commodore 64
machine code master

A library of machine code routines

david lawrence & mark england

commodore 64
machine code master

A library of machine code routines

First published 1983 by:
Sunshine Books (an imprint of Scot Press Ltd.)
Hobhouse Court,
19 Whitcomb Street,
London WC2 7HF

Copyright © David Lawrence and Mark England
Reprinted December 1983

ISBN 0 946408 05 X

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording and/or otherwise, without the prior written permission of the Publishers.

Cover design by Graphic Design Ltd.
Illustration by Stuart Hughes.
Typeset and printed in England by Commercial Colour Press, London E7.

CONTENTS

Contents in detail

CHAPTER 7
The BASIC Extender II
The final changes to link extended BASIC to the existing interpreter.

CHAPTER 8
Simple BASIC Action Keywords
Undead, Subex and Rkill.

CHAPTER 9
The Problem of Parameters
Picking up parameters from a BASIC program. Keywords: DOKE, PLOT, DELETE, BSAVE, BLOAD, BVERIFY, MOVE, FILL and RESTORE.

CHAPTER 10
BASIC Functions
Extending the 64's mathematical functions. Keywords: DEEK, VARPTR, YPOS.

CHAPTER 11
Breaking New Frontiers
The FAST command that wasn't.

APPENDICES:
A) Checksum Generator: test your Mastercode Program as you enter it.
B) A User Guide to the Mastercode program.
C) A complete table of the variables found in the Mastercode program and their uses.
D) A quick guide to the purpose of each subroutine in the Mastercode program.
E) The ROM routines called upon by extended BASIC and what they do.
F) Representation of control characters in the Mastercode program.

FOREWORD

This is *not* yet another machine code book for a popular micro which spends half its pages explaining all the 6502/6510 machine code instructions and their various addressing modes, with a crude loader program and a number of machine code routines of dubious usefulness tacked onto the end. The intention of this book is to provide its readers with a solid BASIC program for the entry of machine code and assembly language routines and, along with it, a selection of machine code programs that are worth having.

The BASIC program is called Mastercode and it is just about a complete machine code programming tool, containing a Monitor to allow you to examine and change the contents of memory, a Disassembler which translates machine code programs into assembly language format, and a File Editor and Assembler which together allow assembly language programs to be developed and transformed into machine code.

For the machine code routines in the second half of the book we have adopted a new approach. What you will find there is a collection of routines which you can use to extend the BASIC language on your 64 with 14 new commands. Apart from increasing the power of BASIC it will also help you to learn the techniques of effective machine code programming and the ways in which the machine code programmer can make use of the routines already built into the 64's BASIC Interpreter.

The book is not intended to be a primer in machine code. That is not to say that it cannot be used by beginners. It is simply that we have assumed that you will possess, or can get hold of, some other book which explains each 6502/6510 instruction in detail. We decided that we could offer better value by concentrating on the programs themselves and on explaining the techniques that went into them. If you have to look up the odd instruction in your 6502 text-book it will be a small price to pay.

All the programs in the book have been tested — in fact the Mastercode program itself has been tested to exhaustion (ours). The machine code routines in this book have all been developed on the Mastercode program because only in that way could we be sure that you would be able to do the same. If you run into bugs in anything you find in this book then the blame is on us but we venture to suggest that they will not be major after the testing that has been put in.

The book is a cooperative venture between two people with very different backgrounds. David Lawrence writes mainly BASIC programs and

is the author of several books on microcomputing. His interest in computer hardware is minimal. Mark England studies electronic engineering, juggles with silicon chips as if it were second nature and learned to write machine code when the BASIC ROM on his micro burned out. He discovered he didn't need it anyway.

The idea came from David Lawrence, who was tired of buying machine code books that never seemed to provide anything of interest when he actually worked through them. Mark England did the vast majority of the programming, though not without heckling. The final form of the book arises out of Mark England's need to explain to his co-author just what the programs were about and have it translated for lesser mortals. In that process of explanation and discussion a style of commentary and explanation has developed which, we believe, does justice to the programs and to the reader's need to understand them.

The other partner in the writing of this book has been the Commodore 64. The book would not have been a possibility on many other home micros. It is only because of Commodore's philosophy of opening up their machines to the programmer, providing access to a host of facilities and routines within the interpreter that others seem to do their best to lock away, that we have been able to take 64 BASIC apart and put it together again. We have had some arguments with the 64, but it remains an invaluable machine for those who wish to go beyond BASIC.

Finally, our thanks are due to the staff of the Microchips shops in Winchester and Southampton, who provided invaluable help when it came to seeking out equipment and supplies. Thanks are also due to Jane Lawrence who regularly, on her way to bed as we pounded the keyboard into the early hours, told us what a wonderful job we were doing. We hope she was right.

Part I

CHAPTER 1
Mastercode Monitor

Every computer program, regardless of the language in which it is written, begins its life as a series of instructions stored in a coded form within the computer's memory. In the case of most languages, the instructions which make up the program are quite meaningless to the central processing unit or CPU, the computer-within-a-computer which will eventually be called on to execute the tasks dictated by the program. To overcome this problem, standing in between the program entered by the user and the CPU will be yet another program, most often built into the machine at the time of its manufacture, which takes the user's program and translates it into a form which the CPU *is* able to understand

The permanent, 'built in' program, however, performs another function, for without its help it would be impossible for the user to enter instructions in the first place. From the moment that the computer is switched on, the built in program begins its task of scanning the key board to detect an input from the outside world. It then takes those inputs and stores them in the memory in such a way that they can later be 'interpreted' for the CPU. The user who writes programs in BASIC will seldom be aware of this process. Program lines will be entered, the return key pressed and the line will become part of the program — provided that the correct grammar of BASIC has been observed. No real effort or thought is required to insert a new instruction into the program, either at the end or embedded in the middle, for the computer's memory is automatically rearranged to make space for the new input.

When we turn to programming in machine code, the situation is not quite as simple. There are no facilities built into the computer to allow a new instruction to be simply entered from the keyboard in the confidence that it will automatically be entered into the computer's memory and the present contents rearranged to make room for it. The first task of a machine code programmer is, therefore, to devise a method of entering instructions, examining memory and rearranging it to suit the developing needs of the program that is being entered. This is true whether the machine code instructions are being entered directly in the form of numbers (which is the eventual form in which they must be presented to the CPU) or by means of a special language called 'assembly language' which makes machine code programs easier to enter and understand. The sim-

plest tool which allows the necessary management of the memory to take place is called a 'Monitor' and in this chapter we shall build up a flexible Monitor program which will allow you to examine individual bytes of memory or extensive chunks and to modify their contents at will.

SECTION 1: Initialisation and Menu

MODULE 1.1

```
10000 REM*********************************
10020 REM GENERAL INITIALISATION
10030 REM*********************************
10031 BASE = 16
10032 IF LEN(PTR$)+LEN(E$)<>255 THEN CLR
 : GOSUB 19000
10035 DEV = 1
10040 DEFFN HEX(X) = (X AND 15)+48-((X A
ND 15)>9)*7
10050 DEFFN DEC(X) = X-48+(X>57)*7
10060 FALSE = 0 : TRUE = -1
10070 POKE 53281,1 : POKE 53280,15
```

The purpose of this module is to set up a number of variables which will be used later in the program. The function of the variables is explained briefly in the table given in Appendix but full understanding will only come as subsequent sections of the program are entered and the variables actually used. At this stage it is enough simply to enter the module correctly — the only visible effect of RUNning it will be to change the screen colour.

CHECKSUM TABLE

```
10000 123     10020 238     10030 123
10031 116     10032 164     10035 2
10040 182     10050 132     10060 21
10070 220
```

This table of checksums is here to help you ensure that you make no errors in the entry of what will be a long and complex program. For an understanding of how to use the table see Appendix A, which gives the listing of the checksum program to be added to the end of the Mastercode program when you start, enabling you to generate your own checksum tables for comparison with those given in the book.

MODULE 1.2

```
19000 REM*******************************
19001 REM TEMPORARY LINES
19002 REM*******************************
19010 RETURN
19011 REM ***END OF MONITOR PROGRAM***
```

This temporary module is placed into the program at this point to allow for the fact that the initialisation routine calls up a later section of the Mastercode program which will not be entered yet. The lines contained in this module will be overwritten when subsequent sections of the Mastercode program are entered. The format is not critical, so no checksum table is needed.

MODULE 1.3

```
10100 REM*******************************
10101 REM CONTROL ROUTINE FOR MONITOR
10102 REM*******************************
10110 DATA EXIT TO BASIC,MEMORY MODIFY,M
EMORY DUMP,MACHINE CODE EXCUTE
10111 DATA LOAD MACHINE CODE FILE,SAVE M
ACHINE CODE FILE
10120 DATA DISASSEMBLER
10130 DATA FILE EDITOR
10140 DATA ASSEMBLER
10190 DATA END
10200 RESTORE
10220 X = 0
10230 PRINT "[BLUE][CLR]--------- MACHIN
E CODE MONITOR ---------[GREEN][CD]"
10250 READ T$
10260 IF T$<>"END" THEN PRINT TAB(5) X "
)" T$ : X= X+1 : GOTO 10250
10265 IF X<15 THEN FOR Y = X TO 15 : PRI
NT : NEXT
10270 PRINT "COMMAND ( 0 -" X-1 " ) : ";
: INPUT T
10300 IF T<0 OR T>X THEN 10100
10305 IF T=0 THEN PRINT "[CLR][4*CD]
              [RVS ON]BYE[RVS OFF][4*CD]"
 : CLOSE 1 : END
10310 ON T GOSUB 13100,13300,13500,14300
,14100,15800,24800,20000
10320 GOTO 10100
```

Every complex program needs to provide the user with a means of selecting which of its many functions is to be used next. Such a facility is called a 'control routine' or, more simply, a 'menu'. The menu given here is more complex than it need strictly have been for the current program. This is because the Monitor program is designed so that it can later be extended by adding subsequent sections of the overall Mastercode Assembler. Rather than having to enter new program lines to take account of the extra functions that will be provided, the menu will automatically extend itself to take account of new option names entered into the data statements.

CHECKSUM TABLE

```
10100 123     10101 101     10102 123
10110 180     10111 62      10120 33
10130 170     10140 65      10190 122
10200 140     10220 122     10230 192
10250 31      10260 210     10265 160
10270 234     10300 8      /10305 233
10310 199     10320 155
```

SECTION 2: Output of Memory Contents to Screen

In this section of the program we shall examine those sections of the program which are necessary to enable us to print out, in an orderly fashion, the contents of a specified area of memory. The modules commented on here may appear very insignificant and you may wonder why it is that they have not been run together to make one module. As you continue through the Mastercode program, however, you will see that individual modules may actually be called up for use from many different parts of the program. Keeping the modules to one particular function and one *only* will enable us to save on the eventual number of program lines employed rather than have to duplicate the same lines later in another section of the program.

MODULE 1.4

```
11000 REM*******************************
11001 REM CONVERT DECIMAL TO HEX
11002 REM*******************************
11010 T = H : H$ = ""
11020 H$ = CHR$(FNHEX(T-INT(T/16)*16))+H
$ : T = INT(T/16) : IF T>0 THEN 11020
11050 RETURN
```

This three line module transforms a decimal number into a hexadecimal number, that is one with a base of 16 rather than a base of 10. Machine-code programmers almost universally use hexadecimal numbers, for the simple reason that they conform much more logically to the system of binary arithmetic used by a computer.

The hexadecimal numbering system has 16, rather than 10 digits, as follows: 0 1 2 3 4 5 6 7 8 9 A B C D E F. Most modern computers store numbers in units of 256 (0 - 255), and the reason that hexadecimal is so convenient is that with a two digit hexadecimal number, the maximum value which can be expressed is also 255 (15*16 for the high digit and 15 for the low). Using hexadecimal means that a much more orderly representation of the values stored in memory can be made. In addition, the binary system used by the computer means that very often apparently significant numbers in hexadecimal like 1000 (or 4096 in decimal) are also significant in terms of the operation of the computer. Beginning to think in hexadecimal is an important aid to beginning to understand the workings of the micro.

Commentary

11020: The operation of this line is best explained by use of an example. Assume that the decimal number 4875 has been stored in the variable H. To convert that value into hexadecimal, we need to first recognise that it is made up of 1*16↑3 (16↑3=4096) + 3*16↑2 (16↑2=256)+ 0*16↑1 (16↑1=16) + 11*16↑0(16↑0=1). This line isolates each of these units of different powers of 16 and then translates them into a character which represents the appropriate hexadecimal digit, using the user defined function FNHEX (see line 10040) to select the correct character. In the case of 4875 the hexadecimal number will be 130B. For units with a value from 0 to 9, FNHEX simply returns the value of the appropriate character 0 - 9 (character codes 48 - 57). If the value of the unit is from 10 - 15, then a further 7 is added to the character code value to take it into the range A - F in the 64's character set.

CHECKSUM TABLE

```
11000 123     11001 167     11002 123
11010 170     11020 74      11050 142
```

MODULE 1.5

```
11950 REM*********************************
11951 REM CONVERT HEX IN H$ TO DEC IN H
11952 REM*********************************
11975 ERR = FALSE : H = 0 : IF LEN(H$)=0
 THEN 12030
11980 FOR X = 1 TO LEN(H$)
11990 T = FNDEC(ASC(MID$(H$,X,1))): H =
H*BASE+T
12010 IF T>BASE-1 OR T<0 THEN ERR = TRUE
12020 NEXT X
12030 RETURN
READY.
```

Although it is more sensible to input numbers in hexadecimal, working in BASIC does mean that they have to be translated into ordinary decimal for use by the program. This is accomplished by the current module.

Commentary

11975: Throughout the program the variable ERR (error) will be used to indicate that an error has been discovered. The normal value of ERR will be 0, which is the value assigned to the variable FALSE in the initialisation routine. Whenever an error is detected ERR is reset to the value of TRUE, which is minus one. The point behind using these 'truth' values is that it also allows ERR to be set by a statement such as ERR = (A> 50). The expression in brackets has a value according to whether it is true or false. If false it will take the value zero, if true it will have the value minus one. By this means ERR can be set to show that something is wrong much more economically than using statements such as IF A> 50 THEN ERR = -1.

11980-12020: Examining each character of the string H$ in turn, this loop extracts the decimal value of the hexadecimal character using the user defined function FNDEC (line 10050). Since the loop works from the left, the result obtained so far must be multiplied by 16 for each subsequent hexadecimal digit. If a character outside the range 0 - F is input, the ERR variable is set to minus one as a warning to subsequent modules.

CHECKSUM TABLE

```
11950 123     11951 230     11952 123
11975 142     11980 128     11990 40
12010 208     12020 250     12030 142
```

MODULE 1.6

```
12050 REM*******************************
12051 REM INPUT START ADDRESS
12052 REM*******************************
12057 H$ = ""
12060 INPUT "START ADDRESS ( IN HEX ) :
"; H$ : GOSUB11950
12080 IF ERR OR H<0 OR H>65535 THEN 1206
0
12090 AD = H : RETURN
```

When printing the contents of an area of memory to the screen, it is necessary to specify the start point in memory. This is done in hexadecimal, and the input is then translated into decimal by the previous module.

CHECKSUM TABLE

```
12050 123     12051 19      12052 123
12057 162     12060 50      12080 128
12090 199
```

MODULE 1.7

```
11850 REM*******************************
11851 REM ASK CONTINUE ?
11852 REM*******************************
11858 T$ = ""
11860 INPUT "CONTINUE ( Y/N ) : "; T$
11870 IF T$="Y" THEN CO = TRUE : GOTO 11
895
11880 IF T$<>"N" THEN PRINT "[CU]"; : GO
TO 11850
11890 CO = FALSE
11895 RETURN
```

When an area of memory is dumped to the screen, this module is called to enquire whether the user wishes to continue with another.

CHECKSUM TABLE

```
11850 123     11851 114     11852 123
11858 174     11860 34      11870 72
11880 235     11890 239     11895 142
```

MODULE 1.8

```
11100 REM*******************************
11101 REM BYTE INTO HEX
11102 REM*******************************
11110 H = PEEK(AD) : AD = AD+1
11120 GOSUB 11000
11130 IF LEN(H$)<2 THEN H$ = "0"+H$
11140 O2$ = O2$+H$
11150 RETURN
```

This module does the actual work of taking a value from a location in the memory specified by the variable AD. Module 2 is then called to transform the value into hexadecimal form — single figure hexadecimal numbers are 'padded out' with a leading zero in order to ensure a standardised format of two digits per byte of memory. Finally the hexadecimal number is added to O2$, which will be used to display the contents of the memory to the screen.

CHECKSUM TABLE

```
11100 123     11101 66      11102 123
11110 35      11120 159     11130 223
11140 82      11150 142
```

MODULE 1.9

```
13300 REM*******************************
13301 REM DUMP MEMORY TO SCREEN
13302 REM*******************************
13310 GOSUB 12050
13320 PRINT "[CLR]" : FOR X1 = 1 TO 18 :
 H = AD : GOSUB 11000
13340 O2$ = "" : O1$ = H$ : O3$ = ""
13350 FOR X2 = 0 TO 7
13360 GOSUB 11100 : O2$ = O2$+" "
13375 IF H>31 AND H<95 THEN O3$ = O3$+CH
R$(H) : GOTO 13380
13377 O3$ = O3$+"."
```

```
13380 NEXT X2
13390 PRINT O1$ TAB(5) O2$ TAB(31) O3$
13400 NEXT X1
13410 PRINT : GOSUB 11850 : IF CO THEN 1
3320
13440 RETURN
```

We have now entered all the modules which are necessary to define a start address and to pick up data from the memory. We can now proceed to the part of the program which actually does something. Having defined the start point, this module prints out the contents of an area of memory to the screen.

Commentary

13320: The X1 loop will be used to print out 18 lines, each with eight values taken from the memory, starting at the address now stored in AD.

13350-13380: The hexadecimal values returned from the previous modules are stored in the string O2$. If the value contained in the particular memory location is the code of an ASCII letter or digit, that character is stored in the string O3$, for display next to the values concerned. In most cases, the characters displayed will make no sense, since the fact that the code is that of a printable character will be purely chance. However, when examining areas of memory such as the variables area of the 64, or the structure of the BASIC program itself, or a machine code program which contains strings, this facility will be indispensable in getting a picture of what an area of memory contains, since any strings held there will be displayed.

CHECKSUM TABLE

```
13300 123     13301 129     13302 123
13310 165     13320 246     13340 173
13350 104     13360 100     13375 177
13377 90      13380 44      13390 89
13400 43      13410 86      13440 142
```

Review

Having entered this section you have the working basis of the program as a whole. In the sections which follow you will find that many of the modules employed are those which are already entered, since functions such as translating into hexadecimal are common to them all. Before moving on to enter the rest of the program, familiarise yourself with the operation of the program so far. Examine the area of memory which contains the start of

the program itself (starting at 801 hex) and the variables area. This section of the Monitor, on its own, is a powerful tool in unlocking the secrets of the 64's memory.

MONITOR: Output of Memory from 801 hex

```
801  25 08 10 27 8F 2A 2A 2A   %..'.***
809  2A 2A 2A 2A 2A 2A 2A 2A   ********
811  2A 2A 2A 2A 2A 2A 2A 2A   ********
819  2A 2A 2A 2A 2A 2A 2A 2A   ********
821  2A 2A 2A 00 42 08 24 27   ***.B.$'
829  8F 20 47 45 4E 45 52 41   . GENERA
831  4C 20 49 4E 49 54 49 41   L INITIA
839  4C 49 53 41 54 49 4F 4E   LISATION
841  00 66 08 2E 27 8F 2A 2A   ....'.**
849  2A 2A 2A 2A 2A 2A 2A 2A   ********
851  2A 2A 2A 2A 2A 2A 2A 2A   ********
859  2A 2A 2A 2A 2A 2A 2A 2A   ********
861  2A 2A 2A 2A 00 74 08 2F   ****...'/
869  27 42 41 53 45 20 B2 20   'BASE .
871  31 36 00 9B 08 30 27 8B   16...0'.
879  20 C3 28 50 54 52 24 29    .(PTR$)
881  AA C3 28 45 24 29 B3 B1   ..(E$)..
889  32 35 35 20 A7 20 9C 20   255 . .

CONTINUE ( Y/N ) :
```

SECTION 3: Modifying the Memory

Having learned how to examine the memory we now proceed to the next stage, which is altering its contents. In this section we present two more modules which will allow you to step through the memory, forwards or backwards, displaying the contents of individual bytes and, if you wish, altering the contents of the byte currently displayed.

MODULE 1.10

```
13000 REM*******************************
13001 REM GET 1 BYTE
13002 REM*******************************
13007 H$ = ""
13010 INPUT "BYTE ( IN HEX ) : "; H$
13030 GOSUB 11950
13040 IF ERR OR H<0 OR H>255 THEN PRINT
"[CU]" : GOTO 13000
13050 RETURN
```

On the basis of previous modules you should have little difficulty in discerning that this module accepts a hex value in the range 0-FF (0-255), calls up a translation into decimal and returns that value to the next module, from which it is called.

CHECKSUM TABLE

```
13000 123     13001 52      13002 123
13007 162     13010 171     13030 173
13040 192     13050 142
```

MODULE 1.11

```
13100 REM*********************************
13101 REM MEMORY MODIFY
13102 REM*********************************
13110 GOSUB 12050
13120 H = AD : GOSUB 11000 : PRINT H$ TA
B(6) "/" ; : 02$ = ""
13140 GOSUB 11100 : AD = AD-1 : PRINT H$
 SPC(6) ;
13150 T$ = ""
13160 INPUT " +,-,I,E : "; T$
13170 IF T$="+" AND AD<65535 THEN AD = A
D+1 : GOTO 13120
13180 IF T$="-" AND AD>0 THEN AD = AD-1
: GOTO 13120
13190 IF T$="E" THEN RETURN
13200 IF T$<>"I" THEN PRINT"[2*CU]" : GO
TO 13120
13210 GOSUB 13000 : POKE AD,H : GOTO 131
20
```

The purpose of this module is to allow the user to step through the memory from a chosen start address and to modify the contents of individual bytes. The major part of the module is concerned with outputting the values in each byte to the screen in a comprehensible format and to moving through the memory. Changes to memory contents are accomplished by the last line, including a call to the previous module

Commentary

13120-13140: Having obtained the start address, the address of the current byte is printed out, together with the value which it contains.

13160-13210: Four prompts are used by the module. '+' means move on to the next byte, '-' means move back one byte and 'E' quits the module. The remaining prompt is 'I', which calls up the previous module and allows a new value to be placed into the current byte.

CHECKSUM TABLE

```
13100 123     13101 112     13102 123
13110 165     13120 220     13140 17
13150 174     13160 192     13170 75
13180 116     13190 211     13200 20
13210 229
```

MODULE 1.12

```
13500 REM*******************************
13501 REM MACHINE CODE EXECUTE
13502 REM*******************************
13510 GOSUB 12050 : SYS AD : RETURN
```

Should you wish to use the monitor to enter machine code programs directly into the memory in hexadecimal form, this one line routine will allow you to call up the machine code routine without having to quit the program. It would be wise not to run any machine code program before ensuring that the program so far entered has been saved.

CHECKSUM TABLE

```
13500 123     13501 18      13502 123
13510 106
```

SECTION 4: Saving and Loading Files

Now that you have the ability to enter new values into the memory and hence to develop a machine code program, you need to be able to save the programs that you will eventually develop and enter. You also need to be able to reclaim those programs from disc or tape, depending on where you wish to store them. The four short routines which follow are designed to make this possible.

MODULE 1.13

```
11250 REM*******************************
11251 REM INPUT FILE NAME
11252 GOSUB 25500 : IF DEV=4 THEN 11290
11255 IN$ = ""
11260 INPUT " FILE NAME : "; IN$ : T = L
EN(IN$)
11280 IF T>16 OR T<0 THEN PRINT "[CD]FIL
E NAME INVALID" : GOTO 11260
11290 RETURN
```

When saving a block of information on tape or disc, this is done in the form of a 'file', a named location which must first be 'opened' before information is sent to it and then 'closed' when all the necessary information has been stored. When the information is recalled, the name of the file needs to be specified. This module allows the necessary file name to be input.

CHECKSUM TABLE

```
11250 123     11251 192     11252 119
11255 241     11260 137     11280 216
11290 142
```

MODULE 1.14

```
11200 REM*******************************
11201 REM INPUT FINISH ADDRESS
11202 REM*******************************
11205 H$ = ""
11210 INPUT "FINISH ADDRESS ( IN HEX) :
"; H$ : GOSUB 11950
11230 IF ERR OR H<0 OR H>65535 THEN 1120
0
11240 EA = H : RETURN
```

The machine code programs which you will eventually develop with the aid of the programs in this book will be contained in blocks of memory. To save them, the program must be given two pieces of information, namely where the block starts and where it finishes. We already have a routine which obtains the start address, this one performs the same function for the finish address.

CHECKSUM TABLE

```
11200 123     11201 70      11202 123
11205 162     11210 101     11230 123
11240 200
```

MODULE 1.15

```
14100 REM*********************************
14101 REM MACHINE CODE SAVE
14102 REM*********************************
14110 GOSUB 11250 : GOSUB 12050 : GOSUB
11200
14115 T$ = "N" : IF DEV=8 THEN INPUT "OV
ERWRITE EXISTING FILE ( Y/N ) : "; T$
14116 IF T$="Y" THEN IN$ = "@0:"+IN$
14120 IF DEV=8 THEN IN$ = IN$+",S,W"
14125 IF SA>EA THEN 14190
14130 OPEN 2,DEV,2,IN$ : PRINT# 2,AD : P
RINT# 2,EA
14150 FOR X = AD TO EA : PRINT# 2,PEEK(X
) : NEXT : PRINT# 2 : CLOSE 2
14190 RETURN
```

Now that we can give a name to the file in which the information contained in an area of memory is going to be stored and can specify the start point and end point, we can proceed to enter this module, which will store the information on tape or disc.

Commentary

14125: This line simply checks that the user has not defined a block of memory whose end point is before its start.

14130: A file is opened, in this case an 'output' file, with the destination of the information being dictated by the value of the variable DEV (device). In the listing of this program it is set at 1 (line 10035), which directs the output to a cassette recorder. If you are using a disc drive, then DEV should be set to 8 in line 10035. Once the output file is opened, the first two pieces of information to be stored in it are the start address (AD) and the end address (EA). Later in the program, a facility will be added to allow you to change the current device number at will.

14150: The contents of each byte in the block of memory to be saved are now stored one by one in the file. At the end of the loop the file is closed.

CHECKSUM TABLE

```
14100 123     14101 46      14102 123
14110 224     14115 32      14116 89
14120 179     14125 92      14130 108
14150 161     14190 142
```

MODULE 1.16

```
14300 REM*******************************
14301 REM MACHINE CODE LOAD
14302 REM*******************************
14310 GOSUB 11250 : IF DEV=8 THEN IN$ =
IN$+",S,R"
14320 OPEN 2,DEV,0,IN$ : INPUT# 2,SA,EA
: IF ST THEN CLOSE 2 : RETURN
14350 FOR X = SA TO EA : INPUT# 2,T : PO
KE X,T : NEXT : CLOSE 2 : RETURN
```

This module is simply the mirror image of the last one. Instead of placing information *into* a file, this module takes previously stored information *from* the file and places it back into the computer's memory.

CHECKSUM TABLE

```
14300 123     14301 31      14302 123
14310 206    /14320 84      14350 50
```

Summary

Having entered the whole of the Monitor you are now free to play about with it, though its full power will only be realised once the rest of the Mastercode program is entered. Try entering a new line: 0 A = 13. Call up the menu option which allows the memory to be changed and alter the contents of byte 805 hex to 8F (143). List the program to 1 and you will see that your first line has changed to a REM statment (143 represents REM in the program file). Unless you are very sure of what you are doing it would be wise not to try to change too many other memory locations at present, and certainly not before you have properly saved your final version of the monitor. If you do want to mess about, try modifying some of the colour attribute bytes from D800-DBFF hex, the colour attributes memory of the screen. Mistakes here are not likely to be disastrous.

CHAPTER 2
Mastercode Disassembler

Having now entered the Monitor program, which allows you to examine areas of memory and to change their contents, we now move on to the next stage, which is to enable you to translate the contents of an area of memory which contains a machine code program into a more understandable form. This more 'readable' form for a machine-code program is known as 'assembly language'.

The advantage of working with assembly language is that while POKEing numbers directly into memory does permit a machine code program to be entered, there is no easy correspondence between the numbers being entered into the memory or read back from it and the operations which the machine code program will carry out. The program is merely a list of numbers and very few programmers are ever capable of reading a program in that form without constant reference to charts containing the relevant codes and their meaning. Assembly language provides a means of inputting instructions which will be both comprehensible to the user (with a little practice) and yet represent every machine-code instruction in the program exactly. In other words the assembly language program consists of a series of instructions, or mnemonics, which correspond to individual machine-code operations which the 6502/6510 chip is capable of recognising and carrying out.

Instructions in assembly language will normally be in two parts:
1) An 'operation code' (opcode) which specifies the type of operation which the 6502/6510 chip is being asked to carry out, such as move a number from one place in memory to another, compare two values or perform an arithmetic operation on a value.
2) Having defined the type of operation which must be performed it is now necessary to define the number on which the operation is to be performed. This part of the instruction is known as the 'operand' and may consist of a number which will be acted upon directly or the address in memory of a number which is to be operated on.

A typical machine code instruction, upon which the assembly language translation is based, will therefore normally consist of one byte specifying the 'opcode' and one or two bytes which are used to derive the number to be operated upon. Some types of instruction need only one byte, that specif-

ying the opcode itself, since they invariably imply that the value to be operated upon is at a fixed location which does not need to be spelled out.

In order to translate a machine code program in the computer's memory into assembly language, a program is needed which will be capable of identifying an opcode and then of deciding how many of the succeeding bytes of memory (0,1 or 2) are part of the operand associated with that opcode. A program which is capable of doing this is known as a 'disassembler'. Its effect is to take the incomprehensible numbers which memory normally contains and to translate them into something which (with a little practice) can be read and understood by the user.

The brief and much simplified explanation given above will bear some study if this is the first time you have been introduced to the idea of a disassembler. Once the concept is straight in your mind, you should have little trouble in understanding the basis on which the following section of the Mastercode program works. By means of a series of tables stored in strings, the program is capable of identifying machine code instructions in a specified area of memory and of printing out the type of operations and their operands in assembly language. The program can be used in at least two ways:

a) For the user who is developing programs in assembly language, the Disassembler allows the program in memory to be more easily checked during the process of entering and debugging.

b) For those who wish to go further in their exploration of the memory of the Commodore 64, the program as listed is quite capable of giving a complete translation of the machine's ROM, the permanent built-in program which actually runs the machine. In this way a better understanding of the 64's internal workings can be built up and it is possible to examine ways in which individual routines within the ROM can be used effectively within the user's own programs.

SECTION 1: Setting up Tables

MODULE 2.1

```
12200 REM*********************************
12201 REM HEX LOADER
12202 REM*********************************
12210 T1$ = ""
12220 FOR X1 = 1 TO LEN(T$) STEP 2
12230 T1$ = T1$+CHR$(FNDEC(ASC(MID$(T$,X
1,1)))*16+FNDEC(ASC(MID$(T$,X1+1,1))))
12260 NEXT X1
12270 RETURN
```

The real purpose of this module will not become apparent until the tables in the following module have been explained. Its function is to take values

from the tables and to compact them into strings. The table values have been set out in the form of two digit hexadecimal values (ie numbers in the range 0-255 decimal). This module converts a pair of hexadecimal values into a single ASCII character. The characters thus formed can be economically stored in a string (T1$).

CHECKSUM TABLE

```
12200 123     12201 107     12202 123
12210 223     12220 216     12230 154
12260 43      12270 142
```

MODULE 2.2

```
19000 REM*********************************
19001 REM INITALISE DECODER TABLES
19002 REM*********************************
19005 BASE = 16
19007 DEFFN DEC(X) = X-48+(X>57)*7
19010 DIM TA$(4)
19011 T$ =     "0A22383838220238242202383
8220238"
19012 T$ = T$+"0922383838220238 0D2238383
8220238"
19013 T$ = T$+"1C0138380601273826012738 0
6012738"
19014 T$ = T$+"0701383838012738 2C0138383
8012738"
19015 T$ = T$+"2917383838172038231720381
B172038"
19016 T$ = T$+"0B1738383817203 80F1738383
8172038"
19017 T$ = T$+"2A0038383800283825002838 1
B002838"
19018 GOSUB 12200 : TA$(0) = T1$
19019 T$ =     "0C0038383800283 82E0038383
8002838"
19020 T$ = T$+"382F3838312F3038163835383
12F3038"
19021 T$ = T$+"032F3838312F3038372F36383
82F3838"
19022 T$ = T$+"1F1D1E381F1D1E38331D32381
F1D1E38"
19023 T$ = T$+"041D38381F1D1E38101D34381
```

```
F1D1E38"
19024 T$ = T$+"131138381311143811A1115381
3111438"
19025 T$ = T$+"0811383838111438ØE1138383
8111438"
19026 GOSUB 12200 : TA$(Ø) = TA$(Ø)+T1$
19027 T$ =     "122B3838122B1838192B21381
22B1838"
19028 T$ = T$+"052B3838382B18382D2B38383
82B18"
19029 GOSUB 12200 : TA$(Ø) = TA$(Ø)+T1$
19030 T$ =     "1711166112011CC1381114411
B111AA1"
19031 T$ = T$+"C7116661120ICCC1381114411
B111AA1"
19032 T$ = T$+"1711166112ØICCC1381114411
B111AA1"
19033 T$ = T$+"1711166112Ø19CC1381114411
B111AA1"
19034 T$ = T$+"1711666111111CCC1381144511
B111A11"
19035 T$ = T$+"2721666112111CCC1381144511
B11AAB1"
19036 T$ = T$+"2711666112111CCC1381114411
B111AA1"
19037 T$ = T$+"271166611211CCC1381114411
B111A"
19038 GOSUB 12200 : TA$(1) = T1$+CHR$(16
Ø)
19040 TA$(2) =          "ADCANDASLBCCBCSBE
QBITBMIBNEBPLBRKBVCBVS"
19041 TA$(2) = TA$(2)+"CLCCLDCLICLVCMPCP
XCPYDECDEXDEYEORINCINX"
19042 TA$(2) = TA$(2)+"INYJMPJSRLDALDXLD
YLSRNOPORAPHAPHPPLAPLP"
19043 TA$(2) = TA$(2)+"ROLRORRTIRTSSBCSE
CSEDSEISTASTXSTYTAXTAY"
19044 TA$(2) = TA$(2)+"TSXTXATXSTYA???"
19046 RETURN



READY.
```

These seemingly daunting tables are in fact remarkably simple if the general explanation of the working of a disassembler given above has been understood.

The three sections of the table defined between lines 19011 and 19029 are used to create, via calls to the previous module, a line in the array TA$, containing characters whose codes are in the range 0-56. These values point to a subsequent table which contains the names, in assembly language, of the 56 opcode types that are available when a machine code instruction is expressed in assembly language, plus one code which shows that an invalid opcode has been found. There are over 150 opcodes available in 6502/6510 machine code, so why only 56 representations (or mnemonics) in assembly language? The answer to the question is that machine code opcodes fall into groups, such as those which load the accumulator with a value, and such groups have a common mnemonic. Within each group, however, there are wide differences between the operands ie the way in which the value to be worked upon is obtained. Thus each opcode will have a unique operand type associated with it but a mnemonic may be capable of being associated with several different types of operand when the machine code program is translated into assembly language.

Thus, an opcode with a value of 127 would have an entry at position 127 in TA$(0). The ASCII code of the character at that position will be used to give a value between zero and 56. This value will then be used to point to three characters in the section of the table which is defined between 19940 and 19944. These five lines of text, when broken up into units of three represent all the available 6502/6510 assembly language mnemonics for opcode types.

The remaining section of the tables, defined by lines 19030 to 19037 give the type of operand which is associated with that particular opcode. The types of operand will be explained more fully subsequently.

SECTION 2: Operands and their Types

As was made clear in the foreword, there is no intention in this book to provide an introduction to 6502/6510 machine code. It is assumed that those who will wish to use the book will either already be familiar to some extent with the concepts that lie behind machine code and assembly language programming or that the book will be used in conjunction with a general 6502/6510 assembly language primer which will explain the various functions available on the 6502/6510 chip. It is, however, necessary for the understanding of the program at this stage, to provide some brief explanation of the manner in which the 6502/6510 chip understands operands, that is to say the values or memory locations on which it is capable of performing its 56 types of operation.

The 6502/6510 chip is capable of recognising 11 distinct methods, known as addressing modes, by which the value which is to be operated upon is obtained from a machine code program. Each individual opcode requires the use of one of these 11 different methods. The disassembler program must be capable of recognising the opcode and then of extracting from that opcode the type of addressing which is to be used.

The two simplest forms of addressing are *accumulator* addressing and *implied* addressing:

1) Accumulator addressing: some opcodes specify, without the need for any further spelling out of a value or memory address, that the operation to be performed is to be carried out on the contents of the accumulator register within the CPU. An example of this type of addressing would be 'shift left accumulator' (SLA in assembly language), which would shift bits 0 - 6 in the accumulator one place to the left, effectively multiplying the value represented by those bits by 2. No further reference is needed when an opcode of this type is specified and only one byte of memory is needed to represent an instruction of this kind in a machine code program.

2) Implied addressing: accumulator addressing is a special case of this addressing mode. There are other opcodes which imply within themselves the place where the value to be operated upon is to be found. An example of this would be 'transfer accumulator to Y register' (TAY). The effect of this operation is exactly what it says and there is no further need to spell out where the value to be transferred is obtained or where it will be placed. This again is a one byte instruction in machine code.

3) Immediate addressing: opcodes which employ this type of addressing require that the value to be acted upon is specified along with the opcode itself, such values being in the range 0 - 255, or the possible contents of a single byte of memory. An example of this type of opcode would be 'load accumulator immediate' (LDA). An instruction involving this opcode might be LDA #127. The effect of this instruction would be to load the accumulator with the value 127. When put into machine code this type of instruction requires one byte of memory to specify the opcode and a further one byte to specify the value to be operated upon.

4) Relative addressing: this is employed when jumps are to be made in a program and a value is required to specify the point in memory to which the execution of the program will jump. As with the previous addressing type, this value is in the range 0 - 255 but this range is split into a positive and negative half, with values from 0 - 127 implying a positive jump and values from 128 - 255 specifying a negative jump (127 is subtracted from the value). The jump is measured relative to the address of the byte following the jump instruction. An example of this type of opcode would be 'branch non-zero' (BNE). An instruction involving this opcode might take the form BNE 127 - the effect of the instruction would be that if the previous operation performed by the program had not resulted in a zero, a jump

forward would be made spanning 127 bytes of the program before execution commenced again. Relative addressing, like the previous type, employs one byte for the opcode and one byte for the operand.

Before discussing the remaining types of addressing it is necessary to understand two types which are not implemented in a pure form but which form the basis for others:

a) Indexed addressing: this method employs one of two registers in the 6502/6510 chip known as the 'index registers'. This type of opcode uses an operand which specifies an address in memory but, before this address is used, it or its contents are modified by the addition of the present contents of one of the index registers. Thus an instruction using indexed addressing requires that

i) there be a value in the index register

ii) that the operand specify an address in memory.

b) Zero page addressing: this refers to the fact that though the 6502/6510 chip has only five registers (locations within the chip into which values can be placed and easily acted upon) accessible to the machine-code programmer, this limitation compared to other popular CPU chips is overcome by regarding the whole of the memory from address zero to address 255 as being a series of 128 two-byte registers which can be called upon to store values for the CPU to operate upon. Zero page addressing is the addressing mode by which this area of memory is accessed.

Going back now to the main addressing modes provided on the 6502/6510 chip we find:

5) Zero-page indexed addressing: in this form of addressing the two modes given above are combined. In an instruction of this type the value contained in the index register might be seven, in which case the two byte operand would refer to an address in zero page memory (0-255) to which would be added the contents of the specified index register plus.

6) Indirect addressing: here the operation specified by the opcode will be performed upon an address which is not directly stated in the assembler instruction but is contained in the two bytes whose address is pointed to by the two byte operand. An example of an instruction of this type would be 'jump' (JMP). This opcode would be followed by a two byte operand. The operand is not itself the address in memory to which program execution should jump, rather the two-bytes beginning at the address specified by the operand *contain* an address. It is this second address to which the jump should be made. Thus JMP ($AAAA) would not specify a jump to address $AAAA but to the address represented by the value stored in the two bytes at $AAAA and $AAAB in the memory.

There are two further forms of indirect addressing available on the 6502/6510 chip which use the concept of indexing described before:

7) Pre-indexed addressing: as with normal indirect addressing, operands of this type contain addresses at which will be found values to be operated

upon. Before obtaining that first address, however, pre-indexed operands are added to the contents of the CPU X register. Thus if the X register contains $100 and the operand is $100, then the address at which the desired value will be sought is $200.

8) Post-indexed addressing: here the operand specifies a location in memory, and the contents of that location are first obtained, then the contents of the CPU Y register are added to that value. The result is an address upon whose contents the operation is to be performed.

9) Absolute addressing: in this type, the two byte operand specifies an address in memory at which will be found the value to be operated upon. Thus the instruction 'load accumulator', when using this type of addressing, might have the form LDA $AAAA, which would result in the loading of the accumulator with the value stored in byte $AAAA in the memory.

10 and 11) Absolute X and absolute Y addressing: in the case of these two types the address specified in the two byte operand is added to the contents of either the X or the Y register to arrive at the final address of the value to be operated upon. Thus if the contents of register X is $5 and the operand is $AAAA, then the address of the value to be operated upon for an instruction such as LDA $AAAA,X would be to load the accumulator with the contents of the byte at $AAAF in the memory.

Having given this brief explanation of the different type of operands which the 6502/6510 chip is capable of understanding, you should now find the sections of the program which deal with the creation of assembly language instructions out of their machine-code equivalents easier to understand without too much further commentary. In the modules that follow, when an opcode is picked up from memory, the program will obtain the correct types of addressing for the opcode by accessing the tables stored in the previous section, obtaining a value which it will record in the variable OP (OPerand). The value of OP when translated into an addressing mode is given in the table below and you will find it useful to refer to this when following the program modules for the Disassembler.

VALUE OF 'OP'	ADDRESSING MODE
0	Accumulator
1	Implied
2	Immediate
3	Relative
4	Zero-page indexed, X
5	Zero-page indexed, Y
6	Zero page
7	Pre-indexed indirect (X)
8	Post-indexed indirect (Y)

9	Absolute indirect
10	Absolute indexed, X
11	Absolute indexed, Y
12	Absolute

CHECKSUM TABLE

```
19000 123     19001 66      19002 123
19005 116     19007 132     19010 228
19011 136     19012 89      19013 78
19014 90      19015 91      19016 116
19017 93      19018 241     19019 164
19020 130     19021 154     19022 226
19023 193     19024 61      19025 87
19026 213     19027 189     19028 58
19029 213     19030 162     19031 141
19032 118     19033 108     19034 111
19035 147     19036 125     19037 11
19038 75      19040 84      19041 175
19042 10      19043 82      19044 238
19046 142
```

MODULE 2.3

```
15450 REM*********************************
15451 REM ACCUMULATOR (OP=0)
15452 REM*********************************
15460 O1$ = O1$+"A"
15500 REM IMPLIED (OP=1)
15510 RETURN
```

This brief module deals with the two simplest types of addressing mode:
a) Accumulator addressing: all that is required for the disassembly of this type is the addition of 'A' to the standard opcode.
b) Implied addressing: here the opcode itself implies its own operand and no further action is needed.

CHECKSUM TABLE

```
15450 123     15451 77      15452 123
15460 105     15500 49      15510 142
```

MODULE 2.4

```
15550 REM*********************************
15551 REM IMMEDIATE (OP=2)
15552 REM*********************************
15560 GOSUB 11100
15570 O1$ = O1$+"#$"+H$
15580 RETURN
```

This module deals with immediate addressing. The byte following the opcode is taken to be an operand in the range 0-255.

CHECKSUM TABLE

```
15550 123     15551 189     15552 123
15560 160     15570 133     15580 142
```

MODULE 2.5

```
15600 REM*********************************
15601 REM RELATIVE (OP=3)
15602 REM*********************************
15610 GOSUB 11100
15620 IF H>127 THEN H = H-256
15630 H = H+AD
15640 GOSUB 11000
15650 O1$ = O1$+"$"+H$
15660 RETURN
```

This module deals with relative addressing and translates the byte following the opcode into a number in the range -128 to + 127.

CHECKSUM TABLE

```
15600 123     15601 139     15602 123
15610 160     15620 239     15630 177
15640 159     15650 98      15660 142
```

MODULE 2.6

```
15300 REM*********************************
15301 REM ADD OPERAND IN OP TO O1$
15302 REM*********************************
15310 ON OP+1 GOTO 15450,15500,15550,156
00
```

```
15330 IF OP>6 AND OP<10 THEN O1$ = O1$+"
("
15340 GOSUB 11100
15350 O1$ = O1$+"$" : T$ = H$
15360 IF OP<9 THEN 15390
15370 GOSUB 11100
15380 O1$ = O1$+H$
15390 O1$ = O1$+T$
15400 IF OP=9 OR OP=8 THEN O1$ = O1$+")"
15410 IF OP-INT(OP/3)*3=1 THEN O1$ = O1$
+",X"
15420 IF OP-INT(OP/3)*3=2 THEN O1$ = O1$
+",Y"
15430 IF OP=7 THEN O1$ = O1$+")"
15440 RETURN
```

This simple section of IF statements formats the assembly language instructions according to the different addressing modes. The best way to understand the section is to compare what it does to the operand, on the basis of the value of OP, given in the table previously.

CHECKSUM TABLE

```
15300 123     15301 158     15302 123
15310 110     15330 10      15340 160
15350 156     15360 31      15370 160
15380 80      15390 92      15400 230
15410 207     15420 209     15430 107
15440 142
```

SECTION 3: Disassembly of Memory

We have now entered the sections of the program which enable the translation to be made from machine code into assembly language. It now remains to add those modules which allow the program to pick up the contents of a specified area of the 64's memory so that that the machine-code instructions it contains may be disassembled and printed to the screen.

MODULE 2.7

```
15700 REM*******************************
15701 REM DISASSEMBLE INSTRUCTION
15702 REM*******************************
```

```
15710 O2$ = ""
15715 GOSUB 11100 : H = H+1
15720 IF H>255 THEN H = 3
15730 T = ASC(MID$(TA$(0),H,1))
15750 O1$ = MID$(TA$(2),T*3+1,3)+" "
15760 OP = ASC(MID$(TA$(1),INT((H+1)/2),
1))
15770 IF (H AND 1) =1 THEN OP = OP/16
15780 OP = OP AND 15
15790 RETURN
```

CHECKSUM TABLE

```
15700 123     15701 93      15702 123
15710 219     15715 119     15720 148
15730 131     15750 148     15760 224
15770 146     15780 133     15790 142
```

This module constructs the assembly language instruction out of the information picked up from memory.

Commentary

15715-15720: The opcode byte having been obtained, its value is placed into the variable 'H'.

15730: The opcode is used to obtain a pointer value from TA$(0) which will indicate the position in TA$(2) of the three letter assembly-language format of that opcode.

15750: A space is added after the opcode to conform with standard assembly language format.

15760-15780: The addressing mode which is associated with the opcode is obtained from the table at TA$(1).

MODULE 2.8

```
15800 REM*******************************
15801 REM DISASSEMBLE MEMORY AREA
15802 REM*******************************
15810 GOSUB 12050
15820 PRINT "[CLR]" : FOR I = 1 TO 20
15825 H = AD : GOSUB 11000 : PRINT H$ TA
```

```
B(6) ;
15830 GOSUB 15700 : GOSUB 15300
15850 PRINT O2$ TAB(14) O1$
15860 NEXT I
15865 PRINT
15870 GOSUB 11850
15880 IF CO THEN 15820
15890 RETURN
```

CHECKSUM TABLE

```
15800 123     15801 13      15802 123
15810 165     15820 93      15825 244
15830 202     15850 115     15860 235
15865 153     15870 172     15880 36
15890 142
```

This is the control module which formats the assembly language instructions obtained by the previous modules. For an explanation of the various subroutine calls, see the Table of Subroutine Functions in the Appendix.

Summary

Even if you are working with this book in conjunction with a good 6502/6510 primer it will be worth, at this stage, spending some time playing with your assembler and monitor. Try disassembling some of the routines within the 64 ROM and trying to understand a little of how they function. Some interesting addresses to begin disassembly are given below, together with the purposes of the routines at that position.

Be warned, however, that any disassembler is only as good as the starting position in memory that it is given. If you start the disassembly of memory at a point which is in fact halfway through a machine-code instruction then the first few bytes, at least, of the disassembly listing will be garbage, since parts of operands will be translated as opcodes. Eventually, after rejecting a number of apparently spurious instructions and perhaps listing some nonsense instructions, the disassembler will get itself into synch with the memory. After this it will be disturbed only by tables contained in the memory, which it will again attempt to translate as if they were machine code instructions. When you run up against such problems the only solution is to move the start address along until you clear the table and meaningful instructions are discovered from the start of the disassembled listing.

Given below is a specimen disassembly of an area of the 64's interpreter starting at the address of a routine whose function is to accept the input of a new BASIC line using various subroutines in the 64's monitor and 'kernal'.

SPECIMEN DISASSEMBLY: Start Address A480 hex

```
A480        6C0203          JMP  (*0302)
A483        2060A5          JSR  *A560
A486        867A            STX  *7A
A488        847B            STY  *7B
A48A        207300          JSR  *0073
A48D        AA              TAX
A48E        F0F0            BEQ  *A480
A490        A2FF            LDX  #*FF
A492        863A            STX  *3A
A494        9006            BCC  *A49C
A496        2079A5          JSR  *A579
A499        4CE1A7          JMP  *A7E1
A49C        206BA9          JSR  *A96B
A49F        2079A5          JSR  *A579
A4A2        840B            STY  *0B
A4A4        2013A6          JSR  *A613
A4A7        9044            BCC  *A4ED
A4A9        A001            LDY  #*01
A4AB        B15F            LDA  (*5F),Y
A4AD        8523            STA  *23

CONTINUE  ( Y/N )  :
```

CHAPTER 3
Mastercode File Editor

Before proceeding to the main part of the Assembler program we shall examine the File Editor, which allows assembly language programs to be entered in a convenient form and practically edited.

In discussing the Disassembler we have already noted the format of some of the individual instructions which will be used in assembly language programs. If you have used the Disassembler to translate part of the 64's ROM then you will also have seen the format in which assembly language *programs* are normally presented, consisting of three items of information for every assembly language instruction:

1) The memory address at which the instruction is to be found.
2) The contents, in Hex, of the bytes involved.
3) The assembly language form of the instruction.

When entering an assembly language program to an assembler, only the assembly language instructions are needed. There are, however, some problems with simply entering a long list of assembly language instructions. What, for instance, if we have entered a long assembly language program and then discover that it needs a few more instructions somewhere in the middle or that some instructions need to be deleted. Does the whole thing have to entered again in the right order? Obviously the ideal method would be something like that provided by the 64's BASIC interpreter — numbered lines which are automatically deleted or inserted in the correct place, with the ability to alter lines anywhere in the program at will. It is the function of the File Editor to provide that facility, though the program section given here goes further than that, allowing renumbering of the program and the saving (or loading) of the assembly language file before the Assembler proper goes to work on it and translates it into machine code.

It should be stressed that the File Editor is not genuinely part of the Assembler in that it makes no test of the material being entered, it is purely there to allow numbered lines of text to be inserted into a file. Nothing will be checked or processed until the Assembler itself is entered and run.

SECTION 1: Setting Up

MODULE 3.1

```
24800 REM*********************************
24801 REM FILE EDITOR MENU
```

```
24802 REM*******************************
24820 PRINT "[CLR][GREEN] ------------ F
ILE EDITOR ------------[BLUE][CD]"
24835 PRINT "       0) EXIT FROM FILE EDI
TOR"
24840 PRINT "       1) INPUT LINE(S)"
24850 PRINT "       2) LIST LINE(S)"
24860 PRINT "       3) DELETE LINE(S)"
24870 PRINT "       4) RENUMBER FILE"
24880 PRINT "       5) INITIALISE FILE"
24890 PRINT "       6) LOAD FILE"
24900 PRINT "       7) SAVE FILE"
24910 PRINT "       8) ADD MACHINE CODE T
O FILE"
24915 PRINT "       9) CHANGE DEVICE NUMB
ER[5*CD]"
24920 INPUT " COMMAND ( 0-9 ) : "; CO
24940 IF CO=0 THEN RETURN
24950 IF CO>0 THEN ON CO GOSUB 24600,244
00,24500,24700,24300,23600,23700,25000
24960 IF CO>8 THEN ON CO-8 GOSUB 25500
24970 GOTO 24800
```

A straightforward menu module.

CHECKSUM TABLE

```
24800 123     24801 11      24802 123
24820 125     24835 235     24840 179
24850 96      24860 216     24870 218
24880 102     24890 156     24900 172
24910 90      24915 243     24920 182
24940 148     24950 30      24960 252
24970 167
```

MODULE 3.2

```
24300 REM*******************************
24301 REM INITALISE FILE
24302 REM*******************************
24310 PTR$ = "" : E$ = "" : FOR X = 0 TO
 254 : E$ = E$+CHR$(X) : NEXT : RETURN
```

This module sets up the variables necessary for handling a new file — calling this option when a file is already in memory will result in the loss of the existing file. The two main variables are PTR$, which will indicate the

position of entries in the file in their correct order, and E$, which will record the position of spaces for new entries. The use of PTR$ will be described under Module 6.

CHECKSUM TABLE

```
24300 123     24301 145     24302 123
24310 217
```

MODULE 3.2A

```
19980 DIM FI$(254) : GOSUB 24300
```

This module is actually a part of the main initialisation routine for the Disassembler tables. Its function is to set up the main file array (FI$) when the program is first run. Once the program is running the array is re-initialised by calling the previous module.

CHECKSUM TABLE

```
19980 101
```

SECTION 2: Inputting Lines

MODULE 3.3

```
24600 REM*******************************
24601 REM INPUT LINE(S)
24602 REM*******************************
24610 PRINT "[CLR]"
24620 IN$ = "" : INPUT IN$ : GOSUB 24000
 : IF LN=-65536 THEN 24665
24650 GOSUB 23900 : IF LEN(IN$)=0 THEN 2
4680
24660 GOSUB 23100 : IF NOT ERR THEN 2462
0
24665 RETURN
24680 GOSUB 23020 : IF NOT ERR THEN GOSU
B 23300
24690 GOTO 24620
```

This is the module which, when a line is input, allocates the necessary tasks to the File Editor's various routines. Other than distributing work around other modules, its only functions are to allow the input of the line in

the form of IN$ and to determine whether a line number without a line attached is being entered ie a deletion.

CHECKSUM TABLE

```
24600 123     24601 43      24602 123
24610 144     24620 251     24650 108
24660 94      24665 142     24680 6
24690 167
```

MODULE 3.4

```
24000 REM*******************************
24001 REM GET LINE NUMBER
24002 REM*******************************
24010 LN = -65536
24020 IF LEN(IN$)=0 OR IN$<"0" OR LEFT$(
IN$,1)>"9" THEN 24090
24030 FOR T = 1 TO LEN(IN$)
24040 IF MID$(IN$,T,1)<="9" AND MID$(IN$
,T,1)>="0" THEN NEXT T
24080 LN = VAL(LEFT$(IN$,T-1)) : IN$ = M
ID$(IN$,T)
24090 RETURN
```

Having obtained an input in the form of IN$, a line number is obtained from the beginning of the string. The string is examined character by character to find the first one which is outside the range 0 - 9, and then the VAL of the string up to that point is obtained. Strings which do not begin with a line number result in the line number (LN) being set to -65536, thus flagging an error, otherwise the line number is stored in LN and the characters containing the line number chopped off the original string.

CHECKSUM TABLE

```
24000 123     24001 192     24002 123
24010 64      24020 99      24030 203
24040 150     24080 79      24090 142
```

MODULE 3.5

```
23900 REM*******************************
23901 REM REMOVE LEADING SPACES
23902 REM*******************************
23910 FOR T = 1 TO LEN(IN$)
23920 IF MID$(IN$,T,1)=" " THEN NEXT T
23950 IN$ = MID$(IN$,T) : RETURN
```

The resulting IN$, stripped of its line number may now begin with one or more spaces — this module removes them.

CHECKSUM TABLE

```
23900 123     23901 112     23902 123
23910 203     23920 81      23950 11
```

MODULE 3.6

```
23000 REM*******************************
23001 REM FILE EDITOR
23002 REM*******************************
23010 REM FILE EDITOR
23020 REM FIND LINE NUMBER IN 'LN' IN FI
LE
23030 T = LEN(PTR$)+1 : T2 = -1
23040 T = T-1 : IF T<=0 THEN GOTO 23080
23050 T1 = ASC(MID$(PTR$,T,1))
23060 T2 = ASC(MID$(FI$(T1),1,1))+256*AS
C(MID$(FI$(T1),2,1))
23070 IF T2>LN THEN 23040
23080 ERR = NOT(T2=LN) : IF ERR THEN T =
 T+1
23090 RETURN
```

Before we proceed to the module which actually inserts a line into the file, we must deal with this one, whose function is to determine the correct position for the new line (if it has a valid line number). In examining the module we shall learn something of the use of PTR$.

Commentary

23030: In searching for the correct position to insert a line we shall make use of the string we have called PTR$, short for 'pointer string'. A pointer string is a standard method of overcoming the problems of inserting new lines in multi-line arrays. It is not that this is difficult, it is simply that to insert a new line at the beginning of what is potentially an array of some 250 lines involves shifting all the current lines, a task which can be time consuming and can also create problems with garbage collection, slowing things down even more. Using a pointer string this can be overcome, since the contents of the array need never be shifted at all. All that needs to be done is to manipulate a single string. Instead of finding the correct place in the array and then shifting everything else to make room for the new line, what we shall do is to find what *should be* the correct position (as dictated by the line number), place the line to be entered in the first empty space we

find and then put an indication of its actual position in the right place in the pointer string.

Thus the pointer string might contain a series of bytes with values of 34,76,233,176..... What this would mean is that the true first line in the list is to be found at position 34, the second line is at position 76, the third at position 233 and so on. To access the array of lines in order we must first look at PTR$, take from that the position of the first line, then look at the second character of PTR$ to find the position of the second line. Because we have accepted the arbitrary limit of 255 lines for any one file all the pointers can be held in the form of single characters in PTR$ — single characters can have an ASCII value of 0-255. To insert a new entry, all that will be necessary is to split PTR$ into two and place a new indicator in the middle of it — a considerable saving of time. In this particular line the main search variable (T) is set to LEN(PTR$) + 1, so that the search will begin at the end of PTR$.

23050: The value of character T in PTR$ is the position of what should be line T in the array (not the line with *line number* T but position T if we counted from the beginning of the file).

23060: This obtains the line number of the line stored in FI$ at position T1.

23070: The search continues until a line number is found which is greater than that of the line being entered (LN).

23080: Note that ERR is set if the line number being entered is not the same as one already in the file. This is so that the next module will know whether a line is being inserted or overwritten.

CHECKSUM TABLE

```
23010 182     23020 183     23030 29
23040 17      23050 160     23060 86
23070 92      23080 16      23090 142
```

MODULE 3.7

```
23100 REM*******************************
23101 REM ADD LINE TO FILE
23102 REM*******************************
23105 IF LN<0 OR LN>65535 THEN 23215
23110 GOSUB 23020
23120 IF NOT ERR THEN T1 = ASC(MID$(PTR$
,T,1)) : GOTO 23150
23130 IF E$="" THEN ERR = TRUE : GOTO 23
```

```
220
23140 T1 = ASC(E$)  : E$ = MID$(E$,2)
23150 T2 = INT(LN/256)
23160 FI$(T1) = CHR$(LN-T2*256)+CHR$(T2)
+IN$
23170 IF NOT ERR THEN 23220
23180 T$ = "" : T1$ = ""
23190 IF T>1 THEN T$ = LEFT$(PTR$,T-1)
23200 IF T<=LEN(PTR$) THEN T1$ = MID$(PT
R$,T)
23210 PTR$ = T$+CHR$(T1)+T1$
23215 ERR = FALSE
23220 RETURN
```

This is the module which actually accomplishes the insertion of the line into the file.

Commentary

23105: Using two bytes, 0-65535 is the maximum range of possible line numbers.

23120: If an error is returned from the previous module, all it means is that a new line number is being entered. If there is no error then the line being entered will simply overwrite an existing line and PTR$ does not need to be altered at all.

23130: To speed up the process of entry even more, a second string (E$) is used to record all the empty spaces in the file. Rather than scanning for the first available empty space, the line will be inserted in the position indicated by the first character in E$ — this character is now lopped off since it will no longer be empty.

23150-23160: The line number bytes are created from LN and the new line inserted. Note that having put the high byte into the variable T2 (=LN/256), there is no need to make another variable equal to LN-256*INT(LN/256). Simply putting LN into ASCII form will lose anything above 255 as if LN had been ANDed with 255.

23170-23210: If we are dealing with a *new* line number then PTR$ must have a character added to it. The position of the character is indicated by T

and all that is necessary is to take LEFT$(PTR$,T-1) and MID$(PTR$,T) then to add the necessary character between them.

CHECKSUM TABLE

```
23100 123     23101 195     23102 123
23105 79      23110 164     23120 1
23130 40      23140 206     23150 98
23160 84      23170 60      23180 7
23190 193     23200 201     23210 30
23215 70      23220 142
```

MODULE 3.8

```
23300 REM*********************************
23301 REM DELETE LINE POINTED AT BY T
23302 REM*********************************
23310 T$ = "" : T1$ = ""
23320 IF T>1 THEN T$ = LEFT$(PTR$,T-1)
23330 IF T<LEN(PTR$) THEN T1$ = MID$(PTR
$,T+1)
23340 E$ = E$+MID$(PTR$,T,1)
23350 PTR$ = T$+T1$
23360 RETURN
```

This may seem a strange module to discuss under the heading of input of lines, since its purpose is to delete them. The reason we talk about it here is that, when inputting lines, if you input a line number without a line attached, the line with that number is deleted in the same way that it would be in BASIC. The module is therefore called from the main control module for input. The procedure followed is a mirror image of that involved in insertion, with a pointer character being removed from PTR$ and the line's position being recorded as a space in E$. Note that there is no need to actually remove the contents of the line — it is still there but the File Editor does not recognise its existence any longer and will overwrite it when a new line is entered.

CHECKSUM TABLE

```
23300 123     23301 193     23302 123
23310 7       23320 193     23330 242
23340 128     23350 215     23360 142
```

SECTION 3: Listing and Deleting

MODULE 3.9

```
24200 REM*******************************
24201 REM  FIRST AND LAST LINES
24202 REM*******************************
24205 IN$ = "" : INPUT "FIRST - LAST LIN
ES : "; IN$
24210 SL = 0 : FL = 65535 : T3 = 0 : ERR
 = FALSE
24220 IF LEN(IN$)=0 THEN 24295
24230 GOSUB 24000
24240 IF LN>=0 THEN SL = LN : GOTO 24260
24250 IF LN>-65536 THEN FL = -LN : GOTO
24295
24260 GOSUB 23900 : IF LEN(IN$)=0 THEN F
L = SL : GOTO 24295
24270 IN$ = MID$(IN$,2) : GOSUB 23900
24290 IF LEN(IN$)>0 THEN GOSUB 24000 : F
L = LN
24295 ERR = SL<0 OR SL>65535 OR FL<0 OR
FL>65535 OR ERR : RETURN
```

This module is used in listing and block deletion to get a pair of line numbers input in the format '100-330'.

Commentary

24210-24220: The start line (SL) and finish line (FL) are set to the ends of the permissible range. If the user simply presses return when the prompt appears, the whole of the file will be listed from start to finish.

24230-24250: The first line number is obtained through the subroutine at 24000. If it is greater than zero then SL is set equal to it. If '-300' were input this will be returned as minus 300. In this case SL will remain at zero but FL will be set to 300 and the file will be listed up to line 300.

24260: Any leading spaces are stripped from what remains of IN$ after the first number has been removed. If there is nothing left then FL is set equal to SL and only one line is listed.

24270-24290: IN$ is stripped of the '-' before the second number, any leading spaces are removed and the second value obtained.

CHECKSUM TABLE

```
24200 123     24201 57      24202 123
24205 168     24210 170     24220 73
```

```
24230 163     24240 11      24250 131
24260 180     24270 6       24290 125
24295 38
```

MODULE 3.10

```
23400 REM*******************************
23401 REM LIST LINES POINTED AT BY T
23402 REM*******************************
23410 PRINT ASC(MID$(FI$(T),1,1))+256*AS
C(MID$(FI$(T),2,1)) TAB(6) ;
23420 PRINT MID$(FI$(T),3)
23430 RETURN
```

This module prints a line whose position is indicated by the variable T. The line number is obtained from the first two characters of the line, then the rest of the line is printed.

CHECKSUM TABLE

```
23400 123     23401 157     23402 123
23410 178     23420 139     23430 142
```

MODULE 3.11

```
23500 REM*******************************
23501 REM START AND FINISH POINTERS
23502 REM*******************************
23510 LN = SL : GOSUB 23020
23520 SP = T
23530 LN = FL : GOSUB 23020
23540 FP = T
23545 IF ERR THEN FP = FP-1
23550 IF FP>LEN(PTR$) THEN FP = LEN(PTR$
)
23560 RETURN
```

Using the start and finish line numbers, this module picks up from PTR$ the pointers to the first and last lines to be listed and stores them in SP and FP.

CHECKSUM TABLE

```
23500 123     23501 165     23502 123
23510 73      23520 233     23530 60
23540 220     23545 117     23550 189
23560 142
```

MODULE 3.12

```
24400 REM*******************************
24401 REM LIST LINES
24402 REM*******************************
24410 GOSUB 24200 : IF ERR THEN 24460
24420 PRINT "[CLR]" : GOSUB 23500 : IF F
P<SPOR FP=0 THEN 24460
24430 FOR T1 = SP TO FP : T = ASC(MID$(P
TR$,T1,1)) : GOSUB 23400 : NEXT : PRINT
24455 IF PEEK(152)=0 THEN GET T$ : IF T$
="" THEN 24455
24460 RETURN
```

Using the start and finish pointers determined by the previous module, this module now calls up the print module to list the lines to the screen. The strange looking line at 24455 checks to see whether the lines are actually being listed to a device such as the tape recorder or a printer. If not, the listing will be displayed on the screen until a key is pressed.

CHECKSUM TABLE

```
24400 123     24401 134     24402 123
24410 154     24420 241     24430 126
24455 214     24460 142
```

MODULE 3.13

```
24500 REM*******************************
24501 REM DELETE LINE(S)
24502 REM*******************************
24510 GOSUB 24200 : IF ERR THEN 24460
24520 GOSUB 23500 : IF FP<SP THEN 24560
24530 T = SP : FOR T1 = SP TO FP : GOSUB
 23300 : NEXT
24560 RETURN
```

This is the block delete module. It is included at this point because its sole function is to call up modules previously entered, the largest of which is the

'get first and last lines' routine. Instead of listing the lines specified, the line delete module is called up for each line in turn. Note that because PTR$ is being shortened with each deletion, the character deleted for each iteration of the loop is always at the same position.

CHECKSUM TABLE

```
24500 123     24501 78      24502 123
24510 154     24520 160     24530 179
24560 142
```

SECTION 4: Loading and Saving

MODULE 3.14

```
23700 REM*********************************
23701 REM SAVE FILE TO DEVICE
23702 REM*********************************
23705 GOSUB 11250
23710 IF DEV=8 THEN IN$ = IN$+",S,W"
23715 T$ = "N" : IF DEV=8 THEN INPUT "OV
ERWRITE EXISTING FILE ( Y/N ) : "; T$
23716 IF T$="Y" THEN IN$ = "@0:"+IN$
23720 OPEN2,DEV,2,IN$ : CMD 2
23730 SL = 0 : FL = 65536
23750 GOSUB 24420 : PRINT#2 , "END"
23760 PRINT#2 : CLOSE 2
23780 RETURN
```

This module allows a file that you have created to be saved onto tape or disc, or output to a printer.

Commentary

23705: The routine from the Monitor which requests a file name.

23710-23716: These lines are included for the benefit of those using disc units for storage. Their effect is to allow the user to overwrite an existing file on drive zero with a sequential file of the contents of FI$. The lines are only accessed if DEV is set to 8 (disc drive).

23720-23760: A file is opened to the specified device and the CMD2 instruction specifies that all further output will be sent to that device. All that remains is to use the normal listing routines to print all the lines of the file, terminate them with 'END' as a marker and finally close the file.

CHECKSUM TABLE

```
23700 123     23701 177     23702 123
23705 166     23710 179     23715 32
23716 89      23720 138     23730 200
23750 116     23760 54      23780 142
```

MODULE 3.15

```
23600 REM*********************************
23601 REM LOAD FILE FROM DEVICE
23602 REM*********************************
23610 GOSUB 11250
23615 IF DEV=8 THEN IN$ = IN$+",S,R"
23630 OPEN2,DEV,0,IN$
23635 INPUT#2 , IN$ : IF ST THEN GOTO 23
650
23640 IF IN$<>"END" THEN GOSUB 24000 : G
OSUB 23900 : GOSUB 23100 : GOTO 23635
23650 CLOSE 2
23660 RETURN
```

The mirror image of the previous module. Note that when loading the file back from tape or disc, all the normal input routines have to be used. This is because the lines were listed in full with their line numbers, not the two byte form of the line numbers that is normally stored in FI$. The cassette saving/loading system has difficulty in saving non-printable ASCII characters and simply saving the contents of FI$ would result in the corruption of some of the line numbers on reloading.

CHECKSUM TABLE

```
23600 123     23601 51      23602 123
23610 166     23615 174     23630 31
23635 57      23640 215     23650 242
23660 142
```

MODULE 3.16

```
25500 REM*********************************
25501 REM CHANGE DEVICE NUMBER
25502 REM*********************************
25510 PRINT SPC(19) DEV
25520 INPUT "[CU]NEW DEVICE NUMBER:";DEV
25530 RETURN
```

The purpose of this module is to allow output to be made to cassette, disc or printer. Note that trying to output to, or input from, a device which is not present, or to input from a device which is not capable of giving an input (such as the printer) may result in the program stopping. Data will not be lost provided that you start the program with GOTO 10000 rather than RUN. Before doing so it would be wise to ensure that file 2 is closed by entering PRINT#2: CLOSE2 if you were saving when the program stopped or simply CLOSE2 if you were loading. This will avoid the possibility of "FILE ALREADY OPEN" errors being given.

CHECKSUM TABLE

```
25500 123     25501 14      25502 123
25510 241     25520 113     25530 142
```

SECTION 5: Renumbering

MODULE 3.17

```
24700 REM*******************************
24701 REM RENUMBER FILE IN STEPS OF 10
24702 REM*******************************
24710 LN = 10 : ERR = FALSE
24720 IF LEN(PTR$)<1 THEN 24780
24730 FOR T = 1 TO LEN(PTR$)
24735 T1 = ASC(MID$(PTR$,T,1))
24740 FI$(T1) = CHR$(LN-INT(LN/256)*256)
+CHR$(LN/256)+MID$(FI$(T1),3)
24750 LN = LN+10 : NEXT
24780 RETURN
```

Hardly worth a section in its own right, but the module does perform independently of everything else you have entered so far. Its purpose is to renumber your file in steps of 10. This is done by stripping each entry in the file of its first two characters and then recreating them from LN, which is incremented by 10 for each line.

CHECKSUM TABLE

```
24700 123     24701 235     24702 123
24710 173     24720 169     24730 42
24735 160     24740 94      24750 45
24780 142
```

MODULE 3.18

```
25000 REM*******************************
25001 REM ADD TO FILE FROM MEMORY
25002 REM*******************************
25010 GOSUB 12050 : GOSUB 11200 : GOSUB
24200
25050 FOR XY = AD TO EA STEP 15
25060 IN$ = " BYT " : LN = SL : SL = SL+
5
25070 FOR XZ = 0 TO 14 : O2$ = ""
25080 GOSUB 11100 : IN$ = IN$+"$"+H$
25100 IF XZ<14 AND AD<=EA THEN IN$ = IN$
+"," : NEXT XZ
25110 GOSUB 23100 : NEXT XY : RETURN
```

We confess that this module is a bit of an afterthought, but a nice one for all that. Its relevance really won't become clear until you have entered the Assembler but what it does is allow you to specify an area of memory and then place it into an assembly language program file in the form of 'byte directives' — the contents of each memory location are specified in the assembler file. No automatic adjustment is made to instructions which access addresses in the area from which the code was originally lifted. Such instructions will still refer to the original area of memory.

CHECKSUM TABLE

```
25000 123     25001 200     25002 123
25010 223     25050 130     25060 78
25070 19      25080 170     25100 13
25110 120
```

Summary

Now that you have entered the File Editor it would be wise to play with it for a while before going on to enter the Assembler. This will help to avoid the upset of entering a long assembly language file and then having it spoiled because you misuse the File Editor. You could, if you wish, enter one or two of the assembly language programs to be found later in this book, saving them to disc or tape and then reloading them to check that you have the procedure off pat.

CHAPTER 4
Mastercode Assembler

Having entered the File Editor, we can now begin on the process of entering the most important and complex part of the Mastercode program, the Assembler. Its purpose is to allow you to enter programs in assembly language, together with a variety of features which make such programming easier, and then to see them automatically translated into a machine code program. The price that has been paid for the flexibility and power of this part of the program is that it is immensely complex. Entering it will be a long job for you, and no doubt there will be many errors along the way, here you must rely on the Checksum Tables to guide you. At the end of the process you will have the same program that we used to develop all our machine code routines for this book. The program works and that is sufficient justification for the effort that it will involve.

SECTION 1: Initialisation

MODULE 4.1

```
19046 TA$(2) = TA$(2)+"BYTWRDDBYENDORGPR
TSYM"
19047 T$ =     "61210690B0F02430D01000507
018D858"
19048 T$ = T$+"B8CDECCCCECA884DEEE8C84C2
0ADAEAC"
19049 T$ = T$+"4AEA0D480868282A6A4060ED3
8F8788D"
19050 T$ = T$+"8E8CAAA8BA8A9A98"
19051 GOSUB 12200 : TA$(3) = T1$
19052 T$ =     "FF11FFFFFF090AFFFF1D0EFFF
F051EFF"
19053 T$ = T$+"FF15FFFFFFFFFFFFF01FFFFF
F1916FF"
19054 T$ = T$+"FF2DFFFF2C293EFFFF3D2EFFF
F2526FF"
19055 T$ = T$+"FF35FFFFFFFFFFFFFF31FFFFF
F3936FF"
19056 T$ = T$+"FF51FFFFFF495EFFFF5D4EFF6
```

```
C4546FF"
19057 T$ = T$+"FF55FFFFFFFFFFFFF41FFFFF
F5956FF"
19058 T$ = T$+"FF6DFFFFFF697EFFFF7D6EFFF
F6566FF"
19059 GOSUB 12200 : TA$(4) = T1$
19060 T$ =     "FF75FFFFFFFFFFFFF71FFFFF
F7976FF"
19061 T$ = T$+"FF91FFFF949D96FFFFFFFFFF8
48586FF"
19062 T$ = T$+"FF95FFFFFFFFFFFFF81FFFFF
F99FFFF"
19063 T$ = T$+"BCB1BEFFA0A9A2FFFFBDFFAFA
4A5A6FF"
19064 T$ = T$+"FFB5FFFFFFFFFFFFFA1FFFFB
4B9B6FF"
19065 T$ = T$+"FFD1FFFFC0C9DEFFFFDDFFFFC
4C5C6FF"
19066 T$ = T$+"FFD5FFFFFFFFFFFFFC1FFFFF
FD9D6FF"
19067 GOSUB 12200 : TA$(4) = TA$(4)+T1$
19068 T$ =     "FFF1FFFFE0E9FEFFFFFDFFFFE
4E5E6FF"
19069 T$ = T$+"FFF5FFFFFFFFFFFFFE1FFFFF
FF9F6"
19070 GOSUB 12200 : TA$(4) = TA$(4)+T1$
19080 SM = 50 : SE = 0 : DIM STABLE$(SM)
19101 DIM ERR$(18)
19103 ERR$(1) = "SINGLE BYTE OUT OF RANG
E"
19104 ERR$(2) = "DOUBLE BYTE OUT OF RANG
E"
19105 ERR$(3) = "INVALID OPRAND OR OPCOD
E"
19106 ERR$(4) = "INVALID OPERATOR"
19107 ERR$(5) = "INDEX IS NOT X OR Y"
19108 ERR$(6) = "LABEL NOT ALPHA-NUMERIC
"
19109 ERR$(7) = "INCORRECT NUMBER BASE"
19110 ERR$(8) = "LABEL DEFINED TWICE"
19112 ERR$(10) = "BRANCH OUT OF RANGE"
19113 ERR$(11) = "UNDEFINED LABEL"
19114 ERR$(12) = "ONLY SINGLE CHR. EXPEC
TED"
```

```
19116 ERR$(14) = "OUT OF SYMBOL SPACE"
19117 ERR$(15) = "DIVISION BY ZERO"
19120 ERR$(18) = "ADDRESSING MODE NOT AV
AILBLE WITH THIS OPCODE"
19980 DIM FI$(254) : GOSUB 24300
19990 RETURN

READY.
```

If you have been taking note of what you have entered so far you will immediately realise that what you are about to enter here is not a module that stands alone in its own right but an addition to an existing module, namely the initialisation module for the tables of opcodes and operand types upon which the Disassembler works. In the case of the Assembler the same tables will be used, but in the reverse direction. Instead of finding a value in the memory and then looking up an appropriate opcode mnemonic and addressing mode, the assembler will scan the files entered through the File Editor and try to construct the machine-code equivalent of each line — either that or reject the line as an invalid instruction.

If you think about it, this requires some more information for, instead of being able to read a value and then choose a format based upon that opcode, the Assembler must, on finding an instruction like 'load' at the beginning of a line like 'LDA $AAAA.X', be able to scan through all the possible formats for a 'load' instruction to see whether or not the present instruction is a permissible one. In order to achieve this, two more tables are added to those already stored in the program. Between 19047 and 19050 is stored a table of two-character hex numbers corresponding to each of the possible opcodes stored in TA$(2) for the Disassembler. These show, for each opcode, the first operand type which may be used. Lines 19052-19067 consist of further operand types for each particular group of opcodes.

Later in the program we shall see how each possible operand type is compared with what is actually in the assembler instruction contained within a line in FI$. For the moment you will do well to simply understand that on detecting an instruction beginning with ADC (the first three character opcode in TA$(2), the Assembler will go to TA$(3). It will then discover that this may possibly be an instruction involving opcode 61 (hex) and will examine the format of the assembler instruction to see whether it fits the format required by opcode 61 hex (eg ADC ($50,X). If the format of the line being entered does not conform to that needed by opcode 61 hex, then the value of 61 hex (97 decimal), will be used to find the next possible opcode in TA$(4). This will be found in the 98th character pair of TA$(4) (numbering always starts from zero) and the opcode there is 6D signifying another instruction involving ADC, but this time taking a format such as ADC $AAA. The value 6D is then used to find the next opcode in the table which would produce an instruction beginning with ADC.

In the case of ADC there are eight possible opcodes and if after examining each one against the actual format of the instruction in the line in FI$ none of them fit, the last possible opcode will contain the value FF, indicating that the end of the chain of possible ADC instructions has been reached and that no permissible opcode conforms to what is actually in the line. If you care to work through the tables with any three letter opcode type you care to choose, first of all finding its position in TA$, then finding the start of the corresponding chain of opcode values in TA$(3) and following the chain through TA$(4) you should quickly be able to see what is happening. The one real addition to the tables set up already by the disassembler is that made by line 19046. This apparently adds seven new opcode types to the list which the Disassembler worked upon. These are the assembler directives, seven instructions which are not actually assembly language instructions but rather instructions to the Assembler to behave in a certain way while it is processing on the assembly language program. The seven directives, BYT, WRD, DBY, END, ORG, PRT and SYM will be explained fully later in the program.

From 19100 to 19120 you will find the various error messages the Assembler is capable of generating when it comes across invalid instructions or omissions from the program. These too will be explained more fully in due course.

CHECKSUM TABLE

```
19046 193     19047 171     19048 189
19049 252     19050 170     19051 244
19052 251     19053 248     19054 187
19055 1       19056 187     19057 8
19058 238     19059 245     19060 79
19061 198     19062 53      19063 216
19064 38      19065 13      19066 68
19067 221     19068 91      19069 192
19070 221     19080 27      19101 109
19103 53      19104 47      19105 77
19106 91      19107 192     19108 1
19109 152     19110 215     19112 253
19113 8       19114 184     19116 40
19117 122     19120 37      19980 101
19990 142
```

MODULE 4.2

```
20000 REM*********************************
20001 REM GENERATE ASSEMBLY LISTING
20002 REM*********************************
```

```
20005 SE = 0 : FMAX = LEN(PTR$) : SY = F
ALSE
20010 INPUT " ERROR ONLY LISTING ( Y/N )
 :"; T$
20020 EO = LEFT$(T$,1)="Y"
20025 INPUT " ASSEMBLE TO MEMORY ( Y/N )
 :"; T$
20029 AM = LEFT$(T$,1)="Y"
20030 AD = 0 : REM SET START ADDRESS
20040 FOR Q = 1 TO FMAX
20050 IN$ = FILE$(ASC(MID$(PTR$,Q,1))) :
 O$ = ""
20060 GOSUB 26400
20070 IF EXIT THEN Q=FMAX+1
20080 NEXT Q
20085 T = FRE(X)
20090 AD = 0 : EC = 0 : PRINT "ADD.  DAT
A      SOURCE CODE"
20100 FOR Q = 1 TO FMAX
20110 IN$ = FILE$(ASC(MID$(PTR$,Q,1))) :
 O$ = ""
20120 Q1 = AD
20130 GOSUB 27600
20140 IF ERR THEN 20250
20145 IF EO THEN 20222
20150 H = Q1 : GOSUB 11000
20160 Q$ = H$
20180 Q2 = 3 : IF LEN(O$)<Q2 THEN Q2 = L
EN(O$)
20185 Q1$ ="" : IF O$="" THEN 20221
20190 FOR Q3 = 1 TO Q2
20200 H = ASC(MID$(O$,Q3,1)) : GOSUB 110
00
20210 IF LEN(H$)=1 THEN H$ = "0"+H$
20220 Q1$ = Q1$+H$ : NEXT Q3
20221 PRINT Q$ SPC(6-LEN(Q$)) Q1$ SPC(8-
LEN(Q1$)) ; : GOSUB 28100
20222 IF NOT AM OR O$="" THEN 20250
20225 FOR X = 1 TO LEN(O$) : POKE Q1+X-1
,ASC(MID$(O$,X,1)) : NEXT
20250 IF EXIT THEN Q = FMAX+1 : REM LEAV
E LOOP
20260 NEXT Q
20270 PRINT : PRINT " TOTAL ERRORS IN FI
```

```
LE ---" EC : PRINT
20280 IF SY THEN GOSUB 26900
20290 IF PEEK(152) <> 0 THEN PRINT#2 : C
LOSE2 : GOTO 20300
20295 GET T$ : IF T$="" THEN 20295
20300 RETURN
READY.
```

EXAMPLE ERRORS: Error only listing

```
ADD.   DATA      SOURCE CODE
                 50 LBL000 LDQ $A000
========================^
   LABEL DEFINED TWICE ERROR
                 60 JSR ($300)
====================^
   ADDRESSING MODE NOT AVAILBLE WITH THI
S OPCODE ERROR
                 70 LDA #LBL000/H
==============================^
   DIVISION BY ZERO ERROR
                 80 LDX #LBL000-LBL000/256
@256
========================================^
   INVALID OPERATOR ERROR
                 140 BCC LBL000
============================^
   BRANCH OUT OF RANGE ERROR
                 150 JMP LBL001
============================^
   UNDEFINED LABEL ERROR
                 150 JMP LBL001
============================^
   UNDEFINED LABEL ERROR
                 160 LBL000 RTS
======================^
   LABEL DEFINED TWICE ERROR

 TOTAL ERRORS IN FILE --- 8


H                 0
LBL000            C800
 TOTAL NUMBER OF SYMBOLS --- 2
```

EXAMPLE ERRORS: Full listing

```
ADD.   DATA      SOURCE CODE
0                10 PRT
0                20 SYM
0                30 ORG $C800
C800             40 H = 0
                 50 LBL000 LDQ $A000
=======================^
   LABEL DEFINED TWICE ERROR
C800             50 LBL000 LDQ $A000
                 60 JSR ($300)
====================^
   ADDRESSING MODE NOT AVAILBLE WITH THI
S OPCODE ERROR
                 70 LDA #LBL000/H
==============================^
   DIVISION BY ZERO ERROR
                 80 LDX #LBL000-LBL000/256
@256
========================================^
   INVALID OPERATOR ERROR
C804   8543      90 STA &103
C806   8642      100 STX &102
C808   60        110 RTS
C809             120 ORG $CA00
CA00   18        130 CLC
                 140 BCC LBL000
============================^
   BRANCH OUT OF RANGE ERROR
                 150 JMP LBL001
============================^
   UNDEFINED LABEL ERROR
                 150 JMP LBL001
============================^
   UNDEFINED LABEL ERROR
                 160 LBL000 RTS
=======================^
   LABEL DEFINED TWICE ERROR
CA06             160 LBL000 RTS

 TOTAL ERRORS IN FILE --- 8

H                 0
LBL000            C800
 TOTAL NUMBER OF SYMBOLS --- 2
```

In previous sections of the overall program we have adopted the approach of first explaining all the modules which are necessary to make a control module work before entering the control module itself. To do that in the case of the Assembler would result in scores of pages of explanations before any picture could be built up of what the program is setting out to do. The sheer complexity of the Assembler dictates that we adopt a 'top down' approach and attempt to work our way from a simple explanation of the working of the program to a detailed examination of the full listing, filling in details all the time. It is for that reason that we begin our commentary on the main part of the Assembler with this main control module. The module on its own is totally helpless, it does almost no work itself but simply allocates work between various other sections of the program. Nevertheless, commenting on it at this early stage will help to give us a much needed overview of the Assembler's functioning.

Commentary

20010-20029: The Assembler is capable of compiling a machine code program in four different ways. It can provide a full listing of the assembly language instructions, together with a notification of any errors present or it can skip the listing and provide only the errors. An example of both of these was provided at the end of this module. It can also be directed to place the machine code program resulting from the assembly into memory or it can be told to run through the program but leave the memory untouched. If, for instance, you wish to place a machine code routine in a specific area of memory between a specified start point and a specified finish point, without corrupting any of the memory outside those points, you would do well to ask first for a full listing without the program being placed into memory. This will show exactly where the assembled machine code *would* have been placed in memory before anything irrevocable is done.

20030-20085: Before starting work on the assembly language program the variable AD is set to zero, signifying that the address at which the eventual machine code program will start is zero. During the course of the assembly language program this start point will in almost every case be reset to some other point in the memory using the assembler directive 'ORG' (origin). The assembler will now work through the program twice in what are called 'passes'. This loop calls up the section of the program which performs 'Pass 1'. During Pass 1 any variables (including a special kind of variable called a 'label', which defines the correct destination in the memory for a jump instruction) are examined and placed, together with their associated values, into a table known as the 'symbol table' which will be used in the later assembly of the program.

Each line of the assembly language program is obtained in IN$ before Pass 1 is executed upon it. On returning from the execution of Pass 1 on any particular line, a test is made of the variable EXIT, which is set to TRUE if the assembler directive END is met with during the examination of the program. At this point the assembly will cease, even if the end of the file in FI$ has not been reached. At the conclusion of the loop the FRE function is called, thus ensuring that garbage collection is done and there are no dangers of running out of memory.

20090-20300: This is the loop which controls the second pass through the program to be assembled. The routine for the second pass is called up at line 20130. During the second pass the program will actually be assembled, with each valid instruction being translated into the bytes necessary to represent the opcode and operand in machine code, including the translation of variables into values and the assignment of values to labels used for jumps. On returning from the routine a test is made of the variable ERR which records whether an error has been detected. If so, no further processing is done on the instruction. If an error only listing has been specified (EO is set to TRUE) the print routine is omitted. In lines 20150-20221 the information returned from the second pass is printed out in a formatted layout. It includes the address at which an instruction will be placed in the memory if the program is in fact assembled to memory, the hex representation of the 1,2 or 3 bytes that will be placed there and the original assembly language instruction. The bytes of the necessary machine code instruction are contained in O$ and in lines 20222-20225. Provided that AM is TRUE, these bytes are placed into memory beginning at the appropriate location. A test is made that the END directive has not been found and the next line of the program is picked up if not. Finally, if the assembler directive SYM has been encountered during the course of the program the variable SY will be set to TRUE and the symbol table will be printed out at the end of the listing of the assembled program

CHECKSUM TABLE

```
20000 123     20001 180     20002 123
20005 3       20010 227     20020 195
20025 197     20029 189     20030 144
20040 37      20050 139     20060 169
20070 214     20080 243     20085 167
20090 244     20100 37      20110 139
20120 249     20130 172     20140 116
20145 30      20150 213     20160 211
20180 109     20185 40      20190 175
20200 5       20210 221     20220 244
```

```
20221 86      20222 58      20225 198
20250 6       20260 243     20270 10
20280 236     20290 63      20295 181
20300 142
```

SECTION 2: Pass One Routines

MODULE 4.3

```
26400 REM*********************************
26401 REM DO PASS 1 ASSEMBLY ON IN$
26402 REM*********************************
26405 PRINT "[HOME][22*CD][40 SPACES]
                                          " ; 
26406 PRINT "                              
            " ;
26407 PRINT "[HOME][22*CD]" ;: GOSUB 281
00
26410 PASS = 1 : EXIT = FALSE : PTR = 2
26420 GOSUB 28850
26430 IF NOT ERR THEN 26540
26440 IF T=58 AND LEN(H$)=0 THEN 26420
26450 IF T=59 OR T=-1 THEN RETURN
26460 GOSUB 28700
26480 GOSUB 28850
26490 IF NOT ERR THEN 26540
26500 IF T=58 AND LEN(H$)=0 THEN 26420
26520 RETURN
26540 IF PO>55 THEN GOSUB 26600 : GOTO 2
6556
26550 GOSUB 26100
26552 GOSUB 26300 : IF ERR AND OP>3 AND
OP<7 THEN OP = OP+6 : GOTO 26552
26555 GOSUB 26560
26556 IF LEN(IN$)>PTR AND NOT EXIT THEN
26420
26557 RETURN
```

Having examined the overall control module for the two passes we now turn to the control module for Pass 1. Once again we shall leave until later an explanation of how the specific tasks necessary are actually accomplished and concentrate on achieving an overview of what the pass actually does.

Commentary

26400-26407: These lines clear the bottom two lines on the screen and print there the assembly language instruction currently being processed, purely to guide the user as to how far the pass has progressed.

26410: The variable PTR indicates where the examination of the current line will begin. It is set to 2 in order to skip over the two bytes containing the line number.

26420: The routine which identifies the mnemonic using the table in TA$(2) is called. The routine will scan IN$ from the character after that indicated by PTR until it finds a character which is not a letter or a digit, then return.

26430-26520: On return from the previous GOSUB, H$ will contain a string of characters which were terminated by a space, colon or any other character which is not a letter or digit. If, on return from the previous GOSUB a valid mnemonic for an opcode has not been found and placed into H$, line 26440 tests whether any characters at all were found before the colon separator between instructions on the same line (or any other character which is not a letter or digit). If not, the colon is ignored and the subroutine to find a menomonic is called again, starting after the colon.

In 26540 a test is made to see whether a semi-colon or the end of line has been reached. If so no further action is taken in respect of the current line. Comments may thus be entered in a program without confusion, provided that they commence with a semi-colon. If H$ does contain characters then, since they have not been identified as a mnemonic they are assumed to be a label of some kind and the name is entered into the symbol table by the subroutine at 28700. On return from placing the assumed label into the symbol table the program goes back to searching for the opcode mnemonic which should follow the label.

26540: At this point in the program a valid mnemonic must have been detected. If its position in the tables, as indicated by the variable PO, is greater than 55 then it is an assembler directive and the subroutine at 26600 is called up to evaluate it.

26550: Having found a valid opcode mnemonic at this point, the subroutine which evaluates the operand type is called.

26552: We now have a valid mnemonic and, hopefully, a valid operand type. However there is no guarantee that the particular operand type is appropriate to the particular opcode. This line calls the subroutine which established whether they do in fact make a valid pair. The subroutine, if it comes across an address which could accessed by a zero page addressing mode, will assume that zero page addressing mode *is* being used, even though this may result in an error being flagged because the particular

opcode cannot use zero page addressing. On return from the subroutine, if the addressing mode selected is zero page and a mismatch is indicated, the operand type represented by the variable OP is incremented by 6 to transform it into an absolute addressing mode.

26555: In assessing the assembly language instruction there is always a need to keep a count of how many bytes the assembled instruction will require so that the next instruction will commence at the correct place in memory. This is accomplished by the subroutine at 26560.

26556: If the line is not exhausted by what has so far been analysed and END has not been detected, the rest of the line is processed in the same way.

CHECKSUM TABLE

```
26400 123     26401 2       26402 123
26405 225     26406 56      26407 227
26410 255     26420 180     26430 68
26440 84      26450 102     26460 174
26480 180     26490 68      26500 84
26520 142     26540 34      26550 166
26552 190     26555 176     26556 247
26557 142
```

MODULE 4.4

```
28100 REM*********************************
28101 REM PRINT IN$ TO THE SCREEN
28102 REM*********************************
28120 PRINT 256*ASC(MID$(IN$,2,1))+ASC(M
ID$(IN$,1,1)) MID$(IN$,3)
28140 RETURN
```

This module simply prints out the current line of the assembly language program, including its line number. This is printed to the bottom of the screen on the first pass. In the second pass the assembler directive PRT can be included in the program to send the output to a printer.

CHECKSUM TABLE

```
28100 123     28101 187     28102 123
28120 80      28140 142
```

MODULE 4.5

```
28150 REM*******************************
28151 REM SYMBOL TO NON-LETTER/DIGIT
28152 REM*******************************
28160 H$ = "" : T = -1
28165 PTR = PTR+1
28170 IF PTR>LEN(IN$) THEN 28210
28180 T = ASC(MID$(IN$,PTR,1))
28185 IF T=32 AND LEN(H$)=0 THEN 28160
28190 IF T<48 OR T>90 OR ( T>57 AND T<65
 ) THEN 28210
28200 H$ = H$+CHR$(T) : GOTO 28165
28210 RETURN
```

This simple module scans the line in IN$ from the point indicated by the variable PTR, making up a string, H$, from which any leading spaces are stripped and which ends whenever a character which is not a letter or digit is encountered. The module is used by the following module to return a string which may contain an opcode mnemonic or a label.

CHECKSUM TABLE

```
28150 123     28151 240     28152 123
28160 62      28165 185     28170 5
28180 178     28185 79      28190 179
28200 9       28210 142
```

MODULE 4.6

```
28850 REM*********************************
28851 REM TEST FOR OPCODE/DIRECTIVE
28852 REM*********************************
28860 GOSUB 28150
28870 ERR = FALSE
28890 PTR = PTR-1
28895 IF LEN(H$)<>3 THEN 28940
28900 PO = -2
28910 PO   PO+3
```

```
28920 IF H$=MID$(TA$(2),PO,3) THEN 28950
28930 IF (PO+3)<=LEN(TA$(2)) THEN 28910
28940 ERR = TRUE
28950 PO = (PO-1)/3
28960 ERR = (PO=56) OR ERR
28970 IF PO>56 THEN PO = PO-1
28980 RETURN

READY.
```

Having obtained a string of characters which may or may not contain a valid opcode or label we now begin the process of actually evaluating what we have found. The rough method of searching for the correct opcode was described under Module 1 and the process begins with this module.

Commentary

28890: If the string returned in H$ is not three characters long then it cannot be a valid opcode and there is no point in searching the tables.

28895: PTR is reset to point to the last character of H$ in the line from which it was drawn.

28900-28940: This is a loop which scans TA$(2) — which contains the valid three letter mnemonics — to compare each set of characters with what has been picked up in H$. The pointer for this purpose is first set to -2 in order that on the first iteration of the loop, when 3 is added, the search starts at one. If the whole loop is executed and line 28940 reached, it can only be because the three characters do not conform to any of the mnemonics for opcodes specified in the table.

28950: The address in TA$(2), which was incremented in steps of three, is now adjusted so that, for instance, the three characters at position 19-21 are now identified in PO as mnemonic 7.

28960: This strange looking line is here to take account of the fact that the Disassembler tables on which the Assembler works contain the three characters '???' to cope with times when the Disassembler cannot provide a valid opcode for what is in the memory. Without this check you would be able to enter '???' into an assembly language program and throw the whole thing into confusion when it was found as a valid menomonic. The line flags an error if H$ consists of '???'.

28970: This line is also there to take account of the '???' in the table. The assembler directives fall *after* the question marks and so, when numbering them for the purposes of the assembler, their position is reduced by one.

CHECKSUM TABLE

```
28850 123     28851 158/    28852 123
28860 173     28870 70      28890 186
28895 176     28900 110     28910 13
28920 57      28930 24      28940 27
28950 61      28960 193     28970 89
28980 142
```

MODULE 4.7

```
28700 REM*********************************
28701 REM ADD SYMBOL TO SYMBOL TABLE
28702 REM*********************************
28710 IF SE>=SM THEN EXIT = TRUE : PASS
= 2 : EN = 14 : GOTO 28000
28720 GOSUB 28250 : IF NOT ERR THEN 2883
0
28740 T$ = LEFT$(H$+"          ",6)
28745 TB = PTR
28750 GOSUB 28150 : REM DOES = FOLLOW
28760 IF T<>61 THEN PTR = TB : RE = AD :
 GOTO 28780
28770 T0 = T : GOSUB 28600
28780 EN = 0
28790 IF RE<0 OR RE>65535 THEN ST$(SE)=T
$+CHR$(0)+CHR$(0)+CHR$(2) : GOTO 28810
28800 ST$(SE) = T$+CHR$(RE-INT(RE/256)*2
56)+CHR$(INT(RE/256))
28810 SE = SE+1
28820 RETURN
28830 IF PASS=1 AND LEN(ST$(T1))<9 THEN
ST$(T1) = ST$(T1)+CHR$(8)
28835 IF PASS<>2 THEN 28840
28836 TA = PTR : GOSUB 28150 : IF T<>61
THEN PTR = TA : GOTO 28840
28837 GOSUB 26000 : REM SCAN PAST = SIGN
 (IF PRESENT) ON PASS 2
28840 IF PASS<>2 OR LEN(ST$(T1))<9 THEN
RETURN
28845 EN = ASC(MID$(ST$(T1),9,1)) : GOTO
 28000
READY.
```

This is the control module for evaluating variables and labels and placing them into the symbol table if they have not already been defined. Ear-

lier, under Module 2, a label was described as a kind of variable for the sake of simplicity. In fact a label is a constant which identifies a point in a machine code program to which a jump may be made. By using labels it becomes possible to assemble a machine code program to a different place in the memory without having to recalculate the addresses to which jumps will be made. The Assembler will automatically identify the position in memory of a labelled instruction and replace a jump to the label with a jump to that address.

Commentary

28710: If there are more symbols than the symbol table is set up for (50), then assembly of the program ceases immediately and the appropriate error is flagged.

28720: The label in H$ is now sent to the next module, which examines the symbol table (ST$) to see whether it is already present. If it is present then ERR is returned as false, and an error 'label defined twice' will be flagged. This reverse use of error may seem confusing but is necessary since a later use of the module at 28250 will require an error to be flagged if a label is *not* in the symbol table.

28740: H$ is padded out to six characters if it is shorter than that. Six characters is the maximum for any label.

28745-28760: The content of H$, if it is valid at all, may be either a variable or a label. If it is a label then all the Assembler will need to know is the address of the instruction so labelled. If it is a variable it will be followed by an ' = ' to set its value. The scanning module at 28150 is called up to get the next character. If the next character after the variable (ignoring spaces) is not an equals sign then it must be a label and RE (REsult) is used to store the current address of the instruction. PTR is backed up to the end of the label again using the temporary variable TB.

28870: If the contents of H$ *do* contain a variable then the 'expression evaluator' section of the program is now called up. No attempt will be made to explain the working of the expression evaluator until the end of the program since it would interrupt our attempt to follow through the main workings of the Assembler. For the moment all that you need to accept is that a call to the expression evaluator for a line like VAR = 256*BYTE1 + 15 would return in the variable RE the result of the right hand part of the equation. All will ultimately be made clear!

28790: If RE is less than zero or greater than 65535 (the maximum value which can be dealt with by the CPU in one instruction) then to the variable name or label in the symbol table are added two characters which represent a result of 0 and another which flags an error 'double byte out of range'.

28800: If RE is a valid number then the number is added in two byte form to the end of the variable or label name in ST$. Since the name has been set to 6 characters in every case it will be simple to recover the value of RE for any variable or label.

28830: This line is only accessed if the current name is already in the symbol table. Provided that an error code is not already attached to the name, it has error code 8 added to it, indicating 'label defined twice'.

28835-28837: This module is also used by the second pass through the program. On pass 2 the name will already be in the symbol table (placed there by pass one) and so the first part of the module will not be carried out. On pass 2, having obtained the name in H$, if the next character is an equals, then the result of the expression has already been obtained and the module at 26000, which finds the end of an expression or line, is called to skip over the rest of the expression.

28840: Error messages are only printed on pass 2 and, fairly obviously, only if there is an error code attached to the end of a name in the symbol table.

CHECKSUM TABLE

```
28700 123     28701 175     28702 123
28710 239     28720 112     28740 31
28745 126     28750 81      28760 227
28770 241     28780 181     28790 40
28800 251     28810 253     28820 142
28830 106     28835 101     28836 117
28837 160     28840 126     28845 55
```

MODULE 4.8

```
28250 REM*********************************
28251 REM FIND LABEL IN ST$
28252 REM*********************************
28260 ERR = FALSE : H = 0 : T1 = 0
28270 IF LEN(H$)<6 THEN H$ = H$+" " : GO
TO 28270
28280 IF T1=SE THEN ERR = TRUE : RETURN
28290 IF MID$(ST$(T1),1,6)<>H$ THEN T1 =
 T1+1 : GOTO 28280
```

```
28295 H = ASC(MID$(ST$(T1),8,1))*256+ASC
(MID$(ST$(T1),7,1)) : RETURN
```

This a straightforward module which is called by the previous one to determine whether a variable or label name is already in the symbol table. The module returns an error flag of 1 or 0 to indicate the presence or absence of the variable or label. Also returned, if present is the result contained in characters 7 and 8.

CHECKSUM TABLE

```
28250 123     28251 242     28252 123
28260 75      28270 249     28280 132
28290 219     28295 92
```

MODULE 4.9

```
28000 REM*********************************
28001 REM ASSEMBLER ERROR ROUTINE
28002 REM*********************************
28005 IF PTR>=300 OR PASS<>2 THEN 28050
: REM SUPRESS SECONDARY ERRORS IN LINE
28010 PRINT SPC(14) ; : GOSUB 28100
28015 EC = EC+1
28020 FOR X = -13 TO PTR : PRINT "=" ;:
NEXT X : PRINT "[CU]"
28030 PRINT "    " ERR$(EN) " ERROR"
28040 PTR = 300 : ERR = TRUE
28050 RETURN
```

This module, which is called by line 28840 in Module 4.7, prints out an error message where an error is flagged.

Commentary

28005: If an error has already been notified for the current line, the variable PTR is set to the impossible value of 300 (maximum length of a string is 255) and no subsequent error messages are printed for that line. Errors are not printed on pass 1.

28010: On the second pass, the memory address and byte data take 14 spaces on the line. When a line is printed and an error is to be flagged the address and data are not printed.

28020-28030: The error is not only flagged by an error message in the output, a pointer is printed at the approximate position in the line where the error has been detected, as indicated by the value of PTR.

CHECKSUM TABLE

```
28000 123     28001 61      28002 123
28005 92      28010 106     28015 221
28020 12      28030 124     28040 16
28050 142
```

MODULE 4.10

```
26000 REM*******************************
26001 REM SYMBOL UP TO COLON ETC.
26002 REM*******************************
26010 H$ = "" : T1 = LEN(IN$)
26020 PTR = PTR+1
26030 IF T1<PTR THEN 26060
26040 T = ASC(MID$(IN$,PTR,1))
26045 IF T=32 THEN 26020
26050 IF T<>58 AND T<>59 THEN H$ = H$+CH
R$(T) : GOTO 26020
26060 RETURN
```

This module is similar to the module at 28150 (Module 5), its purpose being to determine the end point of an assembler instruction. It is different from Module 5 in that it does not return on finding a non-letter/non-digit character but only on finding a delimiter such as a colon, semi-colon or the end of line, dropping any spaces which are present in the original line.

CHECKSUM TABLE

```
26000 123     26001 210     26002 123
26010 98      26020 185     26030 190
26040 178     26045 247     26050 201
26060 142
```

MODULE 4.11

```
26600 REM*******************************
26601 REM CALCULATE DIRECTIVE LENGTH
26602 REM*******************************
26610 T1 = LEN(IN$)
26620 IF PO=56 THEN 26720 : REM BYT DIRE
CTIVE
26625 IF PO=60 THEN GOSUB 28600 : AD = R
ESULT : REM DEAL WITH ORG DIRECTIVE
```

```
26627 IF PO=59 THEN EXIT = TRUE
26630 IF PO>58 THEN RETURN : REM END & O
RG DIRECTIVES
26640 REM FIND LEN. OF WRD & DBY
26650 AD = AD+2
26660 PTR = PTR+1
26670 IF PTR>T1 THEN RETURN
26680 T = ASC(MID$(IN$,PTR,1))
26690 IF T=58 OR T=59 THEN RETURN
26700 IF T<>46 THEN 26660
26710 GOTO 26650
26720 REM LENGTH FOR BYT.
26730 AD = AD+1
26740 PTR = PTR+1
26750 IF PTR>T1 THEN RETURN
26760 T = ASC(MID$(IN$,PTR,1))
26770 IF T=58 OR T=59 THEN RETURN
26780 IF T<>46 THEN 26740
26790 GOTO 26730
```

This module is used exclusively by Pass 1 and acts upon those assembler directives which are relevant to that part of the program execution, these being BYT, WRD, DBY, END, ORG. The module is called from Module 3 whenever an assembler directive is detected.

Commentary

26620: This deals with the BYT directive and will be discussed under 26720.

26625: 60 is the code for the ORG directive, which is used to set the address in memory on which subsequent assembly will be based. ORG will be followed on the line by an expression and the 'expression evaluator' (not explained yet) is called to get the desired address from the expression. AD, the address at which the next machine code byte will be placed is then set equal to the result of the expression.

26627: If the END directive is encountered the variable EXIT is set to TRUE.

26630: Other than ORG and END, none of the directives with a code greater than 58 (SYM, PRT) affect program execution on the first pass.

26640-26710: At this point in the program execution, what has been encountered must be one of the two directives WRD and DBY. These take the form in the assembly language program:
WRD (or DBY) $AAAA.$BBBB.$CCCC ie the directive followed by a series of two byte values which will be placed directly into memory — thus

allowing a table to be defined.

The difference between the two directives is that WRD will take the two bytes specified by, for instance $ABCD and store them in the memory in the order CD, AB while DBY will store them in the order AB, CD. The CPU chip normally works upon two byte numbers where the least significant byte (CD in our example) comes first. The problem with these two directives on the first pass is that, as we have previously noted, a record must be kept of the length in bytes of each instruction in order that labels may be given their correct addresses in the memory when they are defined on the first pass. BYT and WRD can have any number of two byte values after them up to the maximum string length of 255. This loop therefore scans the line, counting the number of two byte values and incrementing the address counter AD by two for each value found.

26720-26790: The BYT instruction specifies single byte values to be placed into memory. This loop performs the same function as the previous one but only increments the address counter by one for each value specified.

CHECKSUM TABLE

```
26600 123     26601 222     26602 123
26610 70      26620 38      26625 40
26627 189     26630 61      26640 4
26650 216     26660 185     26670 76
26680 178     26690 247     26700 184
26710 172     26720 181     26730 215
26740 185     26750 76      26760 178
26770 247     26780 183     26790 171
```

MODULE 4.12

```
26100 REM*********************************
26101 REM OPERAND TYPE TO BE USED
26102 REM*********************************
26110 T6 = PTR : GOSUB 26000
26120 ERR = FALSE
26130 IF LEN(H$)=0 THEN OP = 1 : RETURN
26140 IF H$="A" THEN OP = 0 : RETURN
26145 IF ASC(H$)=35 THEN OP = 2 : RETURN
26170 OP = 12
26180 IF LEFT$(H$,1)="(" THEN OP = OP-3
26190 T = 1 : T1 = LEN(H$)
26200 T2 = ASC(MID$(H$,T,1))
26210 IF T2<>46 AND T<T1 THEN T = T+1 :
GOTO 26200
```

```
26220 IF T2<>46 THEN 26275
26230 T = T+1 : IF T>T1 THEN 26270
26240 T2 = ASC(MID$(H$,T,1))
26250 IF T2=89 THEN OP = OP-1 : GOTO 262
75
26260 IF T2=88 THEN OP = OP-2 : GOTO 262
75
26270 REM NOT A VALID INDEX
26272 EN = 5 : GOTO 28000
26275 IF(OP=12)AND((PO>2ANDPO<6)OR(PO>6A
NDPO<10)ORPO=12ORPO=11)THEN OP = 3
26281 REM ZERO PAGE OPRANDS
26282 IF OP<10 THEN RETURN
26284 T7 = PTR : PTR = T6
26286 GOSUB 28600
26288 IF ERR OR RESULT>255 THEN 26292
26290 OP = OP-6
26292 PTR = T7
26294 RETURN
```

Following on the development of Pass 1, as laid down by the control module, we now come to the two routines which determine the type of operand to be used and whether that operand type actually fits the opcode type, remembering that not every addressing mode can be used by every opcode type. The purpose of the present module is to determine the operand type which fits the format laid down in the instruction.

Commentary

26110: At this point PTR is indicating the character after the opcode mnemonic and the subroutine at 26000 is called to obtain the operand part of the instruction, stripped of its spaces.

26130: If the operand has zero length then the addressing mode must be implied, and the value of OP is zero.

26140: If the operand is 'A' then the addressing mode is accumulator addressing, OP = 1.

26145: If the first character of the operand is '#' then the addressing mode is immediate, OP = 2.

26150-26160: The temporary assumption is now made that the operand type is 3, relative addressing mode. The following subroutine is called to see if the tables in TA$ indicate that this is possible for the opcode that has been derived (ie the opcode must be a branch of some kind). The method by

which this is done will be described under the next module. The references to O$ are only relevant to Pass 2. The subroutine at 26300 will place into O$ the byte whose value represents the relevant opcode.

At this point, however, we are only using the subroutine to determine whether the operand type is 3 — we do not wish to increment O$ at this point so we take a note of its length before the subroutine is called and then reset it to the same length on return. If an error is flagged on return from the subroutine at 26300 then operand type three does not fit the opcode we have and the subroutine continues.

26170-26260: These lines are a mirror image of the lines in the Disassembler which work out the format of the operand from the value of the operand byte in a machine code instruction. In this case we work out the operand type from examining the format.

26270-26280: If a full stop is detected and the format does conform to that for indexed addressing, error 5 is flagged — 'index is not X or Y'.

26282-26294: These lines test whether it is possible to perform the instruction with a zero page addressing mode — the format for absolute addressing and zero-page addressing is the same and the assumption has been made up to this point that operands which could be either are in fact absolute addresses. This is only possible with operand types of 10 and above, representing the absolute addressing modes. The PTR is reset to the end of the opcode and the operand re-evaluated by the Expression Evaluator. If the result falls in the range 0-255 then it is possible to use the faster zero page addressing mode and the operand type OP is reduced by 6 to transform the addressing mode into zero page addressing of some kind.

CHECKSUM TABLE

```
26100 123     26101 213     26102 123
26110 145     26120 70      26130 190
26140 254     26145 250     26170 244
26180 160     26190 232     26200 243
26210 243     26220 236     26230 12
26240 243     26250 112     26260 112
26270 41      26272 215     26275 213
26281 99      26282 211     26284 95
26286 173     26288 156     26290 17
26292 115     26294 142
```

MODULE 4.13

```
26300 REM*********************************
26301 REM EVALUATE OPCODE
26302 REM*********************************
26310 T1 = 3 : T = PO
26320 T = ASC(MID$(TA$(T1),T+1,1))
26330 IF T=255 THEN ERR = TRUE : RETURN
26340 T1 = 4 : T2 = ASC(MID$(TA$(1),INT(
T/2+1),1))
26350 IF (1 AND T)=0 THEN T2 = INT(T2/16
)
26355 T2 = T2 AND 15
26360 IF T2<>OP THEN 26320
26370 O$ = O$+CHR$(T)
26380 ERR = FALSE
26390 RETURN
```

At this point we again make full use of the new tables which were added to TA$ in the first module of the Assembler. The purpose of the module is to match the opcode that has been obtained with the operand type and see whether they are in fact compatible. If not an error must be flagged.

Commentary

26320: T has been set equal to PO, the position of the opcode mnemonic in TA$(2) and thus the string equation in this line points to a pair of bytes in TA$(3). TA$(3) contains, for each of the possible opcodes types, the first of the possible byte forms that the opcode can take. The value is also the first link in the chain of possible byte forms of the opcode.

26330: If, on subsequent iterations, the value found in the table (TA$(4) subsequently) is FF hex, then there are no more forms of the opcode available and an error is flagged.

26340-26355: Having found the possible opcode, it is now compared with the necessary addressing mode in TA$(1). The addressing modes for each opcode are stored in TA$(1), two to a character. A single character can be used to store two numbers in the range 0-15, simply by multiplying one of the numbers by 16 and then adding them together. Thus the addressing mode for opcode one in the table of opcodes will be found in the first character of TA$(1), as will the addressing mode for opcode two. Lines 26350 and 26355 extract the necessary half of the character value (0-15 and 16-255). If the opcode position (PO) in the table is odd, then the high half of the byte is used (T2/16) and if PO is even the low half of the byte is used (T2 AND 15).

26360: If the resulting addressing mode is not the same as that obtained by examining the operand of the instruction in assembly language, then the subroutine returns to 26320 and picks up the next possible form of the opcode, together with its associated addressing mode and so on until there are no further forms of that opcode mnemonic.

26370: If program execution has reached this point it is because an addressing mode has been found in the tables which conforms to the format of the operand picked up from the assembly language instruction. The correct opcode for the operand and the opcode type is added to O$, which is being used to store what will eventually be placed into memory, though this is only relevant on Pass 2.

CHECKSUM TABLE

```
26300 123     26301 224     26302 123
26310 9       26320 191     26330 87
26340 167     26350 81      26355 83
26360 24      26370 238     26380 70
26390 142
```

MODULE 4.14

```
26560 REM*********************************
26561 REM BYTE LENGTH
26562 REM*********************************
26565 AD = AD+1
26570 IF OP>1 THEN AD = AD+1
26580 IF OP>8 THEN AD = AD+1
26590 RETURN
```

We finish following through the work of the Assembler on Pass 1 with this short module. It simply uses the opcode type to determine how many bytes the assembled instruction will require when it is finally placed into memory on Pass 2. This is in order that the variable AD may be correctly updated for the purpose of defining labels.

CHECKSUM TABLE

```
26560 123     26561 197     26562 123
26565 215     26570 234     26580 241
26590 142
```

SECTION 3: Pass Two Routines

MODULE 4.15

```
27600 REM*********************************
27601 REM DO PASS 2 ASSEMBLY
```

```
27602 REM*********************************
27605 PASS = 2
27610 O$ = ""
27620 EXIT = FALSE : ERR = FALSE
27625 PTR = 2
27630 GOSUB 28850
27640 IF NOT ERR THEN 27720
27650 IF T=58 AND LEN(H$)=0 THEN 27630
27660 IF T=59 OR T=-1 THEN ERR = FALSE :
 RETURN
27665 GOSUB 28700
27670 GOSUB 28850
27680 IF NOT ERR THEN 27720
27690 IF T=58 AND LEN(H$)=0 THEN 27630
27695 IF T=59 OR T=-1 THEN ERR = FALSE :
 RETURN
27700 EN = 3 : GOTO 28000
27720 IF PO>55 THEN GOSUB 27200 : GOTO 2
7745
27723 T5 = PTR : GOSUB 26100 : T8 = PTR
 : PTR = T5
27725 GOSUB 26300 : IF NOT ERR THEN 2773
0
27727 IF OP<7 AND OP>3 THEN OP = OP+6 :
PTR = T5 : GOTO 27725
27728 EN = 18 : GOTO 28000
27729 REM THIS BIT ATTEMPS TO MATCH ABSL
OUTE ADD MODE TO OPCODE IF ZP HAS FAILED
27730 GOSUB 26560
27740 IF NOT ERR AND LEN(O$)>0 THEN GOSU
B 27000 : PTR = T8
27745 IF LEN(IN$)>PTR AND NOT EXIT THEN
27630
27750 RETURN
```

Having worked through Pass 1, we now turn our attention to Pass 2. As with Module 4.3, this is the control module of the Pass and we shall follow through the Pass in outline before examining it in detail.

Commentary

27605-27700: Apart from setting the output string (O$) to empty and PASS equal to 2, these lines are similar to the first part of the Pass 1 module in their effect. A test is made for an opcode, if this fails it is assumed that the first part of the line is a label or variable. Following this another search is

made for an opcode and if the line is not setting a variable equal to something and there is no opcode present, error 3 'invalid operand or opcode' is flagged.

27720: If the opcode type is greater than 55 then an assembler directive has been encountered and the relevant module is called up to evaluate it.

27723: The subroutine at 26100 is now called to evaluate the operand.

27725-27728: The subroutine at 26300 examines the match between the opcode and addressing mode so far obtained, then attempts to match them in absolute mode, flagging error 18, 'addressing mode not available with this opcode' if the match is not correct.

27730: The byte counter AD is incremented by this call.

27740: If no error has been found and there is something in O$ then the operand part of the instruction is actually evaluated. T8 is a variable used to move the pointer past an index such as '.X' at the end of the operand, since the operand evaluator will not scan past these. T8 was set in line 27723, when returning from the routine which evaluates the operand type, which scans to the very end of the operand, including any index attached.

27745: This line allows multiple statements on the same line to be evaluated.

CHECKSUM TABLE

```
27600 123     27601 107     27602 123
27605 91      27610 169     27620 87
27625 26      27630 180     27640 69
27650 88      27660 38      27665 174
27670 180     27680 69      27690 88
27695 38      27700 213     27720 32
27723 138     27725 104     27727 178
27728 11      27729 248     27730 176
27740 46      27745 251     27750 142
```

MODULE 4.16

```
27200 REM*******************************
27201 REM DIRECTIVE OPERAND EVALUATOR
27202 REM*******************************
27205 ERR = FALSE
27210 IF PO=60 THEN GOSUB 28600 : AD =RE
SULT
27214 IF PO=62 THEN SY = TRUE
```

```
27215 IF PO=61 THEN OPEN 2,4 : CMD 2 : P
RINT "[CD]ADD.  DATA      SOURCE CODE[CD]
"
27220 IF PO=59 THEN EXIT = TRUE
27230 IF PO>58 THEN RETURN
27240 IF PO=56 THEN 27330
27250 REM DBY & WRD DIRECTIVES
27260 GOSUB 28600
27270 IF RESULT<0 OR RESULT>65535 THEN E
N = 2 : GOTO 28000
27280 IF PO=58 THEN RESULT = INT(RESULT/
256)+256*(RESULT-INT(RESULT/256)*256)
27281 REM 27280 REVERSES HI. & LO. BYTES
 IF DIRECTIVE IS DBY
27290 T1 =T : GOSUB 27100 : AD = AD+2
27300 IF T1=32 THEN GOSUB 28150
27310 IF T1=46 THEN 27260
27320 RETURN
27330 REM BYT DIRECTIVE
27340 GOSUB 28600
27350 IF RESULT<0 OR RESULT>255 THEN EN
= 1 : GOTO 28000
27360 GOSUB 27140 : AD = AD+1
27370 IF T=32 THEN GOSUB 28150
27380 IF T=46 THEN 27340
27390 RETURN
```

In Pass 1 we examined a module which determined any actions necessary when an assembler directive was encountered and also the length of the directive if it was BYT, WRD or DBY. This module is similar except that it performs such actions as opening a channel to the printer for output, flags SY to print out the symbol table or places the data from a BYT, WRD or DBY directive into the output string, O$.

Commentary

27210: If an ORG directive is encountered, its operand is evaluated and the byte counter AD set equal to the result.

27214: Entering SYM in the program flags SY to print the symbol table at the end of the listing of source code.

27215: Entering PRT in the program opens an output channel to the printer for the listing of source code, otherwise output is to the screen.

27220: END results in the pass finishing at this point.

27250-27320: The value of a two byte directive (DBY, WRD) is obtained using the Expression Evaluator and the two bytes reversed if the directive is DBY. This is because the routine at 27100, which is now called, places the two bytes into the string O$ in LO/HI order. AD is incremented by 2. Line 27300 scans past any leading spaces and the subroutine loops back to pick up another double byte if a full stop is encountered.

27340-27380: The same routine as above, but for the single byte directive BYT.

CHECKSUM TABLE

```
27200 123     27201 74      27202 123
27205 70      27210 166     27214 41
27215 154     27220 189     27230 221
27240 77      27250 243     27260 173
27270 176     27280 150     27281 31
27290 30      27300 219     27310 52
27320 142     27330 93      27340 173
27350 67      27360 252     27370 170
27380 2       27390 142
```

MODULE 4.17

```
27000 REM*********************************
27001 REM EVALUATE OPERAND
27002 REM*********************************
27010 ERR = FALSE
27020 IF OP<2 THEN RETURN
27030 IF OP=3 THEN 27500
27040 IF OP=2 THEN 27400
27050 GOSUB 28600
27060 IF ERR OR LEN(O$)=0 THEN RETURN
27070 IF (RESULT<0 OR RESULT>255) AND OP
<9 THEN EN = 1 : GOTO 28000
27080 IF RESULT<0 OR RESULT>65535 THEN E
N= 2 : GOTO 28000
27090 IF OP<9 THEN 27140
27100 T = INT(RESULT/256)
27110 RESULT = RESULT-T*256
27120 GOSUB 27140
27130 RESULT = T
27140 O$ = O$+CHR$(RESULT)
27150 RETURN
```

This module evaluates an operand whose type has already been determined, placing the result in one or two byte form into O$.

Commentary

27020: If OP is less than two then there will be no operand.

27030: If the OP is 3 then relative addressing mode is required and the routine at 27500 is called up.

27040: If OP is two then the addressing mode is immediate and the routine at 27400 is called up.

27050-27080: For all other values of OP, the Expression Evaluator is used to obtain a result and this is tested against the requirements of the addressing mode for 1 or 2 bytes.

27090-27140: These are the two routines that place the result obtained through the Expression evaluator into byte form in O$. Note that two byte numbers are placed into O$ with the high byte second.

CHECKSUM TABLE

```
27000 123     27001 47      27002 123
27010 70      27020 164     27030 20
27040 18      27050 173     27060 98
27070 14      27080 144     27090 27
27100 117     27110 248     27120 171
27130 37      27140 121     27150 142
```

MODULE 4.18

```
27500 REM*********************************
27501 REM EVALUATE RELATIVE EXPRESSION
27502 REM*********************************
27510 GOSUB 28600
27520 IF LEN(O$)=0 OR ERR THEN RETURN
27530 RESULT = RESULT-AD
27540 IF RESULT<0 THEN RESULT = RESULT+2
56
27560 IF RESULT<256 AND RESULT>=0 THEN 2
7140
27570 EN = 10
27580 GOTO 28000
```

This module evaluates the operand of an instruction using relative addressing, ie where a jump is specified from the current address up to 127 positions positively in the memory or 128 negatively.

Commentary

27530: Because we are talking about a relative address, a jump is specified first of all as being to an address and then the relative jump is calculated by subtracting the current address, recorded in AD.

27540: Negative jumps cannot be placed straight into the machine code program, they must be transformed into what is known as 'two's complement' form, where any negative value has 256 added to it, so that it ends up as a positive number in the range 128-255. Thus jumps with a value above 128 are in fact negative jumps and their value can be obtained by subtracting 256.

CHECKSUM TABLE

```
27500 123     27501 178     27502 123
27510 173     27520 98      27530 224
27540 75      27560 32      27570 230
27580 163
```

MODULE 4.19

```
27400 REM*******************************
27401 REM EVALUATE IMMEDIATE EXPRESSION
27402 REM*******************************
27410 T5 = PTR : GOSUB 26000
27420 IF ASC(H$)<>35 THEN 27480
27430 IF MID$(H$,2,1)="'" THEN 27450
27440 PTR = T5
27442 IF PTR>LEN(IN$) THEN 27446
27444 IF ASC(MID$(IN$,PTR,1))<>35 THEN P
TR = PTR+1 : GOTO 27442
27446 OP = 8 : GOSUB 27050 : OP = 2
27448 RETURN
27450 REM SINGLE CHR. EXPECTED
27460 IF LEN(H$)<>3 THEN 27480
27470 O$ = O$+MID$(H$,3,1) : RETURN
27480 EN = 12
27490 GOTO 28000
```

This module obtains the value of an operand for an instruction involving immediate addressing, ie where a register is loaded directly with a value in the range 0-255.

Commentary

27410-27420: The operand is obtained in H$ by the use of the routine at 26000. If it does not begin with a '#' then an error will be flagged.

27430: If the second character of the operand is a single quotation mark, then the routine will expect a single character following quote, whose ASCII value will be the value of the operand. A second quote should not be used.

27442-27444: These two lines skip over any spaces in IN$ to the hash sign.

27446: Since immediate addressing always involves a single byte, the routine at 27050 is used to evaluate the operand as if it were addressing mode 8 (indirect Y) and to place the byte into O$. This is simply a short-cut.

27460-27470: These two lines deal with single characters in quotes. If the length of H$ is not 3 then the format is invalid and an error is flagged. If H$ *is* three characters long then the middle character is taken as the one whose ASCII value is to be the operand.

CHECKSUM TABLE

```
27400 123     27401 229     27402 123
27410 144     27420 230     27430 243
27440 113     27442 15      27444 169
27446 43      27448 142     27450 14
27460 174     27470 205     27480 232
27490 163
```

MODULE 4.20

```
26900 REM*********************************
26901 REM DUMP SYMBOL TABLE TO SCREEN
26902 REM*********************************
26910 IF SE<1 THEN 26975
26915 PRINT
26920 FOR X = 0 TO SE-1
26930 PRINT LEFT$(ST$(X),6) TAB(10) ;
26940 H = ASC(MID$(ST$(X),8))*256+ASC(MI
D$(ST$(X),7))
26950 GOSUB 11000
26960 PRINT H$
26970 NEXT X
26975 PRINT "[CD] TOTAL NUMBER OF SYMBOL
S ---" SE
26980 RETURN
```

This module is not truly part of Pass 2: it simply outputs the symbol table at the end of Pass 2 if the SYM directive was present in the assembly language program.

CHECKSUM TABLE

```
26900 123     26901 6       26902 123
26910 27      26915 153     26920 115
26930 80      26940 64      26950 159
26960 37      26970 250     26975 248
26980 142
```

SECTION 4: The Expression Evaluator

Up to now we have had several references to the shadowy beast known as the Expression Evaluator, taking it on trust that there was such an animal and, more importantly, that it does the necessary job. It would, of course be quite possible to write an assembler which required all values to be spelled out in either decimal or hexadecimal but a great deal of time and effort is saved if the user can enter variables, jump to a position six bytes after a particular label or calculate bytes by entering something like LDA #VAL/256. The Expression Evaluator makes all of this possible, as you will discover when you move on to enter our machine code routines.

MODULE 4.21

```
28300 REM*********************************
28301 REM EVALUATE LABEL OR NUMBER
28302 REM*********************************
28320 GOSUB 28150
28325 IF T=40 AND LEN(H$)=0 THEN GOSUB 2
8150
28330 T1 = LEN(H$)
28335 IF (T=-1 OR T=32 OR T=58 OR T=59 O
R T=41 OR T=46) AND T1 = 0 THEN RETURN
28340 IF T1=0 THEN 28390
28350 IF ASC(H$)<=57 THEN H = VAL(H$) :
GOTO 28492
28360 GOSUB 28250 : REM FIND LABEL IN SY
MBOL TABLE
28370 IF ERR THEN EN = 11 : H = 0 : GOTO
 28000
28380 GOTO 28492
28390 REM HEX,OCTAL OR BINARY NUMBERS EV
ALUATE
```

```
28400 T2 = T : GOSUB 28150
28410 IF LEN(H$)=0 THEN 28450
28420 IF T2=36 THEN 28470
28430 IF T2=37 THEN BASE = 2 : GOTO 2847
0
28440 IF T2=38 THEN BASE = 8 : GOTO 2847
0
28450 REM INVALID LABEL
28460 H = 0 : EN = 6 : GOTO 28000
28470 REM TEST IF VALID NUMBER
28475 GOSUB 11950
28480 BASE = 16 : REM DEFAULT BASE
28490 IF ERR THEN H = 0 : EN = 7 : GOTO
28000
28492 PTR = PTR-1 : GOSUB 28150 : REM GE
T NEXT OPERATOR
28495 RETURN
```

CHECKSUM TABLE

```
28300 123     28301 48      28302 123
28320 173     28325 250     28330 247
28335 198     28340 255     28350 206
28360 173     28370 99      28380 178
28390 140     28400 243     28410 247
28420 56      28430 155     28440 162
28450 54      28460 188     28470 23
28475 173     28480 221     28490 56
28492 213     28495 142
```

This module is the core of the Expression Evaluator, its job being to extract the value of number or label, numbers being permitted in decimal, hexadecimal, octal (base 8) or binary (base 2) form. The subroutine returns a value in the variable H and the permissible range of values is integers from 0 to 65535.

Commentary

28320: the number or label is obtained in H$.

28325: If the first character encountered is an open bracket '(', then the routine is called again to obtain the actual number.

28835: On entering the routine at 28150, the variable T is set to minus one. If it remains at that value then no characters of any significance have been found. The other values for T indicate that a space, colon, semi-colon,

close bracket or full stop have been found. If any of these is combined with an H$ of length zero then there is no number or label in the instruction and the program returns from the subroutine.

28340: If none of the delimiters tested for in the previous line is present it is assumed that the number has a '$', '%' or '&' in front of it, indicating hexadecimal, octal or binary and execution goes to the routine which extracts a decimal number from these.

28350: If the first character is a digit then the VAL of H$ is taken — variables must therefore not begin with a number, since the value of the number will be picked up and the rest of the variable name ignored.

28360-28380: If the first character is not a digit then what has been picked up is assumed to be a label and it is sent to the routine at 28250 in order to obtain its value.

28390-28490: The pointer represented by T has now passed a character which is assumed at this point to indicate a different number base. This assumption is now tested. If a different base is specified, the variable BASE is altered to take account of it and the routine in the Monitor which converts non-decimal numbers to decimal is called up. If the character indicated by T2 is not one of the base changing indicators then it is invalid and the 'label is not alphanumeric' error is flagged. If a different base has been specified but the representation is incorrect (eg binary 101012) then an error 'incorrect number base' will be flagged after the return from the routine at 11950.

28492: The main pointer, PTR, which is indicating the character after the end of the current operator, is now backed up one in order that the procedure may be re-executed on the rest of the line.

MODULE 4.22

```
28500 REM*******************************
28501 REM EVALUATE TERM WITH * OR /
28502 REM*******************************
28510 GOSUB 28300 : TERM = H
28520 IF PTR>LEN(IN$) THEN RETURN
28530 IF T=42 THEN GOSUB 28300 : TERM =
INT(TERM*H) : GOTO 28520
28550 IF T<>47 THEN RETURN
28560 GOSUB 28300
28570 IF H=0 THEN TERM = 0 : EN = 15 : G
OTO 28000
28580 TERM = INT(TERM/H)
28590 GOTO 28520
```

One of the problems of evaluating an expression is that of precedence, ie which part of an expression such as A*B/C + D/E*F must be evaluated first. The Expression Evaluator can deal with the precedence between '+', '-', '*' and '/' but not with precedence forced by the use of brackets. Brackets would anyway make it more difficult to assess the operand type. This particular module deals with the two high precedence operations, multiply and divide.

Commentary

28510: A value is picked up using the previous module and stored in the variable TERM.

28530: If the character pointed to by T is a multiply sign then the next value is immediately picked up and multiplied by TERM.

28550-28580: If the character indicated by T is a '/', representing a division sign, then a test is made that the divisor is not zero — if it is the 'division by zero' is flagged. TERM is now divided by the value just obtained in H.

CHECKSUM TABLE

```
28500 123     28501 20      28502 123
28510 150     28520 150     28530 162
28550 67      28560 170    /28570 152
28580 93      28590 170
```

MODULE 4.23

```
28600 REM*********************************
28601 REM EVALUATE EXPRESSION
28602 REM*********************************
28605 ERR = FALSE
28610 GOSUB 28500 : RESULT = TERM
28620 IFT=-1OR T=32 OR T=58 OR T=59 OR T
=41 OR T=46 OR PTR>LEN(IN$) THEN RETURN
28630 IF T=43 THEN GOSUB 28500 : RESULT
= INT(RESULT+TERM) : GOTO 28620
28640 IF T=45 THEN GOSUB 28500 : RESULT
= INT(RESULT-TERM) : GOTO 28620
28650 RESULT = 0 : EN = 4 : GOTO 28000
READY.
```

This module evaluates the lower precedence operations add and subtract.

Commentary

28610: The module does not directly call the main module at 28300, but rather it calls the high precedence operator module. This ensures that before any values are returned they have been tested to see whether they should first have been divided or multiplied by something else. Thus if the expression being evaluated were A*B + C, A*B would be evaluated before the result was returned to this module.

28620: If one of the delineators is encountered after the operand then there is nothing more to evaluate.

28630-28640: If a plus or minus sign is indicated by T as following the value obtained so far then the appropriate calculation is made.

28650: If the character indicated by T is neither a plus nor a minus sign (multiply or divide would have been dealt with by the previous module) then the 'invalid operator' error is flagged.

CHECKSUM TABLE

```
28600 123     28601 54      28602 123
28605 70      28610 47      28620 5
28630 226     28640 229     28650 81
```

Summary

It's finished! Or at least it's entered. Whether you have the heart to go on and develop the Mastercode program further depends on your own stamina — it is one of the largest single programs ever published in book form for a popular micro. In the rest of the book we examine a series of machine code routines which can be entered using the Assembler. If you have other books on 6502 programming then you may find in them useful subroutines which you can enter. It is not a bad idea to enter one or two small routines before you go on to extending the 64's BASIC with the rest of this book, if only to familiarise yourself with the working of the program. The usual word of caution applies, however: do make sure that the program is saved before you try to do anything with machine code. Anyone can make a mistake — and regret it if the program is lost.

Part 2

CHAPTER 5
The BASIC Extender

In order to achieve our aim of practical machine code routines to extend the BASIC language on the 64 it is first of all necessary to understand a little how BASIC actually works. How is a normal BASIC command picked up and acted upon, let alone the commands we wish to add to the language.

Consider first a straightforward BASIC command such as a line reading 1 GOTO 10. When you press RETURN to enter the line, it is scanned by the BASIC interpreter and the fact that it contains a BASIC keyword is detected. This keyword is then 'crunched', that is to say shrunk down to a single byte in the program file. All the BASIC keywords have such bytes, or 'tokens', with values in the range 128-202 (plus 255 for PI). In the case of GOTO the token is 137.

When the program containing this line is RUN the BASIC interpreter scans the line, skipping the line number and finds the token, which it recognises as such since it is above 128 and not contained within quotes. The token indicates a position in a table and at that position is the address of a machine code routine which will execute the command you see as GOTO 10. The interpreter now executes this machine code routine which first scans the line following the GOTO token for a line number. Having obtained this from the BASIC line the rest of the machine code routine for this particular command is devoted to finding the particular line number referred to and altering a number of system variables so that program execution jumps to that point. If no line number is found alongside the GOTO then a syntax error is indicated. If the line number is found but is not in the program then an undefined line error is given. Assuming that everything has gone according to plan, control of the program now returns to that part of the BASIC interpreter whose job is to seek out the next token in the program.

From all this we learn that a number of actions are necessary for the execution of a BASIC keyword:

1) The interpreter must recognise it as a keyword and be able to crunch it down to the form of a token.

2) The interpreter must be able to recognise the token once the program is run.

3) There must be a table in the memory somewhere from which the inter-

preter can draw the start address of a machine code routine to perform the command.
4) The machine code routine may have to be able to pick up further information for the command (eg the '10' for GOTO 10).
5) There must be provision to recognise and notify errors which prevent the execution of the routine.

Having RUN the program one more requirement is discovered when it is LISTed. It is no use trying to print out the token for the keyword. The interpreter must also have a table which allows it to look up the word which corresponds to the token so that it may be printed in a listing of the program.

In order to enter *new* keywords we must take account of all these requirements. Our keywords must be placed into the interpreter, tokens must be specified for them, routines to execute them must be provided and, most important of all, the interpeter must be persuaded to recognise and act upon the information given. All this will clearly involve *altering* the BASIC interpreter and that itself presents the first problem since, as you no doubt already know, the interpreter is not in a part of memory that we can choose to alter (RAM) but rather in custom-built chips whose contents are fixed at the time of their manufacture. All that is true but fortunately it is not the whole of the truth.

When you switch the 64 on its 64K of memory is taken up (roughly) by 8K of memory for what is known as the Kernal (a set of useful machine code routines common to most Commodore machines), 8K for the BASIC interpreter, IK for system variables (locations which the 64 uses to store important values and addresses for its operation), 4K of RAM which cannot be used by BASIC and 4K for running other devices such as the VIC chip, discs, tape, printers etc. However it is possible to switch certain sections of this memory over to user control (RAM) — in fact there is a complete 64K of RAM available, provided that you are willing to switch everything else in the machine off, meaning no BASIC, no communication with the outside world and so on.

The BASIC interpreter, appears to occupy the 8K of memory from 40960 onwards. In fact the 64 cheats by fooling the CPU into thinking that the entirely separate BASIC interpreter chip occupies that position. The *actual* 8K of RAM at that point is totally unused for the simple reason the 6502/6510 chip can only see 64K of memory at one time, so that if it is to see the BASIC interpreter it must cease to see 8K of the actual user memory. The importance of this for our purposes is that there is 8K of memory going unused, exactly the right amount of space to hold the BASIC interpreter if it were to be held in RAM and not in ROM, and in exactly the place where the CPU would expect to find the BASIC interpreter.

Our first step in altering BASIC, therefore, is to copy the contents of the BASIC interpreter into that area of RAM. This has its drawbacks — the

BASIC interpreter can now be altered (ie corrupted) accidentally in a way that could not happen if were to be held in ROM. But it can also be altered to good effect if we know what we are doing (and we do).

Given below is a short program which places three machine code routines into memory whose effect, when they are run, is to move the BASIC interpreter into RAM. Later, brief additions will be made to the program permitting us to add new keywords and make other necessary changes. For the moment shifting the interpreter will suffice.

BASIC Extender Listing I

```
0 REM12345678901234567890123456789012345
6789012345678901234567890123
100 REM BASIC EXTENDER ROUTINE
110 REM MOVE BASIC ROM INTO RAM AT $A000
 - $BFFF
120 DATA 165,1,41,254,133,1,96,160,255,2
00,32,32,8,190,1,1,32,6,8,138,153,1,1
130 DATA 200,208,240,165,1,9,1,133,1,96,
32,6,8,24,162,255,232,160,255,200,185
135 DATA 75,8,133,20,185,151,8,48,01,96,
133,21,185,227,8,129,20,144,235
139 DATA 0
140 AD = 2054
150 READ A : IF A<>0 THEN POKE AD,A : AD
  = AD+1 : GOTO 150
160 REM DO ACTUAL MOVE
165 POKE 2068,0 : POKE 2075,0
170 FOR X = 160 TO 191
190 POKE 2069,X : POKE 2076,X
200 SYS 2061
210 NEXT
220 POKE 2068,1 : POKE 2075,1
230 SYS 2054
300 END


READY.
```

Commentary

0-150: The purpose of the REM statements is to provide a clear area of memory. Into this area, which begins at 2054 (PEEK 2053 and you will find the token for the first REM, or 143) the line at 150 now POKEs the values contained in the data statements.The data statements represent, as you have no doubt guessed, machine code instructions.

Machine Code Routines

```
ADD.   DATA      SOURCE CODE
0                10 PRT
0                20 SYM
0                30 ORG 2054
806    A501      50 LBL000 LDA 1
808    29FE      60 AND #254
80A    8501      70 STA 1
80C    60        80 RTS
80D    A0FF      90 LDY #255
80F    C8        100 INY
810    202008    120 LBL001 JSR LBL002
813    BE0101    130 LDX 257.Y
816    200608    140 JSR LBL000
819    8A        150 TXA
81A    990101    160 STA 257.Y
81D    C8        170 INY
81E    D0F0      180 BNE LBL001
820    A501      190 LBL002 LDA 1
822    0901      200 ORA #1
824    8501      210 STA 1
826    60        220 RTS
```

Commentary on machine code routines

Given above is a listing of the machine code routines using, for the sake of clarity, the Mastercode Assembler. DO NOT ENTER THE ROUTINES USING THE ASSEMBLER OR YOU WILL CRASH IT, USE THE PROGRAM GIVEN ABOVE.

10-30: Assembler directives which send the output to the printer, print the symbol table and origin the program at 2054 in the memory.

50-80: These instructions load the accumulator with the contents of byte 1 in the memory, AND this value with 254 (thus setting bit zero to zero, then store the result back into address one. This has the effect of switching out the ROM and switching in the equivalent RAM. The routine is self contained. The same effect could be obtained by the

BASIC instruction POKE 1,PEEK(1) AND 254 but this would crash the system since there would no longer be a BASIC interpreter to work on.

90-180: This routine begins with a jump to the third section of the program which switches the ROM in and the RAM out. At this point in the program, line 130 is nonsense. Later the address from which the X register will be loaded will be POKEd into the machine code program. The ultimate effect of the routine is to load the X register with a byte of the interpreter, to switch out the ROM, to transfer the contents of X to the accumulator and to store this byte from the ROM into the same address, but now with RAM switched in. When this has been done, Y is incremented. When Y reaches 255 (ie 256 bytes have been transferred) the BNE instruction will fail and the next routine will be executed.

190-220: The mirror image of 50-80, these lines switch the ROM back in finally before returning control to BASIC.

You might like to note that this listing does not include all of the machine code you have entered — what remains will be explained later, it is not used at the moment.

Returning to the BASIC program itself:
165: Our machine code programs will require the use of two zeros, since the addresses on which our moves of 256 bytes will be based will all be ones with zero in the low byte eg 40960, which is 160 and zero when represented by two bytes. Placing the zeros into the REM statements permanently would create major problems if lines were entered or edited. On encountering the zeros when entering new lines the BASIC 'rechain lines' routine would corrupt the REM statements, seeking to break them up where the zeros occurred. For that reason the zeros are POKEd in temporarily for the purposes of the machine code routines and then replaced again with ones at line 220.

165-200: The POKEs give the machine code instructions at lines 130 and 160 in the assembly listing the start addresses of 32 blocks of 256 bytes (8K in all) which will be moved from ROM to RAM. When the address of each block has been POKEd into the machine code program the program is executed by the SYS call at line 200, which starts the machine code routine at line 120 in the assembly listing. 256 bytes are moved and then the address of the next block is POKEd in.

If you have entered the program and checked it, then SAVE it before it is too late. Now enter SYS 2054 in direct mode. If you have made a mistake you will almost certainly find that the machine has crashed. If not then nothing will appear to have happened. Now try this:

300 END (RETURN)....still nothing
POKE 41118,72 (RETURN)....still nothing
LIST (RETURN)....have a look at the last line. It should now read 300 HND — if not you have made an error and should switch off and start again with the BASIC program you have saved. If the procedure has worked, then entering END (RETURN) will result in the 'SYNTAX ERROR' message, while entering HND as a direct command will not. What you have done is to alter the table which stores the BASIC keywords. If you want to have real fun, type RUN/RESTORE to switch the ROM back in, load the Monitor and examine the memory from 41118 (A09E hex) onwards. There you will find the locations of the keywords, each keyword having the last character apparently missing. Take a note of the locations (or a printout) and then reload this program and relocate the interpreter. Provided that you do not alter the last character of any keywords or their length, you can make them read anything you wish. Try it and then list the program or even print it out. It will still run but its odd appearance will show that you have begun to demonstrate your power over BASIC.

CHAPTER 6
The BASIC/Machine Code Patch

Assembly Language Listing

```
10          PRT
20          ORG $C000
30          SYM
40          ;-------------------------------
---------
50          KEYWRD
60          ; NEW FUNCTION KEYWORDS
70          ; DEEK
80          BYT 68.69.69.75+128
90          ; YPOS
100          BYT 89.80.79.83+128
110          ; VARPTR
120          BYT 86.65.82.80.84.82+128
130          ;-----------------------------
--------
140          ; NEW ACTION KEYWORDS
150          ; DOKE
160          BYT 68.79.75.69+128
170          ; RKILL
180          BYT 82.75.73.76.76+128
190          ; DELETE
200          BYT 68.69.76.69.84.69+128
210          ; MOVE
220          BYT 77.79.86.69+128
230          ; FAST
240          BYT 70.65.83.84+128
250          ; SLOW
260          BYT 83.76.79.87+128
270          ; PLOT
280          BYT 80.76.79.84+128
290          ; UNDEAD
```

```
 300         BYT 85.78.68.69.65.68+128
 310         ; SUBEX
 320         BYT 83.85.66.69.88+128
 330         ; BLOAD
 340         BYT 66.76.79.65.68+128
 350         ; BVERIFY
 360         BYT 66.86.69.82.73.70.89+128
 370         ; BSAVE
 380         BYT 66.83.65.86.69+128
 390         ; FILL
 400         BYT 70.73.76.76+128
 410         WRD 0
 420         ; THIS IS THE END OF KEYWORD
TABLE CHR.
 430         ;-----------------------------
--------
 440         ; NEW ACTION VECTORS
 450         ACTVEC
 460         WRD $A830
 470         WRD $A830
 480         WRD $A830
 490         WRD $A830
 500         WRD $A830
 510         WRD $A830
 520         WRD $A830
 530         WRD $A830
 540         WRD $A830
 550         WRD $A830
 560         WRD $A830
 570         WRD $A830
 580         WRD $A830
 590         ;-----------------------------
--------
 600         ; NORMAL IS THE NORMAL NUMBER
 OF BASIC KEYWORDS
 610         NORMAL = 75
 620         ; NEWACT IS THE NUMBER OF NEW
 ACTION KEYWORDS
 630         NEWACT = 13
 640         ; NEWFUN IS THE NUMBER OF NEW
 FUNCTION KEYWORDS
 650         NEWFUN = 3
 660         ; USE BY POKEING A7E1 WITH 'J
MP EXECUTE'
```

```
 670         EXECUT JSR $73
 680         JSR DOEX
 690         JMP $A7AE
 700         DOEX BEQ LABEL
 710         SBC #$80
 720         BCC DOLET
 730         CMP #NORMAL+NEWFUN+1
 740         BCC RETURN
 750         CMP #NORMAL+NEWFUN+NEWACT+1
 760         BCS RETURN
 770         ; EXECUTE THE NEW ACTION KEYW
ORDS
 780         SBC #NORMAL+NEWFUN
 790         ASL A
 800         TAY
 810         LDA ACTVEC+1.Y
 820         PHA
 830         LDA ACTVEC.Y
 840         PHA
 850         JMP $73
 860         LABEL RTS
 870         DOLET JMP $A9A5
 880         RETURN JMP $A7F3
 890         ;-----------------------------
---------
 900         ; PRINT TOKEN ROUTINE TO USE
POKE 774 & 775 WITH PRTTOK ADDRESS
 910         PRTTOK JSR PUTREG
 920         CMP #NORMAL+129
 930         BCC PRTNOR
 940         ; PRINT THE NEW TOKENS
 950         LDA ASAVE
 960         SBC #NORMAL+1
 970         STA ASAVE
 980         LDA #KEYWRD/256
 990         LDX #KEYWRD-KEYWRD/256*256
 1000         JMP LBL000
 1010         ; PRINT NORMAL TOKENS
 1020         PRTNOR LDA #$A0
 1030         LDX #$9E
 1040         LBL000 STA $A732
 1050         STX $A731
 1060         STA $A73A
 1070         STX $A739
```

```
 1080          JSR GETREG
 1090          JMP $A71A
 1100          ;-----------------------------
---------
 1110          ; CRUNCH TOKENS ROUTINE EXTR
A CODE
 1120          ; USE BY ALTERING $A604 TO '
JMP CRUNCH'
 1130          CRUNCH JSR PUTREG
 1140          LDA $A5FC
 1150          CMP #$A0
 1160          BNE STAND
 1170          LDA #$C0
 1180          LDX #$00
 1190          JSR TOKSTR
 1200          JSR GETREG
 1210          LDY #0
 1220          JMP $A5B8
 1230          STAND LDA #$A0
 1240          LDX #$9E
 1250          JSR TOKSTR
 1260          JSR GETREG
 1270          LDA $200,X
 1280          JMP $A607
 1290          ;-----------------------------
------
 1300          GETREG LDA PSAVE
 1310          PHA
 1320          LDA ASAVE
 1330          LDX XSAVE
 1340          LDY YSAVE
 1350          PLP
 1360          RTS
 1370          ;-----------------------------
------
 1380          PUTREG PHP
 1390          STA ASAVE
 1400          STX XSAVE
 1410          STY YSAVE
 1420          PLA
 1430          STA PSAVE
 1440          LDA ASAVE
 1450          RTS
 1460          ;-----------------------------
------
```

```
 1470          TOKSTR STA $A5BE
 1480          CLD
 1490          STX $A5BD
 1500          STA $A601
 1510          STX $A600
 1520          DEX
 1530          CPX #$FF
 1540          BNE LBL003
 1550          SEC
 1560          SBC #1
 1570          LBL003 STA $A5FC
 1580          STX $A5FB
 1590          RTS
 1600          ;-----------------------------
------
 1610          PSAVE
 1620          ASAVE = PSAVE+1
 1630          XSAVE = ASAVE+1
 1640          YSAVE = XSAVE+1
END
```

Note: this listing is repeated in fully assembled form at the end of the commentary but is included here in unassembled form for the sake of clarity.

Now we know that we can change BASIC. Making a few keywords look odd is not, however, what we are setting out to do. We want to *extend* BASIC, and to do that we have to make far more extensive changes than can be accomplished by a few POKES here and there. So we turn now to consider the biggest piece of machine code which you will enter during the course of this book — unfortunately it has to be at this point or none of the rest would work!

You will remember that in the last chapter we said that in order to process a command, the BASIC interpreter needs to be able to recognise the keyword involved, to be able to recognise the token that it is crunched down to in the program listing and also to know where the machine code routine is that will perform the action that the keyword specifies. This is done by means of a table in memory from A09E to A19D (41118 - 41373). When the BASIC interpreter encounters a keyword on input it scans through that table until it finds one that matches, if there is no match then a syntax error is flagged. If the keyword *is* in the table then its position (in the

table, not in memory eg keyword no. 0 is END) then the correct token for that keyword is its position in the table plus 128, thus the token for END is 128. When a program is executed the value of the token (-128) is multiplied by two and this is interpreted as a position in another table, the table of vectors. This table tells the BASIC interpreter where to find the machine code routine for that particular instruction.

The problems to be solved by anyone wishing to extend BASIC with a number of new keywords are, therefore:
1) Space must be found for the new keyword in the table of keywords.
2) The table of vectors must be extended to provide indicators of where the new machine code routines are to be found.
3) Last but not least, the routines to execute the new commands must be entered.

These problems would be almost trivial if there *were* any room in the tables of keywords or of vectors. In fact, the keyword table is scanned under the control of a single register in the CPU (the Y register). The register is only one byte, and can therefore only be made to cope with an area of memory 256 bytes long. The existing keywords in 64 BASIC exactly fill those 256 bytes (including a mysterious command 'GO' which is not mentioned in the manual but allows GO and TO be separate and still assessed as GOTO). In other words the table cannot be added to. Even if it could be, there is no room in the table of vectors for the addresses of new routines. Admittedly, there is plenty of room in memory for the new routines themselves but how is the BASIC interpreter to be made to recognise the keywords that will run them?

So what is the answer? Well, like all other tinkerings with an existing program that you don't wish to change too much (in this case the BASIC interpreter) the answer is that it is necessary to cheat. When crunching a keyword down to a token, the interpreter scans through the keyword table until it finds a zero (the last byte of the table). If this point is reached then the keyword has not been found and a syntax error will be indicated. Our method of extending the keyword table is to replace the machine code instruction following the one which detects the zero with a jump to a new subroutine which starts the search again at a new address, thus providing a clear table of 256 more bytes in which to store the names of keywords. That little trick solves the problem of crunching the keyword, all that remains is to persuade the interpreter to recognise the new tokens that will be generated when a keyword is discovered in the second table.

Execution of individual BASIC commands is controlled by a section of the interpreter whose purpose is to look up the table of vectors and, on the basis of the information contained there, to call up the appropriate subroutine for the particular token. The RUN routine, which controls the overall execution of the program, calls up this 'single command' routine whenever a token is found in a program. Luckily for us, the RUN routine does not

have the address of the single command routine built into it but has to look up its address in a memory location in RAM (308-309 hex).

To ensure that our new tokens are recognised and acted upon, what we do is to alter the address indicated by 308-309 hex so that it points to a new single command execution routine of our own design. The new routine first tests whether the token encountered is that of one of our new keywords (they can be recognised because their value is CC hex (204) or greater). If it is not one of the new tokens then execution is sent to the normal single command routine. If it *is* a new token then our own routine takes over, indicating a completely new address for the table of vectors. Thus we can add a new vector table to the interpreter as well as a new keyword table.

If you can roughly follow that explanation, then you are ready to go on and take a look at the central machine code routine which is essential to the changes that must be made.

Module 10-30: These three lines represent instructions to the assembler which you should recognise from the commentary on the assembler section of the Mastercode program. PRT indicates that the listing should be sent to the printer, (from the PRT command onwards), ORG instructs the assembler to begin placing the program into memory from address C000 onwards and SYM ensures that the symbol table is printed out at the end of the listing. This machine code routine and those which follow will occupy the area of spare RAM which is located between C000 and CFFF (49152-53247). This memory area is unavailable in BASIC and so makes an ideal location for the storage of machine code.

Module 40-120: This section of the program extends the keyword table with three new function keywords, DEEK, YPOS and VARPTR — what they are will be explained later. At this point, all that needs to be understood is the difference between functions and actions in BASIC. A function is a mathematical operation which must occur after an '=' or an IF for example, as compared to an action, which can stand alone and cannot be entered into a mathematical expression. The reason it is important to note the distinction is that functions are dealt with in a different way than actions when they are encountered in a program — a special routine is called whenever an equals sign is encountered, for instance. Because of this special treatment the interpreter keeps a close track of the position of the function keywords in the keyword table, remembering where they begin and where they end. Keywords outside this range will not be treated as functions and a syntax error will be flagged if they are encountered inside an expression eg LET A = POKE 123,123. In the normal keyword table, the function keywords fall at the end, so the easiest position in which to locate new function keywords is at the

beginning of the second table, since this will mean that the only pointer to the end of the function keywords will have to be changed (in this case it will have to be increased by three).

In this module the only operative lines, that is lines which will be assembled into memory, are those with the byt instructions. These define the characters of the new keywords in ASCII, with the last character having 128 added to its value to mark the end of the keyword (ie bit 7 of the last character is set).

Module 130-420: In this module are entered the new action keywords with which we shall extend BASIC. The format of the table is exactly the same as explained in the previous module with exception that the list of keywords ends with an instruction which will ensure that the table ends with a zero. This will allow the interpreter to detect the end of the table.

Module 430-580: This is the new table of vectors. At this moment you have not entered any of the routines which will allow the new action keywords to *do* anything — entering this listing will merely ensure that the keywords can be placed into a program, crunched and relisted. If they are encountered during a program after this section of the machine code routines is entered, all that will happen is that the program will END since their vectors are all set to point to the END routine in normal BASIC. Note that there are no vectors included for the function keywords: they will be dealt with by a separate function evaluator routine which will be given later.

Module 590-650: This module sets up three labels with the number of normal basic keywords, the number of new action keywords and the number of new function keywords. These will be used to make the rest of this section of code more readable and simpler to change if you wish to add new keywords of your own.

Module 660-880: This module handles the execution of the new BASIC keywords.

Commentary

660-690: You will remember that we have already explained that in order to ensure that our tokens are not rejected by the interpreter, we change the address in 308-309 hex which the RUN routine uses to call the single command execution routine (SCER). In fact 308-309 direct program execution straight back to a jump command in the RUN routine and it is *this* jump which calls the SCER. The reference to 308-309 hex, which is of course in RAM, is a thoughtful provision by Commodore to allow machine code programmers to do exactly what we are doing, namely replace the standard SCER. These three lines replace three lines which

will be missed from the RUN routine, including a call to the routine which picks up the next character in the line, a jump to the SCER (our own) and a jump back to the beginning of the RUN routine.

700-760: These lines determine whether the end of the line has been reached, (ie a zero was picked up by the jump to $73), test for a variable (value less than $80) and finally examines the value of what must now be a token to reach this point, to see whether it falls into the range of the new action keyword table. If it does not, execution is returned to the normal interpreter.

770-850: The value of the end of the new function table (ie the number of original keywords plus the number of new functions) is subtracted from the token value, thus obtaining its place in the new table of action keywords. The Y register is used to obtain the corresponding action vector in the table of vectors. This address is pushed onto the stack. We now make a jump to the routine at $73 which picks up the next character in the line (this is because all the interpreter routines work on the convention that the next character has been placed into the accumulator, even if they do not actually require that character for their own operation). At the end of the subroutine called at $73 there is an RTS instruction which takes the action vector from the stack and assumes that this is the address to which a return should be made. Note that the actual return is made to an address one after the value held on the stack in order that the return is not made to the same instruction that called the subroutine in the first place. This means that the action vectors in the table all point to the byte *before* the routine they are intended to call.

860-880: These labelled lines make the various calls if tests have been 'failed' during the program — they are called by the four branch instructions earlier.

Module 890-1090: This uncrunches the tokens when the program is listed and will be called by the normal listing routine once a few adjustments have been made.

Commentary

910-930: Since this section is called in the middle of the execution of another routine (LIST), the registers are saved first in order that they may be restored on return. The token which has been picked up into the accumulator by LIST is compared with the maximum possible value for one of the normal tokens. If it is less than this value, the normal uncrunch routine is executed.

950-1000: The value of the token in the accumulator is now altered by subtracting the number of normal tokens. The accumulator will now indicate a position in the new table and this value is stored in ASAVE. Registers A and X are loaded with the start address of the new keyword table and this address is then placed into the normal uncrunch tokens routine by a call to LBL000.

1010-1090: If a normal keyword is to be printed, then the address of the normal keyword table is put into the 'uncrunch tokens' routine, which may previously have been working on the new table.

Module 1100-1280: We have already said that on detecting the end of the normal keyword table, the crunch token routine is made to jump to an extra routine of our own, this is it. The purpose of the module is to crunch the new tokens.

Commentary

1140-1160: $A5FC contains the high byte of the address of the current keyword table, which may be the normal one or our new table. If the value there is $A0 then the end of table that has been detected is that of the normal table then the next section is executed.

1170-1220: The address of the new keyword table is placed into the crunch tokens routine (JSR TOKSTR) and the routine is re-executed, loading the Y register with zero to ensure that it starts at the beginning of the new table. At the end of the second search, CRUNCH (line 1130) will be called again but this time the BNE STAND instruction will be executed.

1230-1280: The normal keyword table address is placed back into the crunch tokens routine, the registers are restored (JSR GETREG) and then the instruction in crunch tokens which was be overwritten in order to call up this routine, is reinstated. Execution finally returns to the crunch token routine to finish the job.

Module 1290-1360: This module restores the values of registers which have been saved by the next module. The only complication is restoring the processor status register, which cannot be loaded directly from memory. The operation is accomplished by taking the value into the accumulator, pushing it onto the stack and then pulling it off into the processor register.

Module 13700-1450: The opposite of the previous module, the contents of the registers are saved. Note again the use of the stack and accumulator to save the contents of the processor status register.

Module 1460-1590: The address of the keyword table is held at three distinct locations in the crunch tokens routine. This module places the required address (new or normal table) into those locations with the slight complication that the final location requires the byte *before* the table so 1 is subtracted from the address in lines 1520-1560 before it is stored.

Module 1600-1640: These lines will not affect memory when the program is assembled. They are there to tell the accumulator to set up the variables specified for use by the program.

Summary

At this point you may be sad to hear that if you entered the program into the assembler and assembled it, all you can do is save it for a while. Before the changes that this code effects can be successfully carried out, one or two modifications have to be made to the BASIC Extender program given previously. Given below is a listing of the code as it should look when it has been assembled. Do check your own assembled program against this, comparing byte values and addresses — machine code programs which do not work are seldom inspiring.

MACHINE CODE PATCH: Fully Assembled Listing

```
ADD.   DATA      SOURCE CODE
0                10 PRT
0                20 ORG $C000
C000             30 SYM
C000             40 ;----------------------
----------------
C000             50 KEYWRD
C000             60 ; NEW FUNCTION KEYWORD
S
C000             70 ; DEEK
C000   444545     80 BYT 68.69.69.75+128
C004             90 ; YPOS
C004   59504F    100 BYT 89.80.79.83+128
C008             110 ; VARPTR
C008   564152    120 BYT 86.65.82.80.84.82
+128
C00E             130 ;--------------------
----------------
C00E             140 ; NEW ACTION KEYWORDS
C00E             150 ; DOKE
C00E   444F4B    160 BYT 68.79.75.69+128
C012             170 ; RKILL
```

```
C012  524B49    180 BYT 82.75.73.76.76+12
8
C017            190 ; DELETE
C017  44454C    200 BYT 68.69.76.69.84.69
+128
C01D            210 ; MOVE
C01D  4D4F56    220 BYT 77.79.86.69+128
C021            230 ; FAST
C021  464153    240 BYT 70.65.83.84+128
C025            250 ; SLOW
C025  534C4F    260 BYT 83.76.79.87+128
C029            270 ; PLOT
C029  504C4F    280 BYT 80.76.79.84+128
C02D            290 ; UNDEAD
C02D  554E44    300 BYT 85.78.68.69.65.68
+128
C033            310 ; SUBEX
C033  535542    320 BYT 83.85.66.69.88+12
8
C038            330 ; BLOAD
C038  424C4F    340 BYT 66.76.79.65.68+12
8
C03D            350 ; BVERIFY
C03D  425645    360 BYT 66.86.69.82.73.70
.89+128
C044            370 ; BSAVE
C044  425341    380 BYT 66.83.65.86.69+12
8
C049            390 ; FILL
C049  46494C    400 BYT 70.73.76.76+128
C04D  0000      410 WRD 0
C04F            420 ; THIS IS THE END OF
KEYWORD TABLE CHR.
C04F            430 ;-------------------

----------------
C04F            440 ; NEW ACTION VECTORS
C04F            450 ACTVEC
C04F  30A8      460 WRD $A830
C051  30A8      470 WRD $A830
C053  30A8      480 WRD $A830
C055  30A8      490 WRD $A830
C057  30A8      500 WRD $A830
C059  30A8      510 WRD $A830
```

```
C05B  30A8      520 WRD $A830
C05D  30A8      530 WRD $A830
C05F  30A8      540 WRD $A830
C061  30A8      550 WRD $A830
C063  30A8      560 WRD $A830
C065  30A8      570 WRD $A830
C067  30A8      580 WRD $A830
C069            590 ;----------------------
----------------
C069            600 ; NORMAL IS THE NORMA
L NUMBER OF BASIC KEYWORDS
C069            610 NORMAL = 75
C069            620 ; NEWACT IS THE NUMBE
R OF NEW ACTION KEYWORDS
C069            630 NEWACT = 13
C069            640 ; NEWFUN IS THE NUMBE
R OF NEW FUNCTION KEYWORDS
C069            650 NEWFUN = 3
C069            660 ; USE BY POKEING A7E1
 WITH 'JMP EXECUTE'
C069  207300    670 EXECUT JSR $73
C06C  2072C0    680 JSR DOEX
C06F  4CAEA7    690 JMP $A7AE
C072  F01B      700 DOEX BEQ LABEL
C074  E980      710 SBC #$80
C076  9018      720 BCC DOLET
C078  C94F      730 CMP #NORMAL+NEWFUN+1
C07A  9017      740 BCC RETURN
C07C  C95C      750 CMP #NORMAL+NEWFUN+NE
WACT+1
C07E  B013      760 BCS RETURN
C080            770 ; EXECUTE THE NEW ACT
ION KEYWORDS
C080  E94E      780 SBC #NORMAL+NEWFUN
C082  0A        790 ASL A
C083  A8        800 TAY
C084  B950C0    810 LDA ACTVEC+1.Y
C087  48        820 PHA
C088  B94FC0    830 LDA ACTVEC.Y
C08B  48        840 PHA
C08C  4C7300    850 JMP $73
C08F  60        860 LABEL RTS
C090  4CA5A9    870 DOLET JMP $A9A5
C093  4CF3A7    880 RETURN JMP $A7F3
```

```
C096            890 ;--------------------
----------------
C096            900 ; PRINT TOKEN ROUTINE

 TO USE POKE 774 & 775 WITH PRTTOK ADDRE
SS
C096  20FAC0    910 PRTTOK JSR PUTREG
C099  C9CC      920 CMP #NORMAL+129
C09B  900F      930 BCC PRTNOR
C09D            940 ; PRINT THE NEW TOKEN
S
C09D  AD29C1    950 LDA ASAVE
C0A0  E94C      960 SBC #NORMAL+1
C0A2  8D29C1    970 STA ASAVE
C0A5  A9C0      980 LDA #KEYWRD/256
C0A7  A200      990 LDX #KEYWRD-KEYWRD/25
6*256
C0A9  4CB0C0    1000 JMP LBL000
C0AC            1010 ; PRINT NORMAL TOKEN
S
C0AC  A9A0      1020 PRTNOR LDA #$A0
C0AE  A29E      1030 LDX #$9E
C0B0  8D32A7    1040 LBL000 STA $A732
C0B3  8E31A7    1050 STX $A731
C0B6  8D3AA7    1060 STA $A73A
C0B9  8E39A7    1070 STX $A739
C0BC  20EBC0    1080 JSR GETREG
C0BF  4C1AA7    1090 JMP $A71A
C0C2            1100 ;--------------------
----------------
C0C2            1110 ; CRUNCH TOKENS ROUT
INE EXTRA CODE
C0C2            1120 ; USE BY ALTERING $A
604 TO 'JMP CRUNCH'
C0C2  20FAC0    1130 CRUNCH JSR PUTREG
C0C5  ADFCA5    1140 LDA $A5FC
C0C8  C9A0      1150 CMP #$A0
C0CA  D00F      1160 BNE STAND
C0CC  A9C0      1170 LDA #$C0
C0CE  A200      1180 LDX #$00
C0D0  200CC1    1190 JSR TOKSTR
C0D3  20EBC0    1200 JSR GETREG
C0D6  A000      1210 LDY #0
```

```
C0D8  4CB8A5    1220 JMP $A5B8
C0DB  A9A0      1230 STAND LDA #$A0
C0DD  A29E      1240 LDX #$9E
C0DF  200CC1    1250 JSR TOKSTR
C0E2  20EBC0    1260 JSR GETREG
C0E5  BD0002    1270 LDA $200.X
C0E8  4C07A6    1280 JMP $A607
C0EB            1290 ;-------------------
--------------
C0EB  AD28C1    1300 GETREG LDA PSAVE
C0EE  48        1310 PHA
C0EF  AD29C1    1320 LDA ASAVE
C0F2  AE2AC1    1330 LDX XSAVE
C0F5  AC2BC1    1340 LDY YSAVE
C0F8  28        1350 PLP
C0F9  60        1360 RTS
C0FA            1370 ;-------------------

--------------
C0FA  08        1380 PUTREG PHP
C0FB  8D29C1    1390 STA ASAVE
C0FE  8E2AC1    1400 STX XSAVE
C101  8C2BC1    1410 STY YSAVE
C104  68        1420 PLA
C105  8D28C1    1430 STA PSAVE
C108  AD29C1    1440 LDA ASAVE
C10B  60        1450 RTS
C10C            1460 ;-------------------
--------------
C10C  8DBEA5    1470 TOKSTR STA $A5BE
C10F  D8        1480 CLD
C110  8EBDA5    1490 STX $A5BD
C113  8D01A6    1500 STA $A601
C116  8E00A6    1510 STX $A600
C119  CA        1520 DEX
C11A  E0FF      1530 CPX #$FF
C11C  D003      1540 BNE LBL003
C11E  38        1550 SEC
C11F  E901      1560 SBC #1
C121  8DFCA5    1570 LBL003 STA $A5FC
C124  8EFBA5    1580 STX $A5FB
C127  60        1590 RTS
C128            1600 ;-------------------
--------------
```

```
C128            1610 PSAVE
C128            1620 ASAVE = PSAVE+1
C128            1630 XSAVE = ASAVE+1
C128            1640 YSAVE = XSAVE+1

 TOTAL ERRORS IN FILE --- 0


KEYWRD           C000
ACTVEC           C04F
NORMAL           4B
NEWACT           D
NEWFUN           3
EXECUT           C069
DOEX             C072
LABEL            C08F
DOLET            C090
RETURN           C093
PRTTOK           C096
PRTNOR           C0AC
LBL000           C0B0
CRUNCH           C0C2
STAND            C0DB
GETREG           C0EB
PUTREG           C0FA
TOKSTR           C10C
LBL003           C121
PSAVE            C128
ASAVE            C129
XSAVE            C12A




YSAVE            C12B
 TOTAL NUMBER OF SYMBOLS --- 23
```

CHAPTER 7
BASIC Extender Program II

We have already entered a short program which moves the interpreter from ROM to RAM, thus allowing changes to be made. Now that some of the necessary machine code additions have been discussed we are in a position to update that BASIC program in such a way that the extensions to BASIC can begin. The full listing of this expanded program is given, even though most of the lines are shared with the version you have already entered — only the changes will be commented upon.

BASIC Extender II Listing

```
0 REM12345678901234567890123456789012345
67890123456789012345678901234
1 REM12345678901234567890123456789012345
67890123456789012345678901234567890
2 REM12345678901234567890123456789012345
67890123456789012345678901234567890
3 REM12345678901234567890123456789012345
67890123456789012345678901234567890
20 DEV = 1
100 REM BASIC EXTENDER ROUTINE
110 REM MOVE BASIC ROM INTO RAM AT $A000
 - $BFFF
120 DATA 165,1,41,254,133,1,96,160,255,2
00,32,32,8,190,1,1,32,6,8,138,153,1,1
130 DATA 200,208,240,165,1,9,1,133,1,96,
32,6,8,24,162,255,232,160,255,200,185
135 DATA 75,8,133,20,185,151,8,48,01,96,
133,21,185,227,8,129,20,144,235
139 DATA 0
140 AD = 2054
150 READ A : IF A<>0 THEN POKE AD,A : AD
  = AD+1 : GOTO 150
155 REM LOAD MACHINE CODE FROM TAPE/DISC
156 INPUT " FILE NAME "; IN$ : IF DEV=8
```

```
THEN IN$ = IN$+",S,R"
157 OPEN 2,DEV,0,IN$ : INPUT# 2,SA,EA :
FOR X = SA TO EA : INPUT# 2,T : POKE X,T
158 NEXT X : CLOSE 2
160 REM DO ACTUAL MOVE
165 POKE 2068,0 : POKE 2075,0
170 FOR X = 160 TO 191
190 POKE 2069,X : POKE 2076,X
200 SYS 2061
210 NEXT
220 POKE 2068,1 : POKE 2075,1
221 REM DATA FOR ROM EXCUTE ALTERATION
223 X = 0 : DATA 225,167,76,226,167,105,
227,167,192
227 DATA 0
228 READ T1 : IF T1=0 THEN 230
229 READ T2,T3 : POKE 2123+X,T1:POKE 219
9+X,T2:POKE 2275+X,T3:X=X+1:GOTO 228
230 SYS 2087 : REM ALTER ROM
231 REM ALTER CRUNCH TOKENS ROUTINE
232 POKE 42500,76 : POKE 42501,194 : POK
E 42502,192
240 REM ALTER PRINT TOKEN ROUTINE
241 POKE 774,150 : POKE 775,192
300 END

READY.
```

Commentary

1-3: The data statements in the original program provide for a third machine code routine which was not explained. These REM statements provide room for the data on which this third routine will work.

20: With this expanded program we shall be loading a file of machine code. The device specified is the cassette recorder, so if you are working with a disc you will need to change this to 8.

155-158: These lines load a machine code file from the specified device — the machine code extender we have just discussed. The lines are equivalent to the machine code loader in the Monitor.

223-227: Data for the third machine code routine which begins at item ten of the data line 130. The third routine will be used to POKE into the interpreter's execution routine a new address which will cause a jump to the modified execution routine in the machine code extender. This must be done in machine code because two bytes have to be altered. Altering one byte of the jump with a BASIC POKE would result in a nonsense jump when the interpreter tried to execute the next POKE. The data in line 223 specifies three addresses with values for low byte, high byte and the number to be POKEd there. The assembler listing for the third routine is as follows:

EXECUTION ROUTINE MODIFIER: Assembly Language Listing

```
827   200608   JSR $0806
82A   18       CLC
82B   A2FF     LDX #$FF
82D   E8       INX
82E   A0FF     LDY #$FF
830   C8       INY
831   B94B08   LDA $084B,Y
834   8514     STA $14
836   B99708   LDA $0897,Y
839   3001     BMI $83C
83B   60       RTS
83C   8515     STA $15
83E   B9E308   LDA $08E3,Y
841   8114     STA ($14,X)
843   90EB     BCC $830

CONTINUE ( Y/N ) :
```

Commentary on Machine Code

This routine takes three two-byte addresses from the data in the REM statements and stores new information (again taken from the REM statements) in them. Instructions at addresses 827-82E switch in RAM and initialise the carry flag and the X and Y registers. 830-83C load the low and high bytes of the address from the REM statement at line 1 of the BASIC program, with a test made of the high byte to see that it is in the right range to be an address in the interpreter. 83E-843 pick up and place the new data, branching back for the next address.

Returning to the BASIC program:

228-229: This new data is now POKEd into lines 1,2 and 3 to be picked up by the third machine code routine in line 0.

230: This now becomes SYS 2087 rather than SYS 2054. The machine code routine at 2054 is still called but this is done from within the third machine code routine on line 0. The necessary changes to the execution routine will be completed by this call.

231-232: The new address is POKEd into the 'crunch tokens' routine — this can be done from BASIC since the crunch tokens routine is not used when a program is being executed (it could not be done in direct mode since the tokens have to be crunched during execution).

240-241: The 'print token' vector is altered to point to our modified routine.

Having modified your first BASIC Extender program, or entered this version afresh, the best procedure if you are using tape for storage will be to save this program at the beginning of the tape, leave a gap for further extensions to the program and then to save the assembled version of the BASIC/Machine Code Patch on the same tape. When the program is RUN you will be asked to specify the file name and you should respond with whatever name you have given to the BASIC/Machine Code Patch on saving. Simply press play and it will load, thus saving a great deal of cassette swapping.

Having assembled the BASIC/Machine Code Patch and loaded it into memory using this program you should now be able to enter any of the new action keywords (not functions yet) specified in the keyword table. As mentioned, none of them will do anything but execute END at this stage, but that will soon change.

CHAPTER 8
Simple BASIC Action Keywords

SECTION 1: Keyword UNDEAD

Now that the machine code and BASIC routines have been given for extending BASIC it is time to enter the routines which allow the new keywords to *do* something. In this section of the book we shall discuss three action keywords which are simpler than others in that they require no parameters to be entered before they can be executed. In normal BASIC, for instance, the command GOTO 10 requires not only a routine to execute the GOTO but also one to pick up the 'parameter' 10, without which the GOTO would be meaningless; STOP, on the other hand, requires no parameter, only the keyword itself. Later on we shall show how keywords which require parameters can be added.

The keyword we shall be adding here is called UNDEAD —other people tend to call it 'OLD' but there had been a vampire film on television the night before the routine was written. The effect of the command is to overcome that annoying problem which crops up for everyone from time to time — NEW is entered to clear a program from memory and it is suddenly remembered that the program was not saved. UNDEAD will restore the program as good as new, except that any variables will be lost.

UNDEAD: Assembly Language Listing

```
10        PRT
20        SYM
30        ORG $C05D
40        WRD UNDEAD-1
50        ORG $C1E3
60        UNDEAD LDA #$FF
70        LDY #1
80        STA ($2B),Y
90        JSR $A533
100        LDA $22
110        CLC
120        CLD
130        ADC #2
```

```
 140          STA $2D
 150          LDA $23
 160          ADC #0
 170          STA $2E
 180          JMP $A65E
 190          END
END
```

30-40: You will remember from entering the machine code extender that all the action vectors for the new keywords currently point to END. These two lines place the address of the label UNDEAD into the appropriate action vector in the new table.

50: The machine code routines for the new keywords will occupy, together with the machine code extender, an area of memory between C000-C4B5 hex. They will not be entered in order but this will have no effect on their execution.

60-90: When NEW is executed, one of the changes it makes is to place three bytes containing zero at the beginning of the program area (800-802 hex) though one of these (800 hex) is always zero anyway. Three zeros is the indicator used by the interpreter to show the end of the program so that after NEW, any attempt to execute or LIST the program ends before it begins, despite the fact that the program is intact in the memory. These lines place the value FF hex into the third byte of the program area, this being the high byte of the two link bytes of the first line of the BASIC program — the value in the two link bytes is now nonsense but that is unimportant. The interpreter's 'rechain lines' routine is now called at A533 hex, and this scans through the program restoring all the link bytes which begin each BASIC line and point to the beginning of the next.

100-170: The rechain lines routine uses the two bytes of memory at 22-23 hex as a variable indicating progress through the program and, when the process is finished these bytes point to the end of the last line of the BASIC program. This value is picked up and placed into the main pointer which records the end of the BASIC program, with the 2 being added to take account of the two extra zeros which mark the genuine end of the program.

180: There are various other pointers which must be restored before the program can be successfully retrieved but these can be dealt with by a call to the CLR routine in the normal interpreter.

190: Note the use of the END directive here. It is useful in that not only does it indicate the end of the assembler listing to the Mastercode Assembler, its address is also the first free byte after the end of the routine that it

concludes, so that any routine to be added after this one could be started at the address specified for END in the assembled listing.

UNDEAD: Fully Assembled Listing

```
ADD.   DATA      SOURCE CODE
Ø                10 PRT
Ø                20 SYM
Ø                30 ORG $C05D
C05D   E2C1      40 WRD UNDEAD-1
C05F             50 ORG $C1E3
C1E3   A9FF      60 UNDEAD LDA #$FF
C1E5   A001      70 LDY #1
C1E7   912B      80 STA ($2B).Y
C1E9   2033A5    90 JSR $A533
C1EC   A522      100 LDA $22
C1EE   18        110 CLC
C1EF   D8        120 CLD
C1FØ   6902      130 ADC #2
C1F2   852D      140 STA $2D
C1F4   A523      150 LDA $23
C1F6   6900      160 ADC #Ø
C1F8   852E      170 STA $2E
C1FA   4C5EA6    180 JMP $A65E
C1FD             190 END

 TOTAL ERRORS IN FILE --- Ø


UNDEAD            C1E3
 TOTAL NUMBER OF SYMBOLS --- 1
```

The procedure to follow in entering this routine is as follows:
1) Load the Mastercode program and *first* recall the extender machine code file that you have previously entered. The reason that this is necessary is that in assembling UNDEAD one of the action vectors in the new table is overlaid — this will not achieve much if the program is not there.
2) Enter the assembler listing for UNDEAD using the File Editor.
3) Call up the Assembler and assemble UNDEAD.
4) Save the whole of the memory area from C000 to C1FC as a machine code file or, better still, save C000 to C4B5, which is the whole of the area that will eventually be used by the new routines. Using this second method every time a new routine is entered will ensure that you do not inadvertently chop off any routines which fall later in the area but were entered before the current one.

5) Load BASIC Extender II and run it, giving it the name of the file containing the extender and UNDEAD.
6) If everything has worked as it should you can now NEW the BASIC Extender and then try to LIST it — there will be nothing there, as you would expect.
7) Enter, in direct mode UNDEAD and press RETURN. LIST again and you should see the program fully restored to life.

UNDEAD: Notes On Use

UNDEAD can only restore a program which has not been overlaid in the memory with something else. If, after entering NEW you enter a line of BASIC or declare a variable, there is no way that UNDEAD can help since the program is no longer intact in the memory.

SECTION 2: Keyword Subex

Enter the following program on your 64:

```
10 GOSUB 20
20 GOTO 10
```

Now run it, and you will find that in an instant you have run out memory — how? Well, each time you call a subroutine the address to which execution should be sent when RETURN is encountered in the program is stored in an area of memory called the 'return stack'. Each RETURN takes the last address off the stack and jumps to it, each GOSUB adds another address on top of what is already there. Leaving a subroutine by any other means than a RETURN leaves the return address on the stack. Do it often and the stack runs out of space and an 'out of memory' error is generated.

Admittedly, jumping out of subroutines without a RETURN is not a practice to be encouraged too often, after all, why use a GOSUB at all if you don't want to return? But there are circumstances where it can be extremely useful to leave a subroutine without cluttering up the memory. If you look at the assembler, for instance, you will find that there are many cases of subroutines which call subroutines which call subroutines.... There is nothing wrong with that, indeed it's good programming practice to put as much into subroutines as possible. But what happens if, four or five subroutines removed from the control routine you encounter a condition which means that this chain of subroutines is not actually to be executed — an error in the data input for example. What you really want to do is to flag the error and return immediately to the control routine so that it can take the appropriate action but you can't in standard BASIC. You have to return through the subroutines, each time setting up a line to detect the error flag and RETURN again until the control routine is reached. In a complex program this can mean many extra lines and a considerable cost in terms of time.

The answer to the problem is the command SUBEX. All that it does is to remove the last return address from the stack so that you can leave the last subroutine without cluttering up the stack. Of course, if you want to jump straight from the fifth in a chain of subroutines back to the beginning, then SUBEX will have to executed five times first but the saving in time and program complexity can often be considerable. With a SUBEX a program such as:
10 GOSUB 20
20 SUBEX
30 GOTO 10
will run indefinitely.

SUBEX: Assembly Language Listing

```
 10        PRT
 20        SYM
 30        ORG $C05F
 40        WRD SUBEX-1
 50        ORG $C1FD
 60        SUBEX LDA #$FF
 70        STA $4A
 80        JSR $A38A
 90        TXS
 100         CMP #$8D
 110         BNE RETERR
 120         PLA
 130         PLA
 140         PLA
 150         PLA
 160         PLA
 170         RTS
 180         RETERR JMP $A8E0
 190         END
END
```

60-80: When a RETURN is executed in normal BASIC the routine at A38A hex is called to find on the stack the first return address, this being necessary since there may also be data for FOR loops on the stack. The routine at that address also manipulates the stack so that the first return address is placed at the top of the stack. These lines set the memory address at 4A hex, which initialises the search routine, then the routine is called.

90-110: On return from the stack search routine the last value to be placed on the stack will,hopefully, be a pointer to the position of the return address on the stack and this is stored in the X register. This is confirmed by placing the value 8D into the accumulator. If a return address has not been found then the accumulator will not contain 8D and a jump will be made to RETERR, which specifies a jump to the 'RETURN WITHOUT GOSUB ERROR' message.

120-170: The five bytes of the return address are pulled off the stack and discarded. The SUBEX routine is complete and it returns.

SUBEX: Fully Assembled Listing

```
ADD.   DATA      SOURCE CODE
Ø                10 PRT
Ø                20 SYM
Ø                30 ORG $C05F
C05F   FCC1      40 WRD SUBEX-1
C061             50 ORG $C1FD
C1FD   A9FF      60 SUBEX LDA #$FF
C1FF   854A      70 STA $4A
C201   208AA3    80 JSR $A38A
C204   9A        90 TXS
C205   C98D      100 CMP #$8D
C207   D006      110 BNE RETERR
C209   68        120 PLA
C20A   68        130 PLA
C20B   68        140 PLA
C20C   68        150 PLA
C20D   68        160 PLA
C20E   60        170 RTS
C20F   4CE0A8    180 RETERR JMP $A8E0
C212             190 END

 TOTAL ERRORS IN FILE --- Ø


SUBEX             C1FD
RETERR            C20F
 TOTAL NUMBER OF SYMBOLS --- 2
```

As with UNDEAD, the procedure to load SUBEX into memory is to call up the last machine code file you saved (Extender and UNDEAD), load it into memory and then assemble SUBEX. Save the whole memory area and then load BASIC Extender II, giving it the name of your new file. You

should now be able to run the second BASIC program given above in the introduction to SUBEX without running out of memory.

SUBEX: Notes On Use

SUBEX is a powerful command but some care is needed in its use. If you are in the middle of a chain of subroutines then it is essential that you count exactly how many RETURNS you wish to default through or you will end up either returning to the wrong subroutine or being given a RETURN WITHOUT GOSUB error.

SECTION 3: Keyword RKILL

When you are developing a program it is good practice to include as many REM statements as will be necessary to ensure that you understand the program the next time you come to work on it; readability is also improved by spacing out commands well on the lines. When the program is finished, however, the extra REMs and spaces are simply so much wasted memory which could be put to better use. RKILL solves this problem by removing all REM statements which fall at the end of lines and all spaces outside quotes. Note that RKILL does not remove REM statements which stand alone on a line since they may be, if you are writing your programs properly, the headings to sections of the program. GOTOs and GOSUBs will point at them, thus allowing new first lines to be added to sections without having to go through the program and change several GOTOs and GOSUBs.

RKILL: Assembly Language Listing

```
 10        PRT
 20        SYM
 30        QFLAG = $F
 40        REMTOK = $8F
 50        ASAVE = $C105
 60        XSAVE = $C106
 70        ORG $C051
 80        WRD RKILL-1
 90        ORG $C160
 100        RKILL LDA #$FF
 110        STA $14
 120        STA $15
 130        ; SAVE PRESENT BASIC WARM STA
RT LINK
 140        LDA $302
```

```
 150         LDX $303
 160         STA TEMP
 170         STX TEMP+1
 180         ; PUT NEW WARM START LINK IN
 190         LDA #LBL003-LBL003/256*256
 200         LDX #LBL003/256
 210         STA $302
 220         STX $303
 230         ; GET NEXT LINE NO TO BE TREA
TED
 240         LBL003 INC $14
 250         BNE LBL004
 260         INC $15
 270         ; USE ROM ROUTINE TO GET ADD
OF LINE IN $5F & $60
 280         LBL004 JSR $A613
 290         ; IF HI LINK BYTE = 0 THEN EX
IT
 300         LDY #1
 310         LDA ($5F),Y
 320         BEQ LBL009
 330         ; GET THIS LINE NO. INTO $14
& $15
 340         INY
 350         LDA ($5F),Y
 360         STA $14
 370         INY
 380         LDA ($5F),Y
 390         STA $15
 400         ; COPY LINE TO INPUT BUFFER D
ELETING SPACES-EXCEPT IN QUOTES
 410         LDX #4
 420         STX QFLAG
 430         LBL005 INY
 440         LDA ($5F),Y
 450         ;IF BYTE = 0 THIS IS THE END
OF THE LINE SO INPUT IT
 460         BEQ LBL007
 470         ;IF ITS A QUOTE THEN TOGGLE T
HE QUOTES FLAG
 480         CMP #34
 490         BNE LBL006
 500         LDA QFLAG
 510         EOR #$FF
```

```
 520         STA QFLAG
 530         LDA #34
 540         ;IF THE QUOTFLAG IS SET DONT
DELETE ANYTHING
 550         LBL006 BIT QFLAG
 560         BMI LBL008
 570         ; TEST FOR SPACE & DELETE IT
IF FOUND
 580         CMP #$20
 590         BEQ LBL005
 600         ;TRANSFER FIRST NON-SPACE EVE
N IF ITS A REM
 610         CMP #REMTOK
 620         BNE LBL008
 630         CPX #4
 640         BNE LBL002
 650         INX
 660         STA $1FB.X
 670         INX
 680         LBL002 DEX
 690         LBL007 LDA #0
 700         INX
 710         STA $1FB.X
 720         STX $B
 730         JMP $A4A4
 740         LBL008 INX
 750         STA $1FB.X
 760         JMP LBL005
 770         LBL009 LDA TEMP
 780         LDX TEMP+1
 790         STA $302
 800         STX $303
 810         JMP $A474
 820         TEMP WRD 0
 830         END
END
```

100-120: 14-15 hex are locations used by the BASIC file editor when working through a program for any purpose. They are loaded with 255 each so that when the main routine begins, the addition of 1 will set them to zero.

130-170: During this instruction we shall be calling up the line input routine, which processes a line placed into the input buffer from the keyboard. When this process is finished the routine normally returns to a state

of waiting for a keyboard input. What we want it to do is to return to the RKILL routine so we save the existing link address at 302-303 hex so that we can place a link to this routine in it.

180-220: A new link address pointing to this routine is installed.

230-260: The line number being worked on is incremented by one.

270-280: A ROM routine is used to find the address of that line in memory. If there is no such line, the routine will return the address of the line with the closest number after it.

290-320: If the high byte of the link byte at the address indicated by the ROM routine is zero then what has been found is the end of file and the routine ends.

330-390: The actual number of the line found is picked up and stored into 14-15 hex.

410-420: These lines initialise the routine. X will be the start of the text of the line in the input buffer (after the link bytes and line number), the quotes flag, which records whether a character being input is inside quotes is initialised.

430-440: The next byte of the line is placed into the accumulator.

450-460: If the byte is zero then the end of line has been reached and a jump is made to the ROM routine which actually inputs a line.

470-530: If the character picked up is a quotation mark then the quote flag is either set or reset according to whether this is the first or second of a pair.

550-560: If the quote flag is now set the main part of the routine is omitted since we do not wish to tamper with the contents of lines within quotes.

570-590: If the character picked up from the line is a space then it is ignored and another character is picked up.

610-620: A test is made for a REM token, if the character is not REM then it is placed into the input buffer.

630-680: If a REM token has been picked up then a test is made to see if it falls at the beginning of a line (X reg. is 4). If so then it is placed into the file by 650-660. If it is not the first character then the X register is decremented to point to the colon which will precede it in the input buffer. The pointer contained in the X register. will now point to the colon before the REM if the REM did not fall at the beginning of the line and to the position after the REM if it did (all these are in the input buffer, remember, not in the line itself).

690-730: A zero is now stored in the input buffer thus marking the end of the line. The length of the line is placed into B hex and the input routine is then executed, thus placing the stripped down line into the program in place of the original.

740-760: At this point all the previous tests have failed so the character is not one to be deleted and we need to store the character in the input buffer, then repeat the loop to pick up the next character.

770-810: This is the exit routine from RKILL so the link to the normal keyboard input is restored and a jump made to the direct mode routine, bringing up the 'READY' message.

820: TEMP sets up the storage location for the original warmstart link which is saved at the beginning of RKILL.

RKILL: Fully Assembled Listing

```
ADD.   DATA      SOURCE CODE
Ø                10 PRT
Ø                20 SYM
Ø                30 QFLAG = $F
Ø                40 REMTOK = $8F
Ø                50 ASAVE = $C105
Ø                60 XSAVE = $C106
Ø                70 ORG $C051
C051   5FC1      80 WRD RKILL-1
C053             90 ORG $C160
C160   A9FF      100 RKILL LDA #$FF
C162   8514      110 STA $14
C164   8515      120 STA $15
C166             130 ; SAVE PRESENT BASIC
WARM START LINK
C166   AD0203    140 LDA $302
C169   AE0303    150 LDX $303
C16C   8DE1C1    160 STA TEMP
C16F   8EE2C1    170 STX TEMP+1
C172             180 ; PUT NEW WARM START
LINK IN
C172   A97C      190 LDA #LBL003-LBL003/25
6*256
C174   A2C1      200 LDX #LBL003/256
C176   8D0203    210 STA $302
C179   8E0303    220 STX $303
C17C             230 ; GET NEXT LINE NO TO
 BE TREATED
```

```
C17C  E614      240 LBL003 INC $14
C17E  D002      250 BNE LBL004
C180  E615      260 INC $15
C182            270 ; USE ROM ROUTINE TO
GET ADD OF LINE IN $5F & $60
C182  2013A6    280 LBL004 JSR $A613
C185            290 ; IF HI LINK BYTE = 0
 THEN EXIT
C185  A001      300 LDY #1
C187  B15F      310 LDA ($5F).Y
C189  F047      320 BEQ LBL009
C18B            330 ; GET THIS LINE NO. I
NTO $14 & $15
C18B  C8        340 INY
C18C  B15F      350 LDA ($5F).Y
C18E  8514      360 STA $14
C190  C8        370 INY
C191  B15F      380 LDA ($5F).Y
C193  8515      390 STA $15
C195            400 ; COPY LINE TO INPUT
BUFFER DELETING SPACES-EXCEPT IN QUOTES
C195  A204      410 LDX #4
C197  860F      420 STX QFLAG
C199  C8        430 LBL005 INY
C19A  B15F      440 LDA ($5F).Y
C19C            450 ;IF BYTE = 0 THIS IS
THE END OF THE LINE SO INPUT IT
C19C  F022      460 BEQ LBL007
C19E            470 ;IF ITS A QUOTE THEN
TOGGLE THE QUOTES FLAG
C19E  C922      480 CMP #34
C1A0  D008      490 BNE LBL006
C1A2  A50F      500 LDA QFLAG
C1A4  49FF      510 EOR #$FF
C1A6  850F      520 STA QFLAG
C1A8  A922      530 LDA #34
C1AA            540 ;IF THE QUOTFLAG IS S
ET DONT DELETE ANYTHING
C1AA  240F      550 LBL006 BIT QFLAG
C1AC  301D      560 BMI LBL008
C1AE            570 ; TEST FOR SPACE & DE
LETE IT IF FOUND
C1AE  C920      580 CMP #$20
C1B0  F0E7      590 BEQ LBL005
```

```
C1B2            600 ;TRANSFER FIRST NON-S
PACE EVEN IF ITS A REM
C1B2  C98F      610 CMP #REMTOK
C1B4  D015      620 BNE LBL008
C1B6  E004      630 CPX #4
C1B8  D005      640 BNE LBL002
C1BA  E8        650 INX
C1BB  9DFB01    660 STA $1FB.X
C1BE  E8        670 INX
C1BF  CA        680 LBL002 DEX
C1C0  A900      690 LBL007 LDA #0
C1C2  E8        700 INX
C1C3  9DFB01    710 STA $1FB.X
C1C6  860B      720 STX $B
C1C8  4CA4A4    730 JMP $A4A4
C1CB  E8        740 LBL008 INX
C1CC  9DFB01    750 STA $1FB.X
C1CF  4C99C1    760 JMP LBL005
C1D2  ADE1C1    770 LBL009 LDA TEMP
C1D5  AEE2C1    780 LDX TEMP+1
C1D8  8D0203    790 STA $302
C1DB  8E0303    800 STX $303
C1DE  4C74A4    810 JMP $A474
C1E1  0000      820 TEMP WRD 0
C1E3            830 END

 TOTAL ERRORS IN FILE --- 0


QFLAG            F
REMTOK           8F
ASAVE            C105
XSAVE            C106
RKILL            C160
LBL003           C17C
LBL004           C182
LBL005           C199
LBL006           C1AA
LBL002           C1BF
LBL007           C1C0
LBL008           C1CB
LBL009           C1D2
TEMP             C1E1
 TOTAL NUMBER OF SYMBOLS --- 14
```

As with the previous routines, to enter RKILL it must be assembled into your overall file which, by now, should consist of the extender, UNDEAD and SUBEX. The file should be saved and then placed into memory by BASIC Extender II. When the file has been loaded, entering RKILL in direct mode should strip the BASIC Extender of spaces and REMs.

RKILL: Notes On Use

RKILL is a direct mode command — it cannot be usefully placed in a program line. The reason for this is that, as it shortens the program, it alters the pointers used by the interpeter. Doing this as a program is running is a sure recipe for disaster so RKILL terminates program execution once it has been completed (like LIST).

CHAPTER 9
The Problem Of Parameters

SECTION 1: GETWRD

We have already noted that some action keywords require further information, or 'parameters', to be picked up from the program line before they can be executed. This insignificant — looking routine performs that function for our new keywords and must be entered into the overall machine code file before we can add new keywords with parameters. Its name is GETWRD.

GETWRD: Assembly Language Listing

```
S-GET*
 10         PRT
 20         SYM
 30         ORG $C12C
 40         ; ROUTINE TO GET 16 BIT UNSIGN
ED INTEGER FROM BASIC INTO $14 & $15
 50         GETWRD JSR $AD8A
 60         JMP $B7F7
 70         END
END
```

50: The routine called here is in the standard interpreter and simply picks up a floating point number from the BASIC line which is currently being processed. Invalidly entered numbers will generate a syntax error as for a normal BASIC keyword.

60: This routine converts the number picked up into an integer in the range 0-65535. Numbers outside this range return an ILLEGAL QUANTITY error.

GETWRD: Fully Assembled Listing

```
ADD.   DATA      SOURCE CODE
0                10 PRT
```

```
0                    20 SYM
0                    30 ORG $C12C
C12C                 40 ; ROUTINE TO GET 16 BI
T UNSIGNED INTEGER FROM BASIC INTO $14 &
 $15
C12C  208AAD         50 GETWRD JSR $AD8A
C12F  4CF7B7         60 JMP $B7F7
C132                 70 END

 TOTAL ERRORS IN FILE --- 0


GETWRD               C12C
 TOTAL NUMBER OF SYMBOLS --- 1
```

Yes that *is* all there is to it. You cannot yet do anything with the routine but the commands to come will use it to the full to return parameters for our new keywords. Like the previous routines it must be entered into the overall machine code file before going on to the next section.

SECTION 2: Keyword DOKE

Now that we can pick up parameters for a keyword all kinds of new possibilities are opened up. The first of them is DOKE, which simply POKES a number in the range 0-65535 into a two byte location in memory. This saves all the fuss of entering expressions like VAR-256*INT(VAR/256) every time 2 byte numbers have to be put into memory.

DOKE: Assembly Language Listing

```
 10         PRT
 20         GETWRD = $C12C
 30         SYM
 40         ORG $C04F
 50         WRD DOKE-1
 60         ORG $C212
 70         DOKE JSR GETWRD
 80         ; CHECK FOR A COMMA
 90         JSR $AEFD
 100         ; PUT ADDRESS ON STACK WHILE
GETTING THE DATA
 110         LDA $14
 120         PHA
```

```
 130         LDA #$15
 140         PHA
 150         ; GET VALUE TO BE DOKED
 160         JSR GETWRD
 170         ; PUT ADDRESS INTO TEMPORARY
POINTER
 180         LDX #$15
 190         LDY #$14
 200         PLA
 210         STA #$15
 220         PLA
 230         STA #$14
 240         TYA
 250         ; USE ROM ROUTINE TO SET Y TO
 ZERO & PUT FIRST BYTE IN MEMORY
 260         JSR $B828
 270         INY
 280         TXA
 290         STA (#$14),Y
 300         RTS
 310         END
END
```

Commentary

70: This obtains the number representing the address to which the number is to be DOKEd, by use of GETWRD.

80-90: If there is no comma after the first parameter then a syntax error is flagged.

100-140: The first parameter is saved on the stack after being picked up from 14-15 hex where it is stored by GETWRD.

160: GETWRD is called again to find the value to be DOKEd.

180-230: The value is put into X and Y and the address is placed back into 14-15 hex.

240-260: The low byte of the data to be DOKEd is placed into the accumulator then a tiny section of ROM is called which sets the Y register to zero, and loads the contents of the accumulator into the location indicated by the contents of the two bytes at 14 hex. It could have been done with four bytes in this routine itself but since the two instructions were there in the ROM with a return instruction after them, why not use that.

270-300: The value in Y is increased to 1 and the high byte of the value to be DOKEd is stored in the byte above that where the low byte was placed.

DOKE: Fully Assembled Listing

```
ADD.   DATA      SOURCE CODE
Ø                10 PRT
Ø                20 GETWRD = $C12C
Ø                30 SYM
Ø                40 ORG $C04F
C04F   11C2      50 WRD DOKE-1
C051             60 ORG $C212
C212   202CC1    70 DOKE JSR GETWRD
C215             80 ; CHECK FOR A COMMA
C215   20FDAE    90 JSR $AEFD
C218             100 ; PUT ADDRESS ON STAC
K WHILE GETTING THE DATA
C218   A514      110 LDA $14
C21A   48        120 PHA
C21B   A515      130 LDA $15
C21D   48        140 PHA
C21E             150 ; GET VALUE TO BE DOK
ED
C21E   202CC1    160 JSR GETWRD
C221             170 ; PUT ADDRESS INTO TE
MPORARY POINTER
C221   A615      180 LDX $15
C223   A414      190 LDY $14
C225   68        200 PLA
C226   8515      210 STA $15
C228   68        220 PLA
C229   8514      230 STA $14
C22B   98        240 TYA
C22C             250 ; USE ROM ROUTINE TO
SET Y TO ZERO & PUT FIRST BYTE IN MEMORY
C22C   2028B8    260 JSR $B828
C22F   C8        270 INY
C230   8A        280 TXA
C231   9114      290 STA ($14),Y
C233   60        300 RTS
C234             310 END

 TOTAL ERRORS IN FILE --- Ø

GETWRD            C12C
DOKE              C212
 TOTAL NUMBER OF SYMBOLS --- 2
```

DOKE: Notes On Use

DOKE can be used to replace any double POKE command but do remember that DOKEing a value below 256 will set the byte above the one specified in the first parameter to zero — DOKE < 255 cannot be used as a substitute for POKE.

The correct syntax for DOKE is:
DOKE < address> ,< value>

SECTION 3: Keyword Plot

There is no doubt that the Commodore cursor control characters give a great deal of flexibility when it comes to outputting material to the screen. Nevertheless there *are* times when it would be nice to be able to print something to the middle of the screen without specifying a long list of cursor controls or printing a section of some previously defined string of such characters. The new command PLOT will allow you to move the cursor position to any position on the screen with a single command.

PLOT: Assembly Language Listing

```
10        PRT
20        SYM
30        GETWRD = $C12C
40        ORG $C05B
50        WRD PLOT-1
60        ORG $C3DA
70        PLOT JSR GETWRD
80        JSR $AEFD
90        LDA $15
100        BNE IQERR
110        LDA $14
120        CMP #25
130        BCS IQERR
140        PHA
150        JSR GETWRD
160        PLA
170        TAX
180        LDA $15
190        BNE IQERR
200        LDY $14
210        CPY #40
220        BCS IQERR
230        JMP $FFF0
```

```
 240          IQERR JMP $B248
 250          END
END
```

Commentary

70-130: These lines obtain the parameter for the line number (position down the screen). Check that a comma follows and that the number falls in the range 0-24.

140-220: The line number is stored on the stack then the column number obtained and checked to see that it is in the range 0-39. The X register will be loaded with the row number and the Y register with the column number.

230: This calls the KERNAL routine which sets the cursor position. The KERNAL routine uses the X and Y registers to obtain the correct position.

240: IQERR is the address to print the ILLEGAL QUANTITY error message.

PLOT: Fully Assembled Listing

```
ADD.   DATA      SOURCE CODE
0                10 PRT
0                20 SYM
0                30 GETWRD = $C12C
0                40 ORG $C05B
C05B   D9C3      50 WRD PLOT-1
C05D             60 ORG $C3DA
C3DA   202CC1    70 PLOT JSR GETWRD
C3DD   20FDAE    80 JSR $AEFD
C3E0   A515      90 LDA $15
C3E2   D019      100 BNE IQERR
C3E4   A514      110 LDA $14
C3E6   C919      120 CMP #25
C3E8   B013      130 BCS IQERR
C3EA   48        140 PHA
C3EB   202CC1    150 JSR GETWRD
C3EE   68        160 PLA
C3EF   AA        170 TAX
C3F0   A515      180 LDA $15
C3F2   D009      190 BNE IQERR
C3F4   A414      200 LDY $14
C3F6   C028      210 CPY #40
C3F8   B003      220 BCS IQERR
C3FA   4CF0FF    230 JMP $FFF0
C3FD   4C48B2    240 IQERR JMP $B248
```

```
C400              250 END

 TOTAL ERRORS IN FILE --- 0


GETWRD             C12C
PLOT               C3DA
IQERR              C3FD
 TOTAL NUMBER OF SYMBOLS --- 3
```

PLOT: Notes On Use

PLOT can be used to replace most instructions using strings of cursor control characters, though these still remain useful for moves to positions relative to the current position of the cursor. PLOT can work upon expressions as well as straight values, such as PLOT X+3,Y/2.

The correct syntax for PLOT is:
PLOT < row down> ,< column across>

SECTION 4: Keyword Delete

Deleting lines one by one can be tiresome, especially when the DELETE command can be entered to remove a block of lines in an instant.

DELETE: Assembly Language Listing

```
10        PRT
20        ; BLOCK DELETE OF LINES
30        SYM
40        ORG $C053
50        WRD DEL-1
60        ORG $C400
70        GETWRD = $C12C
80        DEL JSR GETWRD
90        ; CONVERT TO ADDRESS
100        JSR $A613
110        BCC ULERR
120        ; SAVE POINTER ON STACK
130        LDA $5F
140        PHA
150        LDA $60
160        PHA
170        ; CHECK THAT A - SIGN FOLLOWS
```

```
 180         LDA #45
 190         JSR $AEFF
 200         ; GET LAST NO. TO BE DELETED
 210         JSR GETWRD
 220         JSR $A613
 230         BCC ULERR
 240         ; GET ADDRESS OF END OF LAST
LINE TO BE DELETED
 250         LDY #1
 260         LDA ($5F).Y
 270         TAX
 280         DEY
 290         LDA ($5F).Y
 300         TAY
 310         ; NOW STORE THESE BYTES IN FI
RST LINE TO BE DELETEED
 320         PLA
 330         STA $60
 340         PLA
 350         STA $5F
 360         TYA
 370         LDY #0
 380         STA ($5F).Y
 390         INY
 400         TXA
 410         STA ($5F).Y
 420         ; GET LINE NO. TO BE DELETED
 430         INY
 440         LDA ($5F).Y
 450         STA $14
 460         INY
 470         LDA ($5F).Y
 480         STA $15
 490         ; PUT ZERO INTO BASIC INPUT B
UFFER - TELL FILE ED. TO DELETE LINE
 500         LDA #0
 510         STA $200
 520         ; TIDY UP RETURN STACK
 530         PLA
 540         PLA
 550         ; USE ROM ROUTINE TO DELETE L
INE
 560         JMP $A4A4
 570         ULERR JMP $A8E3
```

```
 580          END
END
```

100-110: The routine at A613 hex converts the first line number picked up by GETWRD to an address in program memory. If the line number is not found then the carry flag will be clear on return and the UNDEFINED LINE error message will be called.

130-160: The line address discovered by the routine at A613 hex has been placed by that routine into 5F-60 hex. The two bytes there are now stored on the stack, since we are about to get another line address and do not wish to lose the first.

170-190: A check is made that a - follows the first line number, using the same routine that elsewhere checks for a comma but first defining the character we are searching for and jumping into the routine two bytes later than previously.

200-230: The last line number to be deleted is obtained and its address determined.

240-300: The start address of the line following the last line to be deleted is obtained from the link bytes of the last line and placed into the X and Y registers.

310-410: The address of the start of the line after the last line is now placed into the link bytes of the first line to be deleted.

420-480: The block of lines now constitutes, in the eyes of the interpreter, a single line, since its link bytes point past the end of the block. The actual line number to be deleted is now obtained from the two bytes following the link bytes.

490-510: Zero is stored in the input buffer — this indicates to the BASIC file editor that a line is to be deleted.

520-540: Having removed lines from the program we cannot return to the address from which DELETE was called, since it may now have changed, so the return address is taken from the stack and discarded.

550-560: A jump is made to the ROM routine which deletes lines — the correct line number being stored in 14 and 15 hex for the use of this routine.

570: The jump to the UNDEFINED LINE error routine.

DELETE: Fully Assembled Listing

```
ADD.   DATA      SOURCE CODE
Ø                10 PRT
Ø                20 ; BLOCK DELETE OF LINE
S
Ø                30 SYM
Ø                40 ORG $C053
C053   FFC3      50 WRD DEL-1
C055             60 ORG $C400
C400             70 GETWRD = $C12C
C400   202CC1    80 DEL JSR GETWRD
C403             90 ; CONVERT TO ADDRESS
C403   2013A6    100 JSR $A613
C406   903F      110 BCC ULERR
C408             120 ; SAVE POINTER ON STA
CK
C408   A55F      130 LDA $5F
C40A   48        140 PHA
C40B   A560      150 LDA $60
C40D   48        160 PHA
C40E             170 ; CHECK THAT A - SIGN
 FOLLOWS
C40E   A92D      180 LDA #45
C410   20FFAE    190 JSR $AEFF
C413             200 ; GET LAST NO. TO BE
DELETED
C413   202CC1    210 JSR GETWRD
C416   2013A6    220 JSR $A613
C419   902C      230 BCC ULERR
C41B             240 ; GET ADDRESS OF END
OF LAST LINE TO BE DELETED
C41B   A001      250 LDY #1
C41D   B15F      260 LDA ($5F).Y
C41F   AA        270 TAX
C420   88        280 DEY
C421   B15F      290 LDA ($5F).Y
C423   A8        300 TAY
C424             310 ; NOW STORE THESE BYT
ES IN FIRST LINE TO BE DELETEED
C424   68        320 PLA
C425   8560      330 STA $60
C427   68        340 PLA
C428   855F      350 STA $5F
```

```
C42A  98        360 TYA
C42B  A000      370 LDY #0
C42D  915F      380 STA (*5F).Y
C42F  C8        390 INY
C430  8A        400 TXA
C431  915F      410 STA (*5F).Y
C433            420 ; GET LINE NO. TO BE
DELETED
C433  C8        430 INY
C434  B15F      440 LDA (*5F).Y
C436  8514      450 STA *14
C438  C8        460 INY
C439  B15F      470 LDA (*5F).Y
C43B  8515      480 STA *15
C43D            490 ; PUT ZERO INTO BASIC
 INPUT BUFFER - TELL FILE ED. TO DELETE
LINE
C43D  A900      500 LDA #0
C43F  8D0002    510 STA *200
C442            520 ; TIDY UP RETURN STAC
K
C442  68        530 PLA
C443  68        540 PLA
C444            550 ; USE ROM ROUTINE TO
DELETE LINE
C444  4CA4A4    560 JMP *A4A4
C447  4CE3A8    570 ULERR JMP *A8E3
C44A            580 END

 TOTAL ERRORS IN FILE --- 0
GETWRD           C12C
DEL              C400
ULERR            C447
 TOTAL NUMBER OF SYMBOLS --- 3
```

DELETE: Notes On Use

Like any command which alters the structure of a program, using DELETE during program execution can create problems, so the program terminates when DELETE is executed. Delete will normally be used in direct mode but can be used as a security device in programs, for instance to remove lines which you do not wish to have examined, if the STOP key is pressed. In fact, DELETE is more effective in protecting a program in this way than NEW now that UNDEAD is available. UNDEAD cannot restore lines which have been DELETED since they are already overwritten.

The correct syntax for DELETE is:
DELETE < first line to be deleted> -< last line to be deleted>

SECTION 5: Keyword BSAVE

Now that you are, hopefully, getting a little addicted to the wonders that can be achieved with machine code, you will soon find yourself wanting to be able to save blocks of memory without having to do it through the Monitor in the Mastercode program. In this section and the next we present three new commands which will allow any area of memory to be saved, to be verified and then to be reloaded at a later date. Apart from anything else this is the easy way of loading machine code routines into memory.

BSAVE: Assembly Language Listing

```
 10          PRT
 20          GETWRD = $C12C
 30          SYM
 40          ORG $C065
 50          WRD BSAVE-1
 60          ORG $C234
 70          BSAVE JSR $E1D4
 80          JSR $AEFD
 90          JSR GETWRD
 100          LDA $14
 110          PHA
 120          LDA $15
 130          PHA
 140          JSR $AEFD
 150          JSR GETWRD
 160          LDX $14
 170          LDY $15
 180          PLA
 190          STA $15
 200          PLA
 210          STA $14
 220          LDA #$14
 230          JMP $E15F
 240          END
END
```

70: A KERNAL routine is used to pick up all the parameters of a normal file command ie name, device and secondary address.

100-140: The start of the memory area to be saved, which was picked up by GETWRD is placed onto the stack.

150-210: The finish address of the area is loaded into the X and Y registers, the start address is retrieved from the stack and placed back into 14-15 hex.

220-230: 14 hex is loaded into the accumulator to specify the address which contains the start address of the block of memory to be saved. The save itself is performed by the same KERNAL routine which performs all BASIC saving.

BSAVE: Fully Assembled Listing

```
ADD.   DATA      SOURCE CODE
0                10 PRT
0                20 GETWRD = $C12C
0                30 SYM
0                40 ORG $C065
C065   33C2      50 WRD BSAVE-1
C067             60 ORG $C234
C234   20D4E1    70 BSAVE JSR $E1D4
C237   20FDAE    80 JSR $AEFD
C23A   202CC1    90 JSR GETWRD
C23D   A514      100 LDA $14
C23F   48        110 PHA
C240   A515      120 LDA $15
C242   48        130 PHA
C243   20FDAE    140 JSR $AEFD
C246   202CC1    150 JSR GETWRD
C249   A614      160 LDX $14
C24B   A415      170 LDY $15
C24D   68        180 PLA
C24E   8515      190 STA $15
C250   68        200 PLA
C251   8514      210 STA $14
C253   A914      220 LDA #$14
C255   4C5FE1    230 JMP $E15F
C258             240 END

 TOTAL ERRORS IN FILE --- 0


GETWRD           C12C
BSAVE            C234
 TOTAL NUMBER OF SYMBOLS --- 2
```

BSAVE: Notes On Use

BSAVE always requires a secondary address, normally this will be 2. Omitting the secondary address will result in a SYNTAX ERROR message. Note also that no check is made that the end address of the block to be saved is actually after the start address.

The correct syntax for BSAVE is:
BSAVE <"filename">,<device>,<secondary address>,<start of memory area>,<end of memory area>

SECTION 6: Keywords BLOAD and BVERIFY

Having saved an area of memory to tape or disc it would be nice to think that it could be retrieved again. This is accomplished by the command BLOAD. Almost the same routine which executes BLOAD can also be used to perform BVERIFY, which checks if an area of memory is saved correctly by comparing what has been saved with the contents of the area that it was saved from.

BLOAD/BVERIFY: Assembly Language Listing

```
 10        PRT
 20        SYM
 25        GETWRD = $C12C
 30        ORG $C061
 40        WRD BLOAD-1.BVER-1
 50        ORG $C2F2
 60        BVER LDA #1
 70        BYT $2C
 80        BLOAD LDA #0
 90        STA $A
 100        JSR $E1D4
 110        JSR $AEFD
 120        JSR GETWRD
 130        LDA $A
 140        LDX $14
 150        LDY $15
 160        JMP $E175
 170        END
END
```

100-160: This is the core of the routine, though two previous sections must be explained after this one is understood. What the lines do is to jump to the KERNAL routine which gets file parameters, and calls GETWRD to

obtain the start address of the area of memory into which the saved machine code is to be loaded. The original contents of the accumulator, which indicate whether a BLOAD or a BVERIFY is to be performed, are restored, X and Y are loaded with the result of GETWRD and the KERNAL load routine is called.

60-90: These lines determine the value in the accumulator when the load routine is called — 1 will indicate a verify and 0 will mean load. If BLOAD is called then all that happens is that the accumulator is loaded with zero. If BVERIFY is called however, the accumulator is loaded with one and the next byte of the program (placed there by a BYT directive to assembler) is actually interpreted as the opcode of a three byte instruction with the two bytes of the LDA #0 instruction as its operand. This is, of course, total nonsense but the instruction that is recognised is a bit test operation which makes no change except to one or two flags in the CPU which we are not using. What this means is that the instruction at line 60 is skipped over far more quickly and economically than if a jump were made — the instruction simply disappears when approached from this direction.

BLOAD/BVERIFY: Fully Assembled Listing

```
ADD.   DATA      SOURCE CODE
0                10 PRT
0                20 SYM
0                25 GETWRD = $C12C
0                30 ORG $C061
C061   F4C2F1    40 WRD BLOAD-1.BVER-1
C065             50 ORG $C2F2
C2F2   A901      60 BVER LDA #1
C2F4   2C        70 BYT $2C
C2F5   A900      80 BLOAD LDA #0
C2F7   850A      90 STA $A
C2F9   20D4E1    100 JSR $E1D4
C2FC   20FDAE    110 JSR $AEFD
C2FF   202CC1    120 JSR GETWRD
C302   A50A      130 LDA $A
C304   A614      140 LDX $14
C306   A415      150 LDY $15
C308   4C75E1    160 JMP $E175
C30B             170 END

 TOTAL ERRORS IN FILE --- 0

GETWRD            C12C
BVER              C2F2
```

```
BLOAD              C2F5
 TOTAL NUMBER OF SYMBOLS --- 3
```

BLOAD/BVERIFY: Notes On Use

Once again, a secondary address must be used and this should be zero. BLOAD will load back to the address in memory which you specify. BVERIFY will verify the correct area of memory no matter what is entered as an address — a dummy address must, however, be included.

The correct syntax for BLOAD is:
BLOAD <"filename">,<device>,(secondary address=0>,<start of area to which load is to be made>

The correct syntax for BVERIFY is:
BVERIFY <"filename">, <device>, <secondary address=0>, <dummy value>

SECTION 7: Keyword Move

What you are about to enter, although it may seem a little dry at first, is one of the most flexible commands that can be added to BASIC. MOVE allows you specify an area of memory and then to shift that block to another starting point. Note that the areas are not swapped, what you end up with is two copies of the source block. The command can be useful to machine code programmers who wish to relocate a routine without having to save and reload it. It can even be used for manipulating the screen by copying areas of screen from one place to another. The routine is longer than most you have entered so far but it is really very simple in execution.

MOVE: Assembly Language Listing

```
 10        PRT
 20        GETWRD = $C12C
 30        SYM
 40        ORG $C055
 50        WRD MOVE-1
 60        ORG $C30B
 70        ; BLOCK MOVE OF MEMORY - NO PR
OTECTION AGAINST MOVING VITAL SECTIONS
 80        ; SYNTAX OF COMMAND 'MOVE A1.A
2.L'
 90        ; WHERE A1 = ORGINAL ADDRESS
 100        ;        A2 = FIANL ADDRESS
 110        ;       L  = LENGTH OF BLOCK
 120        ; ALSO NOTE 32K BLOCKS MAX
```

```
 130         NEWADD = $61
 140         OLDADD = NEWADD+2
 150         LENGTH = $14
 160         ; SUBROUTINE TO DECREMENT & T
EST LENGTH
 170         DECLEN LDA LENGTH
 180         BNE LBL000
 190         DEC LENGTH+1
 200         LBL000 DEC LENGTH
 210         LDA LENGTH
 220         ORA LENGTH+1
 230         RTS
 240         ; MAIN ROUTINE
 250         MOVE JSR GETWRD
 260         LDA $14
 270         PHA
 280         LDA $15
 290         PHA
 300         JSR $AEFD
 310         JSR GETWRD
 320         LDA $14
 330         PHA
 340         LDA $15
 350         PHA
 360         JSR $AEFD
 370         JSR GETWRD
 380         ; DECIDE WHICH DIRECTION TO M
OVE IN
 390         LDY #3
 400         LBL001 PLA
 410         STA NEWADD.Y
 420         DEY
 430         BPL LBL001
 440         LDA LENGTH
 450         ORA LENGTH+1
 460         BEQ LBL002
 470         LDA NEWADD+1
 480         CMP OLDADD+1
 490         BCC MVEDWN
 500         BNE MVEUP
 510         LDA NEWADD
 520         CMP OLDADD
 530         BCC MVEDWN
 540         ; MOVE BLOCK UPWARDS IN MEMOR
```

```
Y
 550          MVEUP CLD
 560          CLC
 570          LDA NEWADD
 580          ADC LENGTH
 590          STA NEWADD
 600          LDA NEWADD+1
 610          ADC LENGTH+1
 620          STA NEWADD+1
 630          CLC
 640          LDA OLDADD
 650          ADC LENGTH
 660          STA OLDADD
 670          LDA OLDADD+1
 680          ADC LENGTH+1
 690          STA OLDADD+1
 700          LDY #0
 710          LBL003 LDA (OLDADD),Y
 720          STA (NEWADD),Y
 730          TYA
 740          BNE LBL004
 750          DEC OLDADD+1
 760          DEC NEWADD+1
 770          LBL004 DEY
 780          JSR DECLEN
 790          BNE LBL003
 800          LBL002 RTS
 810          ; MOVE BLOCK DOWN THE MEMORY
 820          MVEDWN LDY #0
 830          LBL005 LDA (OLDADD),Y
 840          STA (NEWADD),Y
 850          INY
 860          BNE LBL006
 870          INC OLDADD+1
 880          INC NEWADD+1
 890          LBL006 JSR DECLEN
 900          BNE LBL005
 910          RTS
END
```

Commentary

160-230: When you specify an area of memory to be MOVEd it will, self-evidently, have a length. The purpose of these lines is to decrement a varia-

ble called LENGTH to record how much of the block has been MOVEd so far. LENGTH is in fact a two byte variable and the lines whether the low byte is zero in order to decide whether to decrement the low byte only, or to decrement both bytes — both bytes are only decremented if the low byte *is* zero, to represent a 'carry'.

240-370: The three parameters for start, finish and new address are obtained by use of GETWRD.

380-530: Before making a MOVE we must know whether the destination is up the memory or down. If it is down (ie negative in terms of address) we shall have to start copying the area from the bottom up so that if the two areas overlay each other, by the time the destination area begins to encroach on the source area, we no longer need the data at the beginning of the source area. The opposite is true when the destination area is up the memory from the source. Thus, if we wished to move a block of memory one byte upwards we would start with the last byte in the source area and MOVE it one place upwards. Starting at the beginning of the source area would mean that the first byte would be placed into the position of the second, then the second would be copied to the third — in fact we would place the same character in the whole of the destination block. Between 470 and 530 the lines perform a 16 bit comparison between the two start addresses and jump to MVEDWN if the destination address is less than the source start address, otherwise MOVEUP is executed.

550-690: This is the start of the routine to MOVE a block up the memory. These lines obtain the addresses of the end of each block and store them in NEWADD and OLDADD, these having previously held the *start* addresses.

700-800: Using the Y register to index the MOVE, these lines begin shifting bytes from the address specified by OLDADD plus the contents of the Y register to the address specified by NEWADD plus the Y register. The Y register is decremented on each transfer and whenever the contents of Y reach zero the high byte of OLDADD and NEWADD are decremented by one to access a new block of 256 bytes. After each decrement of the Y register a jump is made to the subroutine DECLEN, which decides whether the full length of the block has been moved. If it has, then on return the zero flag will be set to zero and line 800 will be reached, ending the routine.

810-910: This is the routine to MOVE a block down the memory. It is simpler because the contents of NEWADD and OLDADD can be left pointing to the start of their respective blocks. Other than that, the only real difference between the two routines is that the Y register is incremented rather than decremented.

MOVE: Fully Assembled Listing

```
ADD.   DATA      SOURCE CODE
Ø                10 PRT
Ø                20 GETWRD = $C12C
Ø                30 SYM
Ø                40 ORG $CØ55
CØ55   17C3      50 WRD MOVE-1
CØ57             60 ORG $C30B
C30B             70 ; BLOCK MOVE OF MEMORY
 - NO PROTECTION AGAINST MOVING VITAL SE
CTIONS
C30B             80 ; SYNTAX OF COMMAND 'M
OVE A1.A2.L'
C30B             90 ; WHERE A1 = ORGINAL A
DDRESS
C30B            100 ;        A2 = FIANL AD
DRESS
C30B            110 ;       L  = LENGTH OF
 BLOCK
C30B            120 ; ALSO NOTE 32K BLOCK
S MAX
C30B            130 NEWADD = $61
C30B            140 OLDADD = NEWADD+2
C30B            150 LENGTH = $14
C30B            160 ; SUBROUTINE TO DECRE
MENT & TEST LENGTH
C30B   A514     170 DECLEN LDA LENGTH
C30D   D002     180 BNE LBLØØØ
C30F   C615     190 DEC LENGTH+1
C311   C614     200 LBLØØØ DEC LENGTH
C313   A514     210 LDA LENGTH
C315   Ø515     220 ORA LENGTH+1
C317   60       230 RTS
C318             240 ; MAIN ROUTINE
C318   202CC1   250 MOVE JSR GETWRD
C31B   A514     260 LDA $14
C31D   48       270 PHA
C31E   A515     280 LDA $15
C320   48       290 PHA
C321   20FDAE   300 JSR $AEFD
C324   202CC1   310 JSR GETWRD
C327   A514     320 LDA $14
C329   48       330 PHA
```

```
C32A  A515     340 LDA $15
C32C  48       350 PHA
C32D  20FDAE   360 JSR $AEFD
C330  202CC1   370 JSR GETWRD
C333           380 ; DECIDE WHICH DIRECT
ION TO MOVE IN
C333  A003     390 LDY #3
C335  68       400 LBL001 PLA
C336  996100   410 STA NEWADD.Y
C339  88       420 DEY
C33A  10F9     430 BPL LBL001
C33C  A514     440 LDA LENGTH
C33E  0515     450 ORA LENGTH+1
C340  F03C     460 BEQ LBL002
C342  A562     470 LDA NEWADD+1
C344  C564     480 CMP OLDADD+1
C346  9037     490 BCC MVEDWN
C348  D006     500 BNE MVEUP
C34A  A561     510 LDA NEWADD
C34C  C563     520 CMP OLDADD
C34E  902F     530 BCC MVEDWN
C350           540 ; MOVE BLOCK UPWARDS
IN MEMORY
C350  D8       550 MVEUP CLD
C351  18       560 CLC
C352  A561     570 LDA NEWADD
C354  6514     580 ADC LENGTH
C356  8561     590 STA NEWADD
C358  A562     600 LDA NEWADD+1
C35A  6515     610 ADC LENGTH+1
C35C  8562     620 STA NEWADD+1
C35E  18       630 CLC
C35F  A563     640 LDA OLDADD
C361  6514     650 ADC LENGTH
C363  8563     660 STA OLDADD
C365  A564     670 LDA OLDADD+1
C367  6515     680 ADC LENGTH+1
C369  8564     690 STA OLDADD+1
C36B  A000     700 LDY #0
C36D  B163     710 LBL003 LDA (OLDADD).Y
C36F  9161     720 STA (NEWADD).Y
C371  98       730 TYA
C372  D004     740 BNE LBL004
C374  C664     750 DEC OLDADD+1
```

```
C376  C662     760 DEC NEWADD+1
C378  88       770 LBL004 DEY
C379  200BC3   780 JSR DECLEN
C37C  D0EF     790 BNE LBL003
C37E  60       800 LBL002 RTS
C37F           810 ; MOVE BLOCK DOWN THE
 MEMORY
C37F  A000     820 MVEDWN LDY #0
C381  B163     830 LBL005 LDA (OLDADD).Y
C383  9161     840 STA (NEWADD).Y
C385  C8       850 INY
C386  D004     860 BNE LBL006
C388  E664     870 INC OLDADD+1
C38A  E662     880 INC NEWADD+1
C38C  200BC3   890 LBL006 JSR DECLEN
C38F  D0F0     900 BNE LBL005
C391  60       910 RTS

 TOTAL ERRORS IN FILE --- 0


GETWRD          C12C
NEWADD          61
OLDADD          63
LENGTH          14
DECLEN          C30B
LBL000          C311
MOVE            C318
LBL001          C335
MVEUP           C350
LBL003          C36D
LBL004          C378
LBL002          C37E
MVEDWN          C37F
LBL005          C381
LBL006          C38C
 TOTAL NUMBER OF SYMBOLS --- 15
```

MOVE: Notes On Use

The use of MOVE is quite straightforward but remember that no protection is provided against you doing something stupid with it, like accidentally overwriting the interpreter (when it is in RAM) or system variables,

or the program area or..... Make sure you know what you are moving and what is in the place you are sending it to BEFORE you do it.

The correct syntax for MOVE is:

MOVE < address to move to> ,< address to move from> ,< length>

SECTION 8: Keyword FILL

Having designed MOVE, the logical development was FILL, which is used to fill a specified area of memory with a specified value. It can be used to clear areas of memory, to clear areas of the screen, to change the colour characteristics of the screen by filling parts of the attributes file. The real work is done by a call to the MVEDWN routine in MOVE itself, so there is little to explain about the working of the command.

FILL: Assembly Language Listing

```
10        PRT
20        SYM
30        ORG $C067
40        WRD FILL-1
50        GETWRD = $C12C
60        MVEDWN = $C37F
70        DECLEN = $C30B
80        ORG $C392
90        FILL JSR GETWRD
100        LDA $14
110        PHA
120        LDA $15
130        PHA
140        JSR $AEFD
150        JSR GETWRD
160        LDA $14
170        PHA
180        LDA $15
190        PHA
200        JSR $AEFD
210        JSR GETWRD
220        LDA $15
230        BNE IQERR
240        LDX $14
250        PLA
260        STA $15
270        PLA
```

```
 280          STA $14
 290          PLA
 300          STA $62
 310          STA $64
 320          PLA
 330          STA $61
 340          STA $63
 350          INC $61
 360          BNE LBL000
 370          INC $62
 380          LBL000 LDY #0
 390          TXA
 400          STA ($63),Y
 410          JSR DECLEN
 420          BEQ EXIT
 430          JMP MVEDWN
 440          IQERR JMP $B248
 450          EXIT RTS
 460          END
END
```

Commentary

100-230: These lines obtain the three parameters of start address, length and the value to be placed into each byte in the block. An illegal quantity error is generated if the value to be loaded is greater than 255.

240: The value to be stored in the block is saved in the X register.

250-280: The length of the block is stored back into 14-15 hex, where the move routine expects to find it.

290-370: The start address is stored in OLDADD and the start address plus one in NEWADD.

380-400: The value to be used in filling the block is placed into the first byte of the block.

410-420: The length is decremented by one and if the length was only one anyway, the routine is terminated.

430: Calls MVEDWN. If you remember anything of the commentary on MOVE you will know that moving a block one byte up the memory means a call to MOVEUP. Calling MOVDWN for the purpose performs the horrendous act of corrupting the very bytes that are to be transferred, constantly rewriting byte one to byte two, byte two to byte

three and so on, filling the whole area with the same value. In MOVE this would have been a disaster but it is also exactly what we want for FILL.

FILL: Fully Assembled Listing

```
ADD.   DATA      SOURCE CODE
Ø                10 PRT
Ø                20 SYM
Ø                30 ORG $CØ67
CØ67   91C3      40 WRD FILL-1
CØ69             50 GETWRD = $C12C
CØ69             60 MVEDWN = $C37F
CØ69             70 DECLEN = $C3ØB
CØ69             80 ORG $C392
C392   2Ø2CC1    90 FILL JSR GETWRD
C395   A514      100 LDA $14
C397   48        110 PHA
C398   A515      120 LDA $15
C39A   48        130 PHA
C39B   2ØFDAE    140 JSR $AEFD
C39E   2Ø2CC1    150 JSR GETWRD
C3A1   A514      160 LDA $14
C3A3   48        170 PHA
C3A4   A515      180 LDA $15
C3A6   48        190 PHA
C3A7   2ØFDAE    200 JSR $AEFD
C3AA   2Ø2CC1    210 JSR GETWRD
C3AD   A515      220 LDA $15
C3AF   DØ25      230 BNE IQERR
C3B1   A614      240 LDX $14
C3B3   68        250 PLA
C3B4   8515      260 STA $15
C3B6   68        270 PLA
C3B7   8514      280 STA $14
C3B9   68        290 PLA
C3BA   8562      300 STA $62
C3BC   8564      310 STA $64
C3BE   68        320 PLA
C3BF   8561      330 STA $61
C3C1   8563      340 STA $63
C3C3   E661      350 INC $61
C3C5   DØØ2      360 BNE LBLØØØ
C3C7   E662      370 INC $62
C3C9   AØØØ      380 LBLØØØ LDY #Ø
```

```
C3CB   8A         390 TXA
C3CC   9163       400 STA (*63),Y
C3CE   200BC3     410 JSR DECLEN
C3D1   F006       420 BEQ EXIT
C3D3   4C7FC3     430 JMP MVEDWN
C3D6   4C48B2     440 IQERR JMP *B248
C3D9   60         450 EXIT RTS
C3DA              460 END

 TOTAL ERRORS IN FILE --- 0


GETWRD             C12C
MVEDWN             C37F
DECLEN             C30B
FILL               C392
LBL000             C3C9
IQERR              C3D6
EXIT               C3D9
 TOTAL NUMBER OF SYMBOLS --- 7
```

FILL: Notes On Use

Once again this command provides no protection against stupidity. The correct syntax for FILL is:

FILL< start address> ,< length> ,< byte value>

SECTION 9: Keyword RESTORE

You may think, on seeing the heading of this section that we have taken leave of our senses. Isn't there already a RESTORE command in normal BASIC? The answer, of course, is yes but one of the fascinating possibilities opened up by moving the interpreter into RAM is that not only can we add new comands, we can change existing ones.

RESTORE is a prime candidate for such alteration. The normal RESTORE routine dates from the time when computers were kept in huge air conditioned vaults, reading their programs and inputting their data from punched cards. Now the thing about a stack of punched cards is that you can only really read from the beginning. If you want to find the 97th card, you have to begin at one and read through 96 cards that you are not at all interested in. There is absolutely no reason why this should be true on a modern micro, yet somehow the convention seems to have stuck that the only way to deal with DATA

statements is to start at the beginning and to work through to the item that you want.

In this section we shall modify the normal RESTORE command so that you will be able to RESTORE to a specified line number and pick up the first item of DATA which follows it (the normal RESTORE command can still be used when required). In this way you can format your DATA into separate tables and jump to exactly the table you want, thus saving considerably on the time taken to access individual items of DATA and also making the functioning of your program more transparent.

RESTORE: Assembly Language Listing

```
 10        PRT
 20        SYM
 30        ORG $C132
 40        ; ALTER TO RESTORE TO LINE NOS
.
 50        ; TO USE TRANSFER ROM TO RAM A
ND ALTER RESTORE VECTOR TO 'START-1'
 60        ; RESTORE VECTOR AT $A022
 70        GETWRD = $C12C
 80        START LDA #0
 90        STA $14
 100        STA $15
 110        JSR $73
 120        LDA $7A
 130        BNE LBL000
 140        DEC $7B
 150        LBL000 DEC $7A
 160        BCS LBL001
 170        JSR GETWRD
 180        LBL001 JSR $A613
 190        LDA $14
 200        ORA $15
 210        BEQ LBL002
 220        BCC ULERR
 230        LBL002 LDA $5F
 240        LDY $60
 250        SEC
 260        SBC #1
 270        JMP $A824
 280        ULERR JMP $A8E3
 290        END
END
```

Commentary

80-100: Initialise the location in which the interpreter will later store a line number.

110-150: The subroutine call gets the next character in BASIC and the remaining lines back up the text pointer which has now moved to the position after that character.

160: If the character picked up by the routine at 73 hex is a digit, the carry flag will be set on return. If it is not set then a jump will be made past GETWRD since the routine will assume that a normal RESTORE is to be executed and there is no line number to be obtained. If you wish to use an expression after RESTORE you will need to precede it with 00+ or 01* in order to ensure that GETWRD is called.

180: This routine finds the address of the line number obtained by GETWRD or, if GETWRD has not been called, of the first line in the BASIC program.

190-210: If the line number storage area contains zero, it is assumed that a normal RESTORE has been executed and no checks are made for parameter errors.

220: If, on return from A613, the carry flag is clear then the line number referred to has not been found and an UNDEFINED LINE error is flagged.

230-270: The address of the line which has been found is picked up from 5F-60 hex, 1 is subtracted from this and execution returns to the normal RESTORE routine, but with the data pointer now set to the line specified.

RESTORE: Fully Assembled Listing

```
ADD.   DATA      SOURCE CODE
Ø                10 PRT
Ø                20 SYM
Ø                30 ORG $C132
C132             40 ; ALTER TO RESTORE TO
LINE NOS.
C132             50 ; TO USE TRANSFER ROM
TO RAM AND ALTER RESTORE VECTOR TO 'STAR
T-1'
C132             60 ; RESTORE VECTOR AT $A
Ø22
C132             70 GETWRD = $C12C
C132  A9ØØ      80 START LDA #Ø
C134  8514       90 STA $14
C136  8515       100 STA $15
```

```
C138  207300    110 JSR $73
C13B  A57A      120 LDA $7A
C13D  D002      130 BNE LBL000
C13F  C67B      140 DEC $7B
C141  C67A      150 LBL000 DEC $7A
C143  B003      160 BCS LBL001
C145  202CC1    170 JSR GETWRD
C148  2013A6    180 LBL001 JSR $A613
C14B  A514      190 LDA $14
C14D  0515      200 ORA $15
C14F  F002      210 BEQ LBL002
C151  900A      220 BCC ULERR
C153  A55F      230 LBL002 LDA $5F
C155  A460      240 LDY $60
C157  38        250 SEC
C158  E901      260 SBC #1
C15A  4C24A8    270 JMP $A824
C15D  4CE3A8    280 ULERR JMP $A8E3
C160            290 END

 TOTAL ERRORS IN FILE --- 0


GETWRD          C12C
START           C132
LBL000          C141
LBL001          C148
LBL002          C153
ULERR           C15D
 TOTAL NUMBER OF SYMBOLS --- 6
```

RESTORE: Notes On Use

RESTORE can be used normally by not specifying a parameter or by specifying zero as the parameter. One limitation is that the extended RESTORE will only work on parameters with more than one digit. If you wish to restore to a line number less than 10, you must specify the line number with a leading zero.

The correct syntax for RESTORE, in its extended form, is:
RESTORE < line number with at least two digits>

Before using the extended RESTORE, one further alteration must be made to the BASIC Extender II program. The following lines must be added:

250 REM ALTER RESTORE VECTOR
251 POKE 40996,49: POKE 40997,193

The machine code routine should be assembled over the existing machine code file of extender and keyword routines and then loaded into the memory using the newly modified BASIC Extender II program.

CHAPTER 10
BASIC Functions

BASIC functions, you may recall, receive a separate treatment by the interpreter, being identified when they are encountered in a program by the fact that they occupy a clearly specified segment of the keyword table. The main difference between the action keywords and the functions is that functions will require that something be evaluated, for which there are a whole series of special interpreter routines. In fact around half the interpreter is devoted to the problem of evaluating expressions in one way or another. When executing a function keyword it is not a simple matter of jumping to a single machine code routine in the memory.

The machine code routine given here, although it is short, uses routines in the interpreter which are far more complex than those used by the action keywords. Without those routines already in the interpreter, it would have been an enormous undertaking to define new functions.

Extend Expression Evaluator: Assembly Language Listing

```
 10        PRT
 20        SYM
 30        ORG $C44A
 40        ; EXTEND EXPRESSION EVALUATOR
 50        ; TO USE POKE $AFAA WITH 'JMP
FUNEVL'
 60        FUNEVL CPX #$8F
 70        BCC LBL002
 80        CPX #$98
 90        BCC LBL001
 100        CPX #$9F
 110        BCS LBL001
 120        JSR $AEF1
 130        PLA
 140        TAX
 150        CPX #$98
 160        BEQ DEEK
 170        CPX #$9A
 180        BEQ YPOS
```

```
 190          BNE VARPTR
 200          LBL001 JMP $AFB1
 210          LBL002 JMP $AFD1
 220          VARPTR LDA 72
 230          LDY 71
 240          LBL003 JSR $B391
 250          LDA $66
 260          BPL LBL004
 270          LDY #CONST/256
 280          LDA #CONST-CONST/256*256
 290          JSR $BA8C
 300          JSR $B86A
 310          LBL004 JMP $AD8D
 320          YPOS SEC
 330          JSR $FFF0
 340          TXA
 350          TAY
 360          LDA #$0
 370          JMP LBL003
 380          ; PERFORM DEEK
 390          DEEK JSR $B7F7
 400          LDY #1
 410          LDA ($14),Y
 420          PHA
 430          DEY
 440          LDA ($14),Y
 450          TAY
 460          PLA
 470          JMP LBL003
 480          CONST BYT 145,0,0,0,0
 490          END
END
```

60-70: This checks to see whether the token is that of a normal numeric function. If so the normal function evaluator is called.

80-110: A check is made to see if the function, whose token is in the X register, is in the range of our three new functions and if not the string function evaluator is called.

120: This interpreter routine evaluates an expression within brackets ie the argument of the function.

130-140: The expression evaluator called in the previous line places the token onto the stack. It is now retrieved and stored in the X register.

150-190: At this point the token must be that of one of our new functions, these lines determine which and jump to the appropriate routine.

220-310: VARPTR: This function returns a pointer to the location in memory of any variable whose name is placed into brackets as the argument.

220-230: On return from the routine at AEF1 which evaluated the variable within the brackets, the *address* of the variable is contained in 47-48 hex (71-72 decimal). The two byte value is loaded into the accumulator and Y register.

240: This address is now sent to the routine which converts this integer number to floating point. This is necessary because the interpreter will expect the result of a function to be a floating point number and will deal with it accordingly.

250-300: Unfortunately, the floating point converter will change the unsigned integer number to a signed floating point one. This will mean that numbers above 32767 will actually come out as negative eg the FRE function in normal BASIC. If you enter FRE(0) when there is no program in the memory then a negative quantity will be returned, to which 65536 must be added to get the correct result. These lines test for a minus sign stored in the sign byte of the floating point accumulator #1 (at 66 hex in zero page memory). If a minus is found there, (ie bit 7 is set) then the accumulator and Y register are loaded with the address of the variable CONSTANT (line 480) which is actually 65536 in floating point format. This value is now placed into floating point accumulator #2 by a call to the interpreter routine at BA8C hex and added to the value in floating point accumulator one by a call to the routine at B86A.

310: A return is made to the expression evaluator, which may still be in the middle of evaluating a larger expression of which VARPTR is only a part.

320-370: YPOS: This new function returns the current position of the cursor down the screen. It is parallel to the normal BASIC command POS. which returns the position across the screen.

330: The kernal routine which returns the position of the cursor on the screen in the X and Y registers.

340-350: The contents of the X register are stored in the Y register (the X register originally holds the vertical position).

360-370: The accumulator and Y register now hold the same value. The accumulator is loaded with zero,thus providing, in the accumulator and Y register, a 16 bit number in the range 0-24 (the line numbers on the screen).

This 16 bit integer number is now sent to the routine at LBL003 which converts it to floating point.

390-470: DEEK: This function returns a number in the range 0-65535. It is actually a Double PEEK and equivalent to the normal BASIC statement PEEK(X) + 256*PEEK(X + 1).

390: The parameter evaluated by the function evaluator is converted to an integer number by this call (PEEKs and DEEKs are to integer addresses).

400-460: The value returned is to be found in our old friends 14-15 hex. The accumulator and the Y register are loaded with this value and a jump is made to LBL003, thus returning the contents of the two bytes to the expression evaluator.

Extend Expression Evaluator: Fully Assembled Listing

```
ADD.   DATA      SOURCE CODE
Ø                10 PRT
Ø                20 SYM
Ø                30 ORG $C44A
C44A             40 ; EXTEND EXPRESSION EV
ALUATOR
C44A             50 ; TO USE POKE $AFAA WI
TH 'JMP FUNEVL'
C44A  E08F       60 FUNEVL CPX #$8F
C44C  901A       70 BCC LBL002
C44E  E098       80 CPX #$98
C450  9013       90 BCC LBL001
C452  E09F       100 CPX #$9F
C454  B00F       110 BCS LBL001
C456  20F1AE     120 JSR $AEF1
C459  68         130 PLA
C45A  AA         140 TAX
C45B  E098       150 CPX #$98
C45D  F02F       160 BEQ DEEK
C45F  E09A       170 CPX #$9A
C461  F020       180 BEQ YPOS
C463  D006       190 BNE VARPTR
C465  4CB1AF     200 LBL001 JMP $AFB1
C468  4CD1AF     210 LBL002 JMP $AFD1
C46B  A548       220 VARPTR LDA 72
C46D  A447       230 LDY 71
C46F  2091B3     240 LBL003 JSR $B391
C472  A566       250 LDA $66
C474  100A       260 BPL LBL004
```

```
C476  A0C4      270 LDY #CONST/256
C478  A99E      280 LDA #CONST-CONST/256*
256
C47A  208CBA    290 JSR $BA8C
C47D  206AB8    300 JSR $B86A
C480  4C8DAD    310 LBL004 JMP $AD8D
C483  38        320 YPOS SEC
C484  20F0FF    330 JSR $FFF0
C487  8A        340 TXA
C488  A8        350 TAY
C489  A900      360 LDA #$0
C48B  4C6FC4    370 JMP LBL003
C48E            380 ; PERFORM DEEK
C48E  20F7B7    390 DEEK JSR $B7F7
C491  A001      400 LDY #1
C493  B114      410 LDA ($14),Y
C495  48        420 PHA
C496  88        430 DEY
C497  B114      440 LDA ($14),Y
C499  A8        450 TAY
C49A  68        460 PLA
C49B  4C6FC4    470 JMP LBL003
C49E  910000    480 CONST BYT 145,0,0,0,0
C4A3            490 END

 TOTAL ERRORS IN FILE --- 0

FUNEVL          C44A
LBL001          C465
LBL002          C468
VARPTR          C46B
LBL003          C46F
LBL004          C480
YPOS            C483
DEEK            C48E
CONST           C49E
 TOTAL NUMBER OF SYMBOLS --- 9
```

VARPTR: Notes On Use

If an expression is put into the brackets instead of a variable the result is meaningless and should not be used as the basis for any changes to the memory. Other than this the function can be used as a short-cut to changing individual characters in string arrays (avoiding garbage collection problems and complex string functions) or simply to get a more accurate idea of what is happening in the variables area. As an example of the

use of VARPTR, enter A% = 10 in direct mode, then PRINT DEEK (VARPTR(A%) + 1). This returns the value of A%. In the case of strings PEEK (VARPTR(A$)) will return the length of A$.
DEEK(VARPTR(A$) + 1) returns the start address of A$ in the memory.

The correct syntax for VARPTR is:
VARPTR (< variable name>)

YPOS: Notes On Use

This function is parallel to POS in normal BASIC, including the fact that the variable specified in the brackets is ignored.

The correct syntax for YPOS is:
YPOS (< dummy argument>)

DEEK: Notes On Use

Very much the same as PEEK except that a two byte value is returned. DEEK is particularly useful in accessing the values stored in two byte registers used by the system eg DEEK (43) returns the address of the start of BASIC.

The correct syntax for DEEK is:
DEEK (< expression>)

Running the Function Extender

As with the extended BASIC keywords, this routine must be assembled over the machine code file that you have built up to include the BASIC extender and the other commands. The routine could stand alone without crashing the system if it were loaded into memory but you would be unable to crunch the token for the new functions.

In order to patch the routine into the existing function evaluator, the following line must be added to your existing BASIC Extender II program:
224 DATA 173,175,76,174,175,74,175,175,196

The effect of this line is to place into the function evaluator a jump to our machine code routine very much as was done for the action keyword execution routine. Once the change is made the function evaluator can be loaded, along with the other commands, by the BASIC Extender II program.

CHAPTER 11
Breaking New Frontiers

When you sit down to write a book such as this one, you toss around all kinds of ideas for the routines that you would like to include. Some prove to be too extensive in the amount of coding they require, others seem irrelevant on further consideration. Some however, stand out as being useful ideas that should not present too many difficulties in their implementation. One such idea that came to us —admittedly it was very late at night — was FAST, a routine which would capitalise on RKILL by removing the interpreter's check for spaces when executing a program. The 64 was switched on, the routine entered and assembled, adding two new keywords FAST and SLOW (the latter simply restores the normal state of affairs). The routine turned out as follows:

FAST and SLOW: Assembly Language Listing

```
 10          PRT
 20          SYM
 30          ORG $C057
 40          WRD FAST-1,SLOW-1
 45          ORG $C4A3
 50          SLOW LDY #0
 60          BYT $2C
 70          FAST LDY #4
 80          LDX #0
 90          LBL000 LDA $E3AF,Y
 100          STA $80,X
 110          INX
 120          INY
 130          CPY #$B
 140          BCC LBL000
 150          RTS
END
```

The BASIC Extender program was loaded and FAST placed into the memory. The memory was cleared and a small program entered which exe-

cuted a large loop, the total time taken being around 57.75 seconds. RKILL was executed on the program, removing all the spaces and then FAST was entered in direct mode. Not a whimper from the 64, the routine was obviously going to work first time — a triumph for proper planning and program design. The loop was run again, with the stopwatch held in trembling hands.

The result was a time of 57.25 seconds, an improvement of 0.89%!

Not everything that you can do in machine code is actually worth doing, and not everything that is worth doing can be done. At the end of this book we are left with the feeling that what we have created *is* worth doing and that other people will be able both to do it and to learn from it. In entering the Mastercode program and the keyword routines (and understanding them) we hope you will have taken a considerable step forward in understanding your 64 and the potential it offers for machine code programming. May your efforts be more successful than FAST.

APPENDICES

APPENDIX A
Checksum Generator

The program given below is the one which was used to generate the checksum tables provided with each module of the Mastercode program. The checksum figure for each line is an almost foolproof method of indicating whether a line has been entered correctly. The program works by adding together the values of all the bytes in the line, rounding down to zero every time 255 is reached. An incorrect character or a character omitted will result in the checksum changing. To make use of the checksums, enter this program into the 64 *before* you begin on the Mastercode program and then RUN 63800 every time you wish to check a batch of lines that you have entered, then compare the results with the Checksum table for the module. If there *is* a difference then the line you have entered is not the same as the line given in the book. Note that spaces count in the calculation of the checksums.

```
63800 REM CHECKSUM PROGRAM
63801 GOSUB 63810
63802 GOSUB 63840
63803 IF FL>=0 THEN 63802
63804 END
63810 DEFFN DEEK(X) = PEEK(X)+256*PEEK(X
+1)
63820 REM DATA FOR MACHINE CODE
63821 DATA ***
63822 DATA 165,252,166,253,133,020,134,0
21,032,019
63823 DATA 166,216,160,001,177,095,133,2
54,240,013
63824 DATA 200,177,095,133,252,200,177,0
95,133,253
63825 DATA 200,169,000,133,251,177,095,2
40,006,024
63826 DATA 101,251,200,208,244,096
63827 DATA -1
```

```
63830 REM PUT DATA INTO MEMORY
63831 AD = 52992
63832 RESTORE
63833 READ T$: IF T$<>"***" THEN 63833
63834 READ T : IF T>=0 THEN POKE AD,T :
AD = AD+1 : GOTO 63834
63835 DEV = 3 : IN$ = "" : INPUT "OUTPUT
 DEVICE NUMBER "; DEV
63836 IF DEV=1 OR DEV>4 THEN INPUT "FILE
 NAME ";IN$
63837 R$ = CHR$(13) : S$ = "************
**************************"+R$
63838 RETURN
63840 REM DO INITALISATION
63841 FL = 0 : INPUT "FIRST LINE "; FL :
 IF FL<0 THEN RETURN
63842 LL = 65536 : INPUT "LAST LINE "; L
L
63843 INPUT "MODULE NAME ";M$
63844 OPEN 1,DEV,2,IN$
63845 PRINT#1,S$ R$SPC((40-LEN(M$))/2)M$
R$ R$"LINE NUMBERS"FL"TO"LL;R$S$R$
63850 REM ACTUAL PROGRAM
63851 LN = FL : C = 0 : C1 = 0
63852 POKE 252,LN-INT(LN/256)*256 : POKE
 253,LN/256
63853 SYS 52992 : CS = PEEK(251) : LN =
FNDEEK(252)+1
63860 REM FORMAT OUTPUT INTO 3 COLUMNS
63861 T$ = LEFT$(STR$(LN-1)+"        ",6)+
LEFT$(STR$(CS)+"         ",7)
63862 PRINT#1,T$;
63864 C = C+1 : IF C>=3 THEN PRINT#1 : C
 = 0 : C1 = C1 + 1
63865 IF C1>=20 AND DEV=3 THEN C1 = 0 :
GOSUB 63998
63866 IF LN<=LL AND PEEK(254) THEN 63852
63867 CLOSE 1 : RETURN
63898 GET T$ : IF T$="" THEN 63998
63899 RETURN

READY.
```

CHECKSUM TABLE

```
63800 58      63801 175     63802 178
63803 186     63804 128     63810 179
63820 32      63821 33      63822 30
63823 38      63824 46      63825 31
63826 14      63827 1       63830 53
63831 130     63832 140     63833 176
63834 38      63835 214     63836 104
63837 145     63838 142     63840 58
63841 3       63842 133     63843 188
63844 64      63845 105     63850 161
63851 13      63852 207     63853 255
63860 119     63861 189     63862 168
63864 79      63865 78      63866 206
63867 249     63898 198     63899 142
```

APPENDIX B
Mastercode User Guide

The Mastercode program is divided into four sections: Monitor, Disassembler, File Editor and Assembler, all of which are fully compatible with each other. On running Mastercode there will be a noticeable wait while the complex tables for the Disassembler and Assembler are generated. The first section of the program available to the user is the Monitor — the Assembler Disassembler and File Editor are all called from the Monitor control subroutine as menu options.

Monitor

Use of the Monitor is straightforward. Simply follow the prompts given when the program is RUN. The Disassembler and Assembler both return to the Monitor when they have completed their current assignment. When using the File Editor, the Monitor is menu option number 0.

Disassembler

The disassembler is capable of providing assembly language translations of all 6502/6510 machine code instructions in the standard format laid down by Mostechnology (now part of the Commodore Semiconductor Group), the designers of the chip. To use the Disassembler all that is necessary is to specify, in hexadecimal, the start address of the area of memory to be disassembled. Some care is necessary in choosing the correct start point since a start address which is not also the first byte of a machine code instruction will result in one or more '???' indicators or invalidly translated instructions before the Disassembler synchronises itself with the memory. The occurrence of '???' indicators within the body of a disassembled area of memory indicates the presence of tables of data. Disassembled instructions surrounded by '???' indicators should be treated with some caution, since they may represent random bytes which merely happen to look like genuine machine code instructions. At the end of tables of data, some corruption of instructions may also occur, for the same reason as when an invalid start point is specified. When disassembling past a table, it is wise to attempt to identify the end

of the table by commencing with the address of the last '???' and making several disassemblies, each one starting one byte later in the memory, until a start point is found which generates sensible assembly language from the beginning.

File Editor

The File Editor is simply a means of entering a series of numbered lines — no check is made on entry that these are valid assembly language instructions. Lines may be inserted into the middle of the existing file by giving them the appropriate number. Lines may be listed and deleted in blocks. Single lines may be deleted in input mode by entering a line number without following text. Files which have not been assembled may be saved on tape or disc and later recalled. Files on tape or disc may be merged in with a file currently in memory, provided that the total length of the file does not exceed 255 lines — each line may contain only one assembly language instruction. Lines from a file which is being merged into another currently in memory may be numbered so that they fall into the present body of text, precede it, follow it or overwrite current lines. The 'change device' facility permits the number of the current input/output device to be altered, thus enabling files to be saved to device 4 (ie printed), or backup tapes to be made by those normally working with discs. No check is made that the input/output device currently specified is connected or capable of the saving or loading operation. Machine code data from memory may be added to a file but will appear in the file in byte form, not assembly language.

Assembler

The assembler accepts all the standard assembler mnemonics in standard formats, with the exception that commas are replaced by full-stops. The main commands available for the assembler are as follows:
1) Assemble to memory: the file entered by means of the File Editor is translated into machine code and placed into the memory. Programs may be conflated with previously assembled programs by loading the machine code program into memory (using the File Editor) and