM A(N%)
 70 I% = RND(-1)
 80 :
 90 FOR I% = 1 TO N%
100   A(I%) = RND(N%)
110   PRINT A(I%);
120   NEXT
130 :
140 PRINT
150 COMP = 0
160 SWAP = 0
170 TIME = 0
180 :
190 PROCsort
200 :
210 NOW = TIME
220 :
230 FOR I% = 1 TO N%
240   PRINT A(I%);
250   NEXT
260 :
270 @% = &90A
280 PRINT ' "Time "; NOW/100 "
    Seconds"
290 PRINT; N% " Numbers"
300 PRINT; COMP " Comparisons"
310 PRINT; SWAP " Swaps" '
320 END
330 :
340 :
350 DEF PROCsort
360 LOCAL C%,F%,G%,I%,T
370 G% = N%
380 :
```

```
390 REPEAT
400   G% = G% DIV 2
410   C% = N% - G%
420   :
430   REPEAT
440     F% = FALSE
450     :
460     FOR I% = 1 TO C%
470       P% = I% + G%
480       IF A(I%) > A(P%) THEN T =
          A(I%) : A(I%) = A(P%) : A(P%)
          = T : F% = TRUE : SWAP = SWAP
          + 1
490       COMP = COMP + 1
500       NEXT
510     :
520     UNTIL F% = FALSE
530   :
540   UNTIL G% <= 1
550 :
560 ENDPROC
```

```
 10 REM **** SHELL/INSERTION SORT ****
 20 :
 30 CLS
 40 @% = 4
 50 N% = 100
 60 DIM A(N%)
 70 I% = RND(-1)
 80 :
 90 FOR I% = 1 TO N%
100   A(I%) = RND(N%)
110   PRINT A(I%);
120   NEXT
130 :
```

```
140 PRINT
150 COMP = 0
160 SWAP = 0
170 TIME = 0
180 :
190 PROCsort
200 :
210 NOW = TIME
220 :
230 FOR I% = 1 TO N%
240   PRINT A(I%);
250   NEXT
260 :
270 @% = &90A
280 PRINT ' "Time "; NOW/100 "
    Seconds"
290 PRINT; N% " Numbers"
300 PRINT; COMP " Comparisons"
310 PRINT; SWAP " Swaps" '
320 END
330 :
340 :
350 DEF PROCsort
360 LOCAL C%,F%,G%,I%,J%,T
370 G% = N%
380 :
390 REPEAT
400   G% = G% DIV 2
410   C% = N% - G%
420   :
430   FOR I% = 1 TO C% STEP G%
440     T = A(I%)
450     P% = I%
460     :
470     FOR J% = I% + G% TO N% STEP G%
480       IF A(J%) < T THEN T = A(J%) :
          P% = J%
490       COMP = COMP + 1
500       NEXT
510     :
```

```
520       T = A(I%)
530       A(I%) = A(P%)
540       A(P%) = T
550       SWAP = SWAP + 1
560       NEXT
570    :
580    UNTIL G% <= 1
590 :
600 ENDPROC
```

The Quick Sort

The Quick Sort, as its name indicates, is significantly faster for out-of-order arrays than the sort routines that we have looked at so far.

Take your pack of cards. Look at each card in turn. If it is lower than the Six of Hearts, put it on to a left-hand pile, otherwise put it on to a right-hand pile. (The reason for choosing the Six of Hearts is that it is half-way through a pack of cards which is already in order.) Next, pick up the left-hand pile and divide it into two smaller piles, using the 13th card, Five of Clubs, as the test card. Do the same with the right-hand pile, using the 39th card (Nine of Spades).

Continue subdividing into left- and right-hand piles. If you keep the piles in their correct positions relative to each other, you'll end up with the Ace of Clubs as the left-most card and the King of Spades as the right-most, with all of the other cards in order in between.

That's the general idea of the Quick Sort - but it's organised slightly differently for use on the computer. To work it with the cards, you'll need a paper-clip and a coin.

Reshuffle the cards and lay them out, face up, in one long row. Now carry out the following steps:

1 Put the paper-clip on the left-most card and the coin on the right-most card.

2 Compare the two 'marked' cards. If the left-hand card is higher than the right-hand one, swap them over.

3 Remove the coin (leaving the card where it is) and replace it on one card nearer towards the card with the paper-clip.

4 If the paper-clip and the coin are not on the same card, go back to Step 2.

5 When the coin and the paper-clip meet, that card is in its correct position. Furthermore, all the cards to its left are earlier in the pack and all of those to its right come later in the pack. Turn the card over to show that it has been sorted.

6 Now repeat the whole process, first with all of the cards to the left of the correct card and then with all of the cards to its right.

7 Gradually, you'll end up with more and more subdivisions consisting of only one card. When there are no more multiple-card subdivisions, the pack will be in order.

Obviously, the computer doesn't 'turn cards over'. Instead, it keeps track of duplicate sets of 'paper-clips and coins' - actually left-hand and right-hand pointers to the appropriate positions of the items in the array - until they all meet up, but the concept is the same. You'll notice from the program listing that two special arrays have to be created to hold the pointers so that the Quick Sort, although fast, takes up extra memory space. You would use it if speed were important and you had plenty of spare memory capacity.

```
 10 REM **** QUICK SORT ****
 20 :
 30 CLS
 40 @% = 4
 50 N% = 100
 60 DIM
    A(N%),SL%(SQR(N%)),SR%(SQR(N%))
 70 I% = RND(-1)
 80 :
 90 FOR I% = 1 TO N%
100   A(I%) = RND(N%)
110   PRINT A(I%);
120   NEXT
130 :
140 PRINT
150 COMP = 0
160 SWAP = 0
170 TIME = 0
180 :
190 PROCsort
200 :
210 NOW = TIME
220 :
230 FOR I% = 1 TO N%
240   PRINT A(I%);
250   NEXT
260 :
270 @% = &90A
280 PRINT ' "Time "; NOW/100 "
    Seconds"
290 PRINT; N% " Numbers"
300 PRINT; COMP " Comparisons"
310 PRINT; SWAP " Swaps"
320 END
330 :
340 :
```

```
350 DEF PROCsort
360 LOCAL F%,L%,L2%,P%,R%,R2%
370 P% = 1
380 SL%(P%) = 1
390 SR%(P%) = N%
400 :
410 REPEAT
420   L% = SL%(P%)
430   R% = SR%(P%)
440   P% = P% - 1
450   :
460   REPEAT
470     L2% = L%
480     R2% = R%
490     F% = TRUE
500     :
510     REPEAT
520       IF A(R2%) < A(L2%) THEN T =
          A(L2%) : A(L2%) = A(R2%) :
          A(R2%) = T : F% = NOT F% :
          SWAP = SWAP + 1
530       IF NOT F% THEN L2% = L2% + 1
          ELSE R2% = R2% - 1
540       COMP = COMP + 1
550       UNTIL L2% = R2%
560     :
570     IF L2% + 1 < R% THEN P% = P% +
        1 : SL%(P%) = L2% + 1 : SR%(P%)
        = R%
580     R% = R2% - 1
590     UNTIL L% >= R%
600   :
610   UNTIL P% = 0
620 :
630 ENDPROC
```

The Heap Sort

The final sorting routine on the tape is the Heap Sort, which makes use of the idea known as the 'binary tree'. Although it's only the fastest routine with 150 unsorted items, its timings average well and it is a good contender for the 'best all-rounder' prize. If you look at the listing, you'll also notice that it uses up very little extra memory space in which to operate.

It is more difficult to understand how it works than the other sorting routines. To make it easier, the following description, using the pack of cards, is a somewhat simplified version of the actual steps carried out by the computer.

Set out the cards in rows. Put one card in the first row, two in the second row, four in the third row, and so on, doubling up the number of cards in each row. You should end up with 21 cards in the bottom row. Now, so that you can see what is happening more easily, move the cards into the form of a pyramid. Each card in the upper rows (except where there are cards short in the bottom row) should be in a position half-way between the two cards below it. Now we are ready to begin the sort.

Go through the following steps:

1 Put a paper-clip on the last card at the right-hand end of the row above the bottom (the fifth row).

2 If there are no cards below the card with the paper-clip on it, move the paper-clip on to the card to its left. When you reach the left-hand end of a row, continue at the right-hand end of the row above.

3 If there are cards below the card with the paper-clip on it, compare the card with the paper-clip on it with the two cards below it. (With a pack of cards, there will be one card with a single card below it.) If either of the two lower cards comes earlier in the pack than the upper card, swap the smaller of them with the upper card. (Don't swap the lower two cards with each other.)

4 When you reach the left-hand edge of the fifth row, you'll notice that each of the cards in that row is earlier in the pack than the cards, if any, below it.

5 Repeat the process with the third row. This time, however, if you swap cards in the third and fourth rows, you may affect the relationship between the cards in the fourth and fifth rows. If you do, you'll have to carry out a secondary swap between the fourth and fifth rows to keep them in order.

6 After you have checked out the third row, move up to the second row and finally up to the top card. As before, you'll have to do subsidiary swaps if you disturb lower cards.

7 At this point, the Ace of Clubs, being the first card in the pack, should be at the top of the 'heap'. Put it on one side as the start of the sorted pack. The rest of the cards won't necessarily be in order, although any particular card will always be higher than all of the cards in the 'mini-pyramid' beneath it.

8 Next, you have to carry out the somewhat laborious process of gathering up the cards in row order, that is, starting at the left-hand card of the second row and working towards the right-most card in the bottom row. Then, you have to make a new pyramid and sort it out as before. This time, however, the cards will be almost in order so that very little swapping should be needed. The Ace of Diamonds will now be the top card. Remove it and put it under the Ace of Clubs.

9 Continue making new pyramids – or 'heaps' – until the old pack is exhausted. Each time, the top card will be the next card in order in the sorted pack.

```
 10 REM **** HEAP SORT ****
 20 :
 30 CLS
 40 @% = 4
 50 N% = 100
 60 DIM A(N%)
 70 I% = RND(-1)
 80 :
 90 FOR I% = 1 TO N%
100   A(I%) = RND(N%)
110   PRINT A(I%);
120   NEXT
130 :
140 PRINT
150 COMP = 0
160 SWAP = 0
170 TIME = 0
180 :
190 PROCsort
200 :
210 NOW = TIME
220 :
230 FOR I% = 1 TO N%
240   PRINT A(I%);
250   NEXT
260 :
270 @% = &90A
280 PRINT ' "Time "; NOW/100 "
    Seconds"
```

```
290 PRINT; N% " Numbers"
300 PRINT; COMP " Comparisons"
310 PRINT; SWAP " Swaps" '
320 END
330 :
340 :
350 DEF PROCsort
360 LOCAL I%,J%,T
370 J% = N%
380 :
390 FOR I% = N% DIV 2 TO 1 STEP -1
400   T = A(I%)
410   PROCsubsort
420   NEXT
430 :
440 I% = 1
450 :
460 FOR J% = N%-1 TO 1 STEP -1
470   T = A(J%+1)
480   A(J%+1) = A(1)
490   PROCsubsort
500   NEXT
510 :
520 ENDPROC
530 :
540 :
550 DEF PROCsubsort
560 LOCAL F%,K%,L%
570 K% = I%
580 F% = FALSE
590 SWAP = SWAP + 1
600 :
610 REPEAT
620   COMP = COMP + 1
630   L% = K% + K%
640   IF L% > J% THEN UNTIL TRUE :
      A(K%) = T : ENDPROC
650 :
```

```
660 IF L% < J% THEN IF A(L%+1) > A(L%)
    THEN L% = L% + 1
670 IF T < A(L%) THEN A(K%) = A(L%) :
    K% = L% ELSE F% = TRUE
680 UNTIL F%
690 :
700 A(K%) = T
710 ENDPROC
```

SPACER

SPACER is a utility, written in machine-code, which adds spaces around keywords and other selected items in a BASIC program, so making it easier to read, edit and debug. To a large extent, it reverses the effect of the Crunch utility.

There are two versions of the program on the tape. SPACER is the version for use with tape-based computers and resides between &E00 and &FFF. SPACERDISK is for use with disks and is loaded between &1700 and &18FF. Please refer to 'Using the Programming Utilities' for installation instructions. Other than the addresses at which they start, the two programs operate identically.

As the utility is co-resident, you can load it before or after you get your BASIC program into memory. Once the utility is in memory, start it working with **CALL &E00** (tape version) or **CALL &1700** (disk version).

The routine begins by setting Mode 7 and displaying the message **Uncrunching**... After a short while - just how long depends on the length of your program and how many spaces need to be inserted - the prompt will return, leaving the spaced BASIC program in memory.

There is a table at the end of this section which sets out the BASIC keywords in the numerical order of their tokens. Spacer will insert a space, provided that there isn't one there already, around all of the keywords between **AND** (&80) and **HIMEM** (&93), with the exception of **TAB**, and between **AUTO** (&C6) and the end of the table, with the exception of **PROC** (&F2). It will also add spaces around **TO** (&B8). We have made this selection in an attempt to compromise between a fast-running, compact utility routine and adding spaces where a programmer would probably want them, bearing in mind that, in some cases, adding an extra space would create a syntactically incorrect BASIC statement.

Spacer will also add spaces around colons and (except as mentioned below) equals signs.

It won't insert a space:
- at the very beginning of a line
- at the very end of a line
- after an asterisk at the beginning of a line (indicating a call to the Operating System)
- anywhere between opening and closing quotation marks
- to separate >= or <=, since > = or < = (with an intermediate space) causes a syntax error

- after a **DATA** statement
- after a **REM** statement
- if it would cause the line to exceed its maximum permitted length.

Since the utility is adding to the size of your program with the extra spaces, it will stop with a **'No room'** error if the program is about to exceed the amount of RAM available to it.

Please refer to 'Using the Programming Utilities' for notes on tacked-on bytes, embedded control characters and other general hints.

BBC BASIC KEYWORD TOKENS

80 AND	81 DIV	82 EOR	83 MOD
84 OR	85 ERROR	86 LINE	87 OFF
88 STEP	89 SPC	8A TAB(	8B ELSE
8C THEN	8D Line no	8E OPENIN[1]	8F PTR
90 PAGE	91 TIME	92 LOMEM	93 HIMEM
94 ABS	95 ACS	96 ADVAL	97 ASC
98 ASN	99 ATN	9A BGET	9B COS
9C COUNT	9D DEG	9E ERL	9F ERR
A0 EVAL	A1 EXP	A2 EXT	A3 FALSE
A4 FN	A5 GET	A6 INKEY	A7 INSTR(
A8 INT	A9 LEN	AA LN	AB LOG
AC NOT	AD OPENIN[2]	AE OPENOUT	AF PI
B0 POINT(	B1 POS	B2 RAD	B3 RND
B4 SGN	B5 SIN	B6 SQR	B7 TAN
B8 TO	B9 TRUE	BA USR	BB VAL
BC VPOS	BD CHR$	BE GET$	BF INKEY$
C0 LEFT$(	C1 MID$(	C2 RIGHT$(	C3 STR$
C4 STRING$(	C5 EOF	C6 AUTO	C7 DELETE
C8 LOAD	C9 LIST	CA NEW	CB OLD
CC RENUMBER	CD SAVE	CE	CF PTR
D0 PAGE	D1 TIME	D2 LOMEM[3]	D3 HIMEM[3]
D4 SOUND	D5 BPUT	D6 CALL	D7 CHAIN
D8 CLEAR	D9 CLOSE	DA CLG	DB CLS
DC DATA	DD DEF	DE DIM	DF DRAW
E0 END	E1 ENDPROC	E2 ENVELOPE	E3 FOR
E4 GOSUB	E5 GOTO	E6 GCOL	E7 IF
E8 INPUT	E9 LET	EA LOCAL	EB MODE
EC MOVE	ED NEXT	EE ON	EF VDU
F0 PLOT	F1 PRINT	F2 PROC	F3 READ
F4 REM	F5 REPEAT	F6 REPORT	F7 RESTORE
F8 RETURN	F9 RUN	FA STOP	FB COLOUR
FC TRACE	FD UNTIL	FE WIDTH	FF OSCLI[1]

1 New BASIC only
2 OPENUP in new BASIC
3 Old BASIC only

```
 10 REM **** SPACER ****
 20 :
 30 REM (c) Ian Trackman 1982
 40 :
 50 DIM msg(2) : REM Text messages
 60 :
 70 REM Basic pointers
 80 lomem  = &0
 90 himem  = &6
100 vartop = &2
110 top    = &12
120 page   = &18
130 :
140 memloc = &70 : REM + &71
150 split  = &72 : REM + &73
160 newtop = &74 : REM + &75
170 source = &76 : REM + &77
180 destin = &78 : REM + &79
190 length = &7A
200 quoteflag = &7B
210 asave  = &7C
220 ysave  = &7D
230 offset = &7E
240 :
250 oswrch = &FFEE
260 osnewl = &FFE7
270 :
280 REM Constants
290 eol    = &0D
300 space  = ASC " "
310 quote  = &22 : REM "
320 colon  = ASC ":"
330 star   = ASC "*"
340 equals = ASC "="
350 maxsize = &7C : REM for Mode 7
360 abs    = &94
370 auto   = &C6
380 data   = &DC
390 proc   = &F2
400 rem    = &F4
```

```
410 tab      = &8A
420 to       = &B8
430 :
440 org = &E00
450 :
460 opt = 2
470 :
480 FOR I% = 0 TO opt STEP opt
490 P% = org
500 [
510 OPT I%
520 :
530  LDY #0
540 .msg1loop LDA msg(1),Y \ Mode 7
     and title
550  BEQ msg1done
560  JSR oswrch
570  INY
580  BNE msg1loop
590 :
600 .msg1done LDA #maxsize
610  STA himem+1 \ Himem under Mode 7
620  LDX #0
630  STX himem
640 :
650  LDA #1 \ Start at PAGE + 1
660  STA memloc
670  LDA page
680  STA memloc+1
690 :
700 .nextline LDY #0
710  LDA (memloc),Y
720  CMP #&FF \ End of program flag
730  BNE morelines
740  JMP finish
750 :
```

```
 760 .morelines INY
 770  INY
 780  LDA (memloc),Y
 790  STA length \ Offset to start of next line
 800  CMP #5 \ One byte line
 810  BEQ endline
 820 :
 830  INY
 840  LDA (memloc),Y \ First item in line
 850  CMP #star \ OS command ?
 860  BEQ endline
 870  DEY \ reset it
 880 :
 890 .clearquote LDA #0
 900  STA quoteflag
 910 :
 920 .nextbyte INY
 930  LDA (memloc),Y
 940  CMP #eol
 950  BEQ endline
 960 :
 970  BIT quoteflag \ Ignore colons in quotes
 980  BPL quote_test
 990 :
1000  CMP #quote
1010  BNE nextbyte \ Loop until closing quote
1020  BEQ clearquote
1030 :
1040 .quote_test CMP #quote
1050  BNE tokentest
1060 :
1070  DEC quoteflag \ Set flag
1080  BNE nextbyte
1090 :
```

```
1100 .tokentest CMP #&8D \ Line-number
      token. Mustn't unpack
1110  BNE tokens2
1120 :
1130  INY \ Next 3 bytes are encoded
      line number
1140  INY
1150  INY
1160  BNE nextbyte
1170 :
1180 .tokens2 CMP #&80 \ A token ?
1190  BCS openup
1200 :
1210  CMP #colon
1220  BEQ openup
1230 :
1240  CMP #equals
1250  BNE nextbyte
1260 :
1270 \ Don't split <= or >=
1280  DEY
1290  LDA (memloc),Y
1300  INY
1310  CMP # ASC "<"
1320  BEQ nextbyte
1330 :
1340  CMP # ASC ">"
1350  BEQ nextbyte
1360  LDA #equals
1370 :
1380 .openup STA asave
1390  STY ysave
1400  JSR unpack
1410  LDA asave
1420  LDY ysave
1430  CMP #data \ Can't space after data
      or rem
1440  BEQ endline
1450 :
```

```
1460  CMP #rem
1470  BNE nextbyte
1480 :
1490 .endline LDA memloc
1500  CLC
1510  ADC length
1520  STA memloc
1530  LDA memloc+1
1540  ADC #0
1550  STA memloc+1
1560  JMP nextline
1570 :
1580 .finish JSR osnewl
1590  JMP osnewl \ & exit to caller
1600 :
1610 :
1620 .unpack LDA top
1630  CLC
1640  ADC #2
1650  LDA top+1
1660  ADC #0
1670  CMP #maxsize \ Room to unpack ?
1680  BCC inmem
1690  JMP outmem
1700 :
1710 .inmem LDA #0
1720  STA offset
1730  DEY
1740  CPY #2 \ Was it first byte ?
1750  BEQ typetest
1760 :
1770  LDA (memloc),Y
1780  CMP #space
1790  BEQ typetest
1800 :
1810  INC offset
1820  INC length
1830 :
1840 .typetest INY
```

```
1850  INY \ Now one up
1860  LDA (memloc),Y
1870  CMP #eol
1880  BEQ movetest
1890 :
1900  CMP #space
1910  BEQ movetest
1920 :
1930  LDA asave
1940  CMP #tab
1950  BEQ movetest
1960 :
1970  CMP #abs
1980  BCC goodtype
1990 :
2000  CMP #proc
2010  BEQ movetest
2020 :
2030  CMP #to
2040  BEQ goodtype
2050 :
2060  CMP #auto
2070  BCC movetest
2080 :
2090 .goodtype LDA length
2100  CMP #&F0 \ Maximum line length
2110  BCS no_move \ Too big
2120 :
2130  INC offset
2140  INC length
2150 :
2160 .movetest LDA offset
2170  BNE go_move
2180 .no_move RTS \ No move needed
2190 :
2200 .go_move LDY #2
2210  LDA length
2220  STA (memloc),Y \ Over-write
      current length
2230 :
```

```
2240 \ Point split at token
2250  LDA memloc
2260  CLC
2270  ADC ysave
2280  STA split
2290  LDA memloc+1
2300  ADC #0
2310  STA split+1
2320 :
2330 \Set up move vectors
2340  LDA split
2350  STA source
2360  LDA top+1
2370  STA source+1
2380  LDA source
2390  CLC
2400  ADC offset
2410  STA destin
2420  LDA source+1
2430  ADC #0
2440  STA destin+1
2450 :
2460  LDA top
2470  SEC
2480  SBC split
2490  TAY \ Difference over exact &100s
2500  LDA top+1
2510  SEC
2520  SBC split+1
2530  TAX \ Pages to move
2540  TYA
2550  BEQ shift2
2560 :
2570 \ Move odd bytes
2580  INY
2590 .shift1 DEY
2600  LDA (source),Y
2610  STA (destin),Y
2620  CPY #0
```

```
2630  BNE shift1
2640 :
2650 .shift2 TXA
2660  BEQ shiftdone
2670 :
2680 \ Move full pages
2690 .shift3 DEC source+1
2700  DEC destin+1
2710 :
2720 .shift4 DEY
2730  LDA (source),Y
2740  STA (destin),Y
2750  CPY #0
2760  BNE shift4
2770  DEX
2780  BNE shift3
2790 :
2800 .shiftdone LDA #space
2810  LDY ysave
2820  CPY #3
2830  BEQ after
2840 :
2850  DEY
2860  CMP (memloc),Y
2870  BEQ after
2880 :
2890  INY
2900  STA (memloc),Y
2910  LDX offset
2920  CPX #1
2930  BEQ repoint
2940 :
2950 .after LDY offset
2960  STA (split),Y
2970  DEY
2980  LDA asave
2990  STA (split),Y
3000 :
```

```
3010 .repoint LDA top
3020  CLC
3030  ADC offset
3040  STA top
3050  STA lomem
3060  STA vartop
3070  LDA top+1
3080  ADC #0
3090  STA top+1
3100  STA lomem+1
3110  STA vartop+1
3120  INC ysave
3130  RTS
3140 :
3150 .outmem LDY #0
3160 .msg2loop LDA msg(2),Y \ Out of
      memory
3170  BEQ msg2done
3180  JSR oswrch
3190  INY
3200  BNE msg2loop
3210 :
3220 .msg2done PLA \ POP RTS
3230  PLA
3240  JMP finish \ Exit at once
3250 ]
3260 :
3270 PROCtext (1, CHR$22 + CHR$7 +
     CHR$31 + CHR$14 + CHR$12 +
     "Uncrunching ")
3280 PROCtext (2, CHR$10 + CHR$10 +
     CHR$13 + "No room" + CHR$7)
3290 :
3300 NEXT
3310 :
3320 END
3330 :
3340 :
```

```
3350 DEF PROCtext (N,A$)
3360 msg(N) = P%
3370 $msg(N) = A$
3380 P% = P% + LEN(A$) + 1
3390 P%?-1 = 0
3400 ENDPROC
```

SPACE REMOVER

CRUNCH is a machine-code utility for removing spaces from BASIC programs. It is the second of the three 'squeeze' utilities which will help to shorten a BASIC program. Use it after REMSTRP and before PACKER.

Get the utility into memory as described in 'Using the Programming Utilities'. It takes up exactly &100 bytes starting at location &E00. CRUNCHDISK is the disk version and loads at &1800.

When you invoke the utility, the screen will clear (to Mode 7) and the message **Crunching** will be displayed in the middle of the screen. After a short while - depending on the length of your program and the number of spaces in it - the normal prompt will return and the crunch will have been completed.

CRUNCH does not remove *all* of the spaces in a program. It will leave quoted strings - that is, anything between quotation marks - as originally entered. It will also not touch anything in a line following a **REM** or **DATA** statement.

Since spaces can be used as delimiters in ***FX** calls, CRUNCH will not remove spaces from any line which *starts* with an asterisk. Normally, asterisks elsewhere in the line mean 'multiply', so that
A = B * C
is correctly shortened to
A=B*C
However, this will cause problems with something like
IF A = 5 THEN *FX 15 0 (with a space between **15** and **0**)
which will be reduced to
IFA=5THEN*FX150
The solution is to use commas as delimiters in in-line ***FX** commands, like this
IF A = 5 THEN *FX 15,0

Two further checks are made in case your program contains an assembly language listing. Anything following a reverse oblique ('\') is considered to be an assembly language comment and anything beyond it to the end of the line is left uncrunched. Secondly, a space can be used as a delimiter between a symbolic address label and an opcode, as in:
.loop INX
The rule here is that a dot at the *start* of a line will prevent crunching. On the other hand, if you have a label as part of a multiple-statement line, as here:
BEQ skip : .add INC hibyte : etc. . . .
CRUNCH will create the compound variable name **addINC** and assembly will fail. The solution is keep all labels at the start of program lines.

Notice that in all of the above cases where we refer to the start of a line, we mean literally the first byte. One or more spaces before an asterisk or full stop will prevent the check from operating properly.

There is one other case where removing a space will cause difficulties. You can reserve a fixed number of bytes in memory with the special use of the **DIM** statement, as in **DIM A 25** or **DIM B% 30** (see page 237 of the User Guide for further details). If you use an integer variable, the percentage sign enables the interpreter to handle the crunched **DIMB%25**, but if you use a real variable, as in **DIMA25**, **A25** will be treated as a compound variable name and will generate an error when you run your program. You must either use integer variable names or reinstate the space after using CRUNCH.

You may be aware that you can omit **THEN** after **IF**, as in
IF A = B C = 5
meaning
IF A = B THEN C = 5
provided that you leave a space between the variable names **B** and **C**. If you put such a statement through CRUNCH, the space will be removed and the result will become
IFA=BC=5
leading to a syntax error. Since, when tokenised, both the space and **THEN** take up one byte each, there seems to be no good reason to leave out the **THEN** when writing programs except, of course, to save work for lazy typists and to make the program code less readable!

One final point arises if you subsequently want to edit your program. Normally, when you add lines to your program or edit them, you have to make sure that you insert spaces wherever there is the possibility of ambiguity, particularly after a variable name. For example, you cannot type
IF A = BTHEN PRINT
since the line editor treats **BTHEN** as the name of a variable. (You must type a space between **B** and **THEN**.) As CRUNCH is working on a program which has already been tokenised, it will produce lines of code without spaces in them such as
IFA=BTHENPRINT
which will be interpreted correctly and will not cause any errors. However, if you edit such a line by copying over it, you must re-introduce the spaces that you would have used in the first place.

```
  10 REM **** CRUNCHER ****
  20 :
  30 REM (c) Ian Trackman 1982
  40 :
  50 REM Packs in-line assembler '.'s -
     causes assembly failure. Not worth
     checking time
  60 :
  70 DIM msg(1)
  80 :
  90 REM Basic pointers
 100 lomem   = &0
 110 vartop  = &2
 120 top     = &12
 130 page    = &18
 140 :
 150 memloc  = &72 : REM + &73
 160 split   = &74 : REM + &75
 170 source  = &7A : REM + &7B
 180 destin  = &7C : REM + &7D
 190 length  = &80
 200 quoteflag = &83
 210 count   = &84
 220 ysave   = &85
 230 :
 240 oswrch  = &FFEE
 250 osnewl  = &FFE7
 260 :
 270 REM Constants
 280 eol    = &0D
 290 space  = ASC " "
 300 quote  = &22 : REM "
 310 colon  = ASC ":"
 320 star   = ASC "*"
 330 slash  = ASC "\"
 340 dot    = ASC "."
 350 maxsize = &7C : REM for Mode 7
 360 rem    = &F4
 370 data   = &DC
 380 :
```

```
390 org = &E00
400 :
410 opt = 2
420 :
430 FOR I% = 0 TO opt STEP opt
440 P% = org
450 [
460 OPT I%
470 :
480  LDY #0
490 .msg1loop LDA msg(1),Y \ Mode 7
     and title
500  BEQ msg1done
510  JSR oswrch
520  INY
530  BNE msg1loop
540 :
550 .msg1done LDA #1 \ Start at PAGE +
     1
560  STA memloc
570  LDA page
580  STA memloc+1
590 :
600 .nextline LDY #0
610  LDA (memloc),Y
620  CMP #&FF \ End of program flag
630  BNE morelines
640  JMP finish
650 :
660 .morelines INY
670  INY
680  LDA (memloc),Y
690  STA length \ Offset to start of
     next line
700  CMP #5 \ One byte line
710  BEQ endline
720 :
```

```
 730  INY
 740  LDA (memloc),Y
 750  CMP #star
 760  BEQ endline \ Opening '*' don't crunch
 770 :
 780  CMP #dot
 790  BEQ endline \ Opening Assembler '.' don't crunch
 800  DEY
 810 :
 820 .clearquote LDA #0
 830  STA quoteflag
 840 :
 850 .nextbyte INY
 860  LDA (memloc),Y
 870  CMP #eol
 880  BEQ endline
 890 :
 900  BIT quoteflag \ Ignore colons in quotes
 910  BPL quote_test
 920 :
 930  CMP #quote
 940  BNE nextbyte \ Loop until closing quote
 950  BEQ clearquote
 960 :
 970 .quote_test CMP #quote
 980  BNE tokentest
 990 :
1000  DEC quoteflag \ Set flag
1010  BNE nextbyte
1020 :
1030 .tokentest CMP #&8D \ Line-number token. Mustn't examine
1040  BNE test2
1050 :
```

```
1060  INY \ Next 3 bytes are encoded
      line number
1070  INY
1080  INY
1090  BNE nextbyte \ Always
1100 :
1110 .test2 CMP #rem \ Can't crunch
      after rem, data or \
1120  BEQ endline
1130 :
1140  CMP #data
1150  BEQ endline
1160 :
1170  CMP #slash
1180  BEQ endline
1190 :
1200  CMP #space
1210  BNE nextbyte
1220 :
1230  STY ysave
1240  JSR crunch
1250  DEC ysave
1260  LDY ysave
1270  BNE nextbyte \ Always
1280 :
1290 .endline LDA memloc
1300  CLC
1310  ADC length
1320  STA memloc
1330  LDA memloc+1
1340  ADC #0
1350  STA memloc+1
1360  JMP nextline
1370 :
1380 .finish JSR osnewl
1390  JMP osnewl \ & exit to caller
1400 :
1410 :
```

```
1420 .crunch LDA memloc \ Point destin
      at space
1430  CLC
1440  ADC ysave
1450  STA destin
1460  LDA memloc+1
1470  ADC #0
1480  STA destin+1
1490 :
1500 \ Set up move vectors
1510  LDA destin
1520  CLC
1530  ADC #1
1540  STA source
1550  LDA destin+1
1560  ADC #0
1570  STA source+1
1580 :
1590  LDA top
1600  SEC
1610  SBC source
1620  STA count
1630  LDA top+1
1640  SEC
1650  SBC source+1
1660  LDY #0
1670  TAX \ Pages to move
1680  BEQ shift2
1690 :
1700 .shift1 LDA (source),Y
1710  STA (destin),Y
1720  INY
1730  BNE shift1
1740  INC source+1
1750  INC destin+1
1760  DEX
1770  BNE shift1
1780 :
```

```
1790 .shift2 LDX count \ Move odd bytes
1800  BEQ shiftdone
1810 :
1820 .shift3 LDA (source),Y
1830  STA (destin),Y
1840  INY
1850  DEX
1860  BNE shift3
1870 :
1880 .shiftdone LDY #2
1890  DEC length
1900  LDA length
1910  STA (memloc),Y \ Over-write current length
1920  LDA top
1930  SEC
1940  SBC #1
1950  STA top
1960  STA lomem
1970  STA vartop
1980  LDA top+1
1990  SBC #0
2000  STA top+1
2010  STA lomem+1
2020  STA vartop+1
2030  RTS
2040 ]
2050 :
2060 PROCtext (1, CHR$22 + CHR$7 + CHR$31 + CHR$15 + CHR$12 + "Crunching ")
2070 :
2080 NEXT
2090 :
2100 END
2110 :
2120 :
```

```
2130 DEF PROCtext (N,A$)
2140 msg(N) = P%
2150 $msg(N) = A$
2160 P% = P% + LEN(A$) + 1
2170 P%?-1 = 0
2180 ENDPROC
```

SPEECH CHIP NUMBER GENERATOR

SPEAK, written in BASIC, is a routine for converting numerical values into words o be 'spoken' by the BBC Microcomputer Voice Chip.

The main loop simply repeats, collecting keyboard input and passing it to **PROCspeak**, where all the work is done.

The procedure first tests whether it has, in fact, been given a number. If the parameter is invalid, it will say 'zero'.

We next test for range. Since the word 'million' is not available on the Chip, any number above 999,999 or below −999,999 will trigger the response **Number too large**.

Any minus sign is recognised and any numbers following a decimal point are separated out.

If the number is greater than 999 (or less than −999), the procedure recursively calls itself in order to handle repeated phrases. Because of the recursion, it is important that D$ is declared to be local, so that previous values are saved.

The procedure then deals with smaller integers, with special attention to the numbers between 10 and 20.

Finally, if we have a floating point number, the digits following the decimal point are pronounced, although rounding will occur if the number cannot be stored with sufficient accuracy as a real variable.

```
 10 REM **** SPEECH CHIP DRIVER ****
 20 :
 30 REM (c) Ian Trackman 1982
 40 :
 50 REPEAT
 60   INPUT A
 70   PROCspeak (A)
 80   UNTIL FALSE
 90 :
100 END
110 :
120 :
```

```
130 DEF PROCspeak (N)
140 IF N = 0 THEN SOUND -1,ASC"0",0,0 :
    ENDPROC : REM Zero or invalid
150 IF ABS N >= 1E6 THEN SOUND
    -1,229,0,0 : SOUND -1,50,0,0 :
    SOUND -1,214,0,0 : ENDPROC : REM
    "Number too large"
160 :
170 LOCAL D$
180 :
190 IF N < 0 THEN SOUND -1,219,0,0 : N
    = ABS N : REM Minus
200 :
210 D$ = STR$(N + 1) : REM Prevent E
    format if 1 < .1
220 D$ = MID$(D$,INSTR(D$,".")) : REM
    Save decimal part
230 :
240 IF N > 999 THEN PROCspeak (N DIV
    1000) : SOUND -1,141,0,0 : IF N MOD
    1000 AND N MOD 1000 < 100 THEN
    SOUND -1,165,0,0
250 :
260 N = N MOD 1000
270 IF N < 1000 AND N > 99 THEN SOUND
    -1,ASC STR$(N DIV 100),0,0 : SOUND
    -1,140,0,0 : IF N MOD 100 THEN
    SOUND -1,165,0,0
280 :
290 N = N MOD 100
300 IF N > 19 THEN SOUND -1,140 + 2 *
    (N DIV 10),0,0 : SOUND -1,137,0,0
310 IF N < 20 AND N > 12 THEN SOUND
    -1,140 + 2 * (N MOD 10),0,0 : SOUND
    -1,135,0,0
320 IF N = 12 THEN SOUND -1,273,0,0
330 IF N = 11 THEN SOUND -1,190,0,0
340 IF N = 10 THEN SOUND -1,264,0,0
350 :
```

```
360 IF N > 20 OR N < 10 THEN N = N MOD
    10 : IF N THEN SOUND -1,ASC STR$
    N,0,0
370 :
380 IF INSTR(D$,".") = 0 THEN ENDPROC
390 :
400 FOR N = 1 TO LEN D$
410   SOUND -1,ASC MID$(D$,N,1),0,0
420   NEXT
430 :
440 ENDPROC
```

UNPACKER

UNPACK is a utility, written in machine-code, which splits up multi-statement BASIC program lines into single statements on separate lines, so making the program easier to read, edit and debug. To a large extent, it reverses the effect of the PACKER utility.

There are two versions of the program on the tape. UNPACK is the version for use with tape-based computers and resides between &E00 and &10FF. UNPACKDISK is for use with disks and is loaded between &1600 and &18FF. Please refer to 'Using the Programming Utilities' for installation instructions. Other than the addresses at which they start, the two programs operate identically.

As the utility is co-resident you can load it before or after you get your BASIC program into memory. Once the utility is in memory, start it working with **CALL &E00** (tape version) or **CALL &1600** (disk version).

UNPACK will create a new single-statement line for every statement in a multiple-statement line, except for statements following an **IF** or **ON ERROR** command. Essentially, it is looking for colons, although it will ignore colons in a string between quotation marks, a colon in a **REM** or **DATA** statement and a colon following an asterisk at the start of a line (indicating a call to the Operating System). It will also not unpack after a reverse oblique ('\'), which indicates the start of a comment in an assembly language program.

The additional program lines will need new line-numbers and these are created in increments of one, following the multi-statement line. For example:

100 CLS : INPUT X : A=B+X

will become:

100 CLS
101 INPUT X
102 A=B+X

This means that there must be a large enough gap in the line-numbering sequence before the next line to allow sufficient new numbers to be created.

The routine begins by testing the line-number spacing. If it finds that there are too many statements to fit in, it will exit with the message **Renumbering needed at line number:—** followed by a list of the one or more line-numbers at which the problem lies. Cure it by renumbering the program with a larger increment - the second parameter in the RENUMBER command.

Once UNPACK is satisfied that it can work properly, it will display the message **Unpacking** and start to work. After a short while - just how long depends on the length of your program and how many statements need to be unpacked – the prompt will return, leaving the unpacked BASIC program in memory.

Whilst unpacking, the utility will also remove any spaces on either side of a statement-separating colon, since they will no longer be needed.

There are two situations which will stop UNPACK from completing its task. The first is when the program, which is being expanded by the creation of new line-numbers, runs out of memory in which to grow. The second case is if you try to make it unpack a multi-statement line numbered 32767, since BASIC will not accept a line-number greater than 32767. In both cases, UNPACK will stop with a **No room** error.

Please refer to 'Using the Programming Utilities' for notes on tacked-on bytes, embedded control characters and other general hints.

```
   10 REM **** UNPACKER ****
   20 :
   30 REM (c) Ian Trackman 1982
   40 :
   50 DIM msg(3) : REM Text messages
   60 :
   70 REM Basic pointers
   80 lomem  = &0
   90 himem  = &6
  100 vartop = &2
  110 top    = &12
  120 page   = &18
  130 :
  140 linenum = &70 : REM + &71
  150 memloc  = &72 : REM + &73
  160 gap     = &74 : REM + &75
  170 mod10   = &76 : REM + &77
  180 newtop  = &78 : REM + &79
  190 source  = &7A : REM + &7B
  200 destin  = &7C : REM + &7D
  210 chars      = &7E
  220 renumflag  = &7F
  230 length     = &80
  240 colon_count  = &81
  250 if_flag    = &82
  260 quoteflag  = &83
  270 pass1      = &84
```

```
280 newlength = &85
290 offset    = &86
300 :
310 oswrch = &FFEE
320 osnewl = &FFE7
330 :
340 REM Constants
350 numsize = 8
360 eol      = &0D
370 space    = ASC " "
380 quote    = &22 : REM "
390 colon    = ASC ":"
400 star     = ASC "*"
410 slash    = ASC "\"
420 maxsize = &7C : REM for Mode 7
430 error    = &85 : REM "ERROR" token
440 if       = &E7
450 rem      = &F4
460 data     = &DC
470 :
480 org = &E00
490 :
500 opt = 2
510 :
520 FOR I% = 0 TO opt STEP opt
530 P% = org
540 [
550 OPT I%
560 :
570  LDA #&16 \ Set MODE 7
580  JSR oswrch
590  LDA #7
600  JSR oswrch
610 :
620  LDA #maxsize
630  STA himem+1 \ Himem under Mode 7
640  LDX #0
650  STX renumflag \ Clear it
660  STX himem
670  DEX
```

```
 680  STX pass1 \ Set "pass number"
      index
 690 :
 700 .mainloop LDA #1 \ Start at PAGE +
      1
 710  STA memloc
 720  LDA page
 730  STA memloc+1
 740 :
 750 .nextline LDY #0
 760  LDA (memloc),Y
 770  CMP #&FF \ End of program flag
 780  BNE morelines
 790  JMP endprog
 800 :
 810 .morelines STA linenum+1 \ Hi-lo

 820  INY
 830  LDA (memloc),Y
 840  STA linenum
 850  INY
 860  LDA (memloc),Y
 870  STA length \ Offset to start of
      next line
 880 :
 890  CMP #5
 900  BEQ endline
 910 :
 920  LDA #0
 930  STA colon_count \ Clear them
 940  STA if_flag
 950  INY
 960  LDA (memloc),Y \ First item in
      line
 970  CMP #star \ OS command ?
 980  BEQ endline
 990  DEY \ reset it
1000 :
```

```
1010 .clearquote LDA #0
1020  STA quoteflag
1030 :
1040 .nextbyte INY
1050  LDA (memloc),Y
1060  CMP #eol
1070  BEQ endline
1080 :
1090  BIT quoteflag \ Ignore colons in quotes
1100  BPL iftest
1110 :
1120  CMP #quote
1130  BNE nextbyte \ Loop until closing quote
1140  BEQ clearquote
1150 :
1160 .iftest CMP #if
1170  BEQ endline
1180 :
1190  CMP #rem
1200  BEQ endline
1210 :
1220  CMP #error
1230  BEQ endline
1240 :
1250  CMP #slash
1260  BEQ endline
1270 :
1280  CMP #data
1290  BEQ endline
1300 :
1310  CMP #&8D \ Line number follows
1320  BNE quotetest
1330 :
1340  INY
1350  INY
1360  INY
1370 :
```

```
1380 .quotetest CMP #quote
1390  BNE colontest
1400 :
1410  DEC quoteflag \ Set flag
1420  BNE nextbyte
1430 :
1440 .colontest CMP #colon
1450  BNE nextbyte
1460 :
1470  INC colon_count
1480  BIT pass1
1490  BMI nextbyte
1500 :
1510  CPY #3 \ Pass 2 only
1520  BEQ nextbyte \ Don't unpack
      opening colon
1530 :
1540  JSR unpack
1550  JMP nextline
1560 :
1570 .endline LDA memloc
1580  CLC
1590  ADC length
1600  STA memloc
1610  LDA memloc+1
1620  ADC #0
1630  STA memloc+1
1640  BIT pass1
1650  BPL nextjump \ i.e. on pass 2
1660 :
1670  LDA colon_count
1680  BEQ nextjump
1690 :
1700  CPY #4 \ Single colon line
1710  BEQ nextjump
1720 :
1730  JSR roomtest
1740 .nextjump JMP nextline
1750 :
```

```
1760 .endprog BIT renumflag
1770  BMI finish \ If set
1780 :
1790  INC pass1 \ To 0 or 1
1800  BNE finish \ 1 means second pass done
1810 :
1820  LDY #0
1830 .msg2loop LDA msg(2),Y \ Unpacking
1840  BEQ msg2done
1850  JSR oswrch
1860  INY
1870  BNE msg2loop
1880 .msg2done JMP mainloop
1890 :
1900 .finish JSR osnewl
1910  JMP osnewl \ & exit to caller
1920 :
1930 :
1940 .roomtest LDY #0
1950  LDA (memloc),Y
1960  CMP #&FF \ End of program ?
1970  BEQ test32767
1980 :
1990  LDA linenum
2000  CLC
2010  ADC colon_count
2020  STA gap
2030  LDA linenum+1
2040  ADC #0
2050  STA gap+1
2060  CMP (memloc),Y / Next linenum hi
2070  BCC test32767
2080 :
2090  LDA gap
2100  INY
2110  CMP (memloc),Y / Next linenum lo
2120  BCS no_room
2130 :
```

```
2140 \ Can't unpack beyond 32767
2150 .test32767 LDA linenum+1
2160  CMP #&7F
2170  BNE jumpback
2180  LDA linenum
2190  CMP #&FF
2200  BNE jumpback
2210 :
2220 .no_room BIT renumflag
2230  BMI listit
2240  DEC renumflag \ Set it
2250 :
2260  LDY #0
2270 .msg1loop LDA msg(1),Y \ Renumber
2280  BEQ listit
2290  JSR oswrch
2300  INY
2310  BNE msg1loop
2320 :
2330 .listit JSR number
2340 .jumpback RTS
2350 :
2360 :
2370 \ Convert 2-byte hex line-number
      to decimal ASCII
2380 \ value holds value Div 10
2390 \ mod10 holds value Mod 10
2400 :
2410 .number LDX #numsize
2420  STX chars \ ASCII digit counter
2430 :
2440 .convert DEC chars
2450  LDA #0
2460  STA mod10
2470  STA mod10+1
2480  LDX #&10 \ Double byte
2490  CLC
2500 :
```

```
2510 .divloop ROL linenum \ Bit 0 (carry) becomes quotient
2520  ROL linenum+1
2530  ROL mod10
2540  ROL mod10+1
2550 :
2560  LDA mod10
2570  SEC
2580  SBC #10
2590  TAY \ Low byte
2600  LDA mod10+1
2610  SBC #0
2620  BCC deccount \ if dividend < divisor
2630 :
2640  STY mod10 \ Next bit of dividend = 1.
2650  STA mod10+1 \ Dividend = Dividend - divisor
2660 :
2670 .deccount DEX
2680  BNE divloop
2690 :
2700  ROL linenum \ Shift in last carry for quotient
2710  ROL linenum+1
2720 :
2730  LDA mod10
2740  ORA #ASC "0" \ ASCII mask
2750  PHA \ Stack it (starts at right-hand digit)
2760 :
2770  LDA linenum \ Continue if value <> 0
2780  ORA linenum+1
2790  BNE convert
2800 :
```

```
2810  LDX chars
2820  BEQ outnum1 \ Only possible if
      numsize = 5
2830 :
2840 \ Pad to right-justify
2850  LDA #ASC " "
2860 .blank PHA
2870  DEX
2880  BNE blank
2890 :
2900 .outnum1 LDX #numsize
2910 .outnum2 PLA \ Unstack ASCII
2920  JSR oswrch
2930  DEX
2940  BNE outnum2
2950  RTS
2960 :
2970 :
2980 .unpack LDA top
2990  CLC
3000  ADC #3
3010  STA newtop
3020  LDA top+1
3030  ADC #0
3040  CMP #maxsize \ Room to unpack ?
3050  BCC inmem
3060  JMP outmem
3070 :
3080 .inmem STA newtop+1
3090  LDA #3
3100  STA offset
3110 :
3120 \ Spaces around colon ?
3130  INY
3140  LDA (memloc),Y
3150  CMP #space
3160  BNE back
3170  DEC offset
3180  DEC length
3190 :
```

```
3200 .back DEY
3210  DEY
3220  LDA (memloc),Y
3230  CMP #space
3240  BNE forward
3250  DEC offset
3260  DEC length
3270  DEY \ Obliterate the space
3280 :
3290 .forward INY
3300  LDA #eol
3310  STA (memloc),Y \ In last byte of line
3320  INY
3330  STY newlength \ Point to next (new) line start
3340  LDA length
3350  SEC
3360  SBC newlength
3370  CLC
3380  ADC #3
3390  STA length \ New line's length
3400  LDY #2
3410  LDA newlength
3420  STA (memloc),Y \ Over-write current length
3430 :
3440 \ Point memloc at new line start
3450  LDA memloc
3460  CLC
3470  ADC newlength
3480  STA memloc
3490  LDA memloc+1
3500  ADC #0
3510  STA memloc+1
3520 :
3530 \Set up move vectors
3540  LDA memloc
3550  STA source
3560  LDA top+1
```

```
3570  STA source+1
3580  LDA source
3590  CLC
3600  ADC offset \ 3, 2 or 1
3610  STA destin
3620  LDA source+1
3630  ADC #0
3640  STA destin+1
3650 :
3660  LDA top
3670  SEC
3680  SBC memloc
3690  TAY \ Difference over exact &100s
3700  LDA top+1
3710  SEC
3720  SBC memloc+1
3730  TAX \ Pages to move
3740  TYA
3750  BEQ shift2
3760 :
3770 \ Move odd bytes
3780  INY
3790 .shift1 DEY
3800  LDA (source),Y
3810  STA (destin),Y
3820  CPY #0
3830  BNE shift1
3840 :
3850 .shift2 TXA
3860  BEQ shiftdone
3870 :
3880 \ Move full pages
3890 .shift3 DEC source+1
3900  DEC destin+1
3910 :
3920 .shift4 DEY
3930  LDA (source),Y
3940  STA (destin),Y
3950  CPY #0
3960  BNE shift4
```

```
3970  DEX
3980  BNE shift3
3990 :
4000 \ New line's info.
4010 .shiftdone LDY #2
4020  LDA length
4030  STA (memloc),Y
4040  DEY
4050  LDA linenum
4060  CLC
4070  ADC #1
4080  STA (memloc),Y
4090  DEY
4100  LDA linenum+1
4110  ADC #0
4120  STA (memloc),Y
4130 :
4140  LDA newtop
4150  STA top
4160  STA lomem
4170  STA vartop
4180  LDA newtop+1
4190  STA top+1
4200  STA lomem+1
4210  STA vartop+1
4220  RTS
4230 :
4240 .outmem LDY #0
4250 .msg3loop LDA msg(3),Y \ Out of
      memory
4260  BEQ msg3done
4270  JSR oswrch
4280  INY
4290  BNE msg3loop
4300 :
4310 .msg3done PLA \ POP RTS
4320  PLA
4330  JMP finish \ Exit at once
4340 ]
4350 :
```

```
4360 PROCtext (1,"Renumbering needed at
     line number :-" + CHR$10 + CHR$10 +
     CHR$13)
4370 PROCtext (2,CHR$31 + CHR$14 +
     CHR$12 + "Unpacking ")
4380 PROCtext (3,CHR$10 + CHR$10 +
     CHR$13 + "No room" + CHR$7)
4390 :
4400 NEXT
4410 :
4420 END
4430 :
4440 :
4450 DEF PROCtext (N,A$)
4460 msg(N) = P%
4470 $msg(N) = A$
4480 P% = P% + LEN(A$) + 1
4490 P%?-1 = 0
4500 ENDPROC
```

VARIABLE NAMES DUMP

When you have written a reasonably long program or if you are reading through someone else's program, it is often a good idea to make a print-out of a list of the names of all the variables used in the program. You can then add notes as to how the variables are used. It can also help to ensure that you don't accidentally re-use the name of an existing variable and so over-write its value when the program runs.

VARDUMP is a machine-code utility which resides at &E00 (tape) or &1800 (disk) and takes up less than &100 bytes. The disk version is VARDMPDSK. Please refer to 'Using the Programming Utilities' for loading instructions. Having loaded VARDUMP into memory, use it with **CALL &E00** (tape) or **CALL &1800** (disk).

You can either install the utility before you load your BASIC program with a ***LOAD** command or you can use it directly from tape or disk with a ***RUN** command after loading or running the BASIC program. (Disk users can type ***VARDUMP**.) Provided that you don't over-write it, the utility will remain in memory whilst you load and save BASIC programs.

VARDUMP prints out a list of all the names of the variables created by a BASIC program. It does not print out the names of the 'system variables' (**A%** to **Z%**), since these remain in memory at all times whilst the computer is on and it is not possible to tell, at least with this utility, whether the BASIC program makes use of system variables.

Other integer variables are suffixed with a **%** sign. String variables are suffixed with a **$** sign. Arrays, whether real, integer or string, are followed by the size of the element(s) with which they have been dimensioned.

Notice that we particularly referred to variables *created* by the BASIC program. If the program does not execute a statement which creates a new variable, its name will not be included in the print-out. For example, if the program contains a test which is always passed, you will never see a variable which is used only within a procedure called when the test fails. A similar situation will arise if you start the program but interrupt it before it has generated all of its variables.

If you want to search your program to see whether a particular variable is *referred to* in it, for example a system variable, use the XREF utility in the Toolbox. You can also use XREF to produce a list of lines in which any particular variable occurs in the program.

Remember also that variables are destroyed if you alter or renumber the program, type **CLEAR,** or press Break.

To send a list to your printer, simply preface the **CALL** with a **VDU 2** command or a Control B.

The program works by referencing an index table which starts at location &482. The table contains the link addresses of further details of the variables themselves. Since the table is only ordered to the extent of the first character of a variable's name, the list produced by VARDUMP will not necessarily be in true alphabetical order.

The utility contains a useful subroutine called 'convert', which converts a hexadecimal number to decimal and then prints it out in ASCII.

```
  10 REM **** VARIABLE NAMES DUMP ****
  20 :
  30 REM (c) Ian Trackman 1982
  40 :
  50 REM Does not list system variables
     A% to Z%
  60 :
  70 *KEY 1 CALL &1800|M
  80 :
  90 vector = &70 : REM + &71
 100 temp   = &72 : REM + &73
 110 number = &74 : REM + &75
 120 mod10  = &76 : REM + &77
 130 count  = &80
 140 dimens = &81
 150 ysave  = &82
 160 :
 170 base = &482
 180 :
 190 oswrch = &FFEE
 200 osnewl = &FFE7
 210 :
 220 org = &E00
 230 :
 240 opt = 2
 250 :
 260 FOR I% = 0 TO opt STEP opt
 270 P% = org
```

```
280 [
290 OPT I%
300 :
310  LDX #0
320  STX count \ Zero it
330 :
340 .nextletter LDA base,X \ lsb of
     next variable start letter
350  STA vector
360  INX
370  LDA base,X \ msb of letter
380  BEQ endloop \ Zero if no names
390 :
400  STA vector+1
410 :
420 .inloop LDY #0
430  LDA (vector),Y
440  STA temp \ Keep our place
450  INY
460  LDA (vector),Y
470  STA temp+1
480  JSR print
490  LDA temp \ Restore place before
     print-out
500  STA vector
510  LDA temp+1
520  STA vector+1
530  BNE inloop \ Always
540 :
550 .endloop INC count \ Move on two
     bytes
560  INC count
570  LDX count \ 58 2-byte variable
     names
580  CPX #&74 \ (A - Z, a - z incl.
     intermediate ASCII)
590  BNE nextletter \ More ?
600 :
```

```
610  RTS \ All done, so exit
620 :
630 :
640 \ Print-out subroutine
650 :
660 \ Convert count to ASCII letter
670 .print LDA count
680  LSR A \ DIV 2
690  CLC
700  ADC #&41 \ Add conversion offset
710 :
720 .name JSR oswrch \ Print it
730  INY
740  LDA (vector),Y \ Next character
750  CMP #ASC "0" \ Still alpha-numeric
     part of name ?
760  BCS name \ If so, continue
     printing
770 :
780  CMP #0 \ End of variable ?
790  BEQ endname \ Exit subroutine
800 :
810  CMP #ASC "(" \ Start of array info
     ?
820  BNE name \ If not, must be % or $,
     so print it
830 :
840 \ Print dimensions of array
850  JSR oswrch \ Print opening
     bracket
860  INY
870  INY
880  LDA (vector),Y
890  STA dimens \ Encoded no. of
     elements
900  DEC dimens \ Decode to actual
     number
910  LSR dimens
920 :
```

```
 930 \ Print dimension of each element
 940 .element INY
 950  LDA (vector),Y
 960  SEC
 970  SBC #1 \ Element includes 0
 980  STA number \ lsb
 990  INY
1000  LDA (vector),Y
1010  SBC #0 \ Borrow from -1 ?
1020  STA number+1 \ msb
1030  STY ysave \ Keep our place
1040  LDA #0 \ End of string flag
1050  PHA
1060 :
1070 \ Convert to decimal ASCII
1080 .convert LDA #0
1090  STA mod10 \ Zero them
1100  STA mod10+1
1110  LDX #&10 \ Double byte (16 bits)
1120  CLC
1130 :
1140 .divloop ROL number \ Bit 0 (carry) becomes quotient
1150  ROL number+1
1160  ROL mod10
1170  ROL mod10+1
1180 :
1190  LDA mod10
1200  SEC
1210  SBC #10 \ Division by subtracting tens
1220  TAY \ Save low byte
1230  LDA mod10+1
1240  SBC #0
1250  BCC deccount \ If dividend < divisor
1260 :
```

```
1270  STY mod10 \ Next bit of dividend = 1
1280  STA mod10+1 \ Dividend = dividend - divisor
1290 :
1300 .deccount DEX
1310  BNE divloop \ Done 16 bits ?
1320 :
1330  ROL number \ Shift in last carry for quotient
1340  ROL number+1
1350 :
1360  LDA mod10
1370  ORA #ASC "0" \ ASCII mask
1380  PHA \ Stack it (starts at right-hand digit)
1390 :
1400  LDA number
1410  ORA number+1
1420  BNE convert \ Not two zeros
1430 :
1440 .outnum PLA \ Unstack ASCII
1450  BEQ numdone \ End of string ?
1460  JSR oswrch
1470  JMP outnum \ Do some more
1480 :
1490 .numdone DEC dimens
1500  BEQ close \ All elements printed ?
1510 :
1520  LDA #ASC"," \ Separate next element with comma
1530  JSR oswrch
1540  LDY ysave \ Restore position
1550  JMP element \ Do next element
1560 :
1570 .close LDA #ASC ")" \ End of array
1580  JSR oswrch
1590 :
```

```
1600 .endname JMP osnewl \ CRLF and
      exit subroutine
1610 ]
1620 :
1630 NEXT
1640 :
1650 END
```


Printed in England by Popper & Company Ltd, Welwyn Garden City, Hertfordshire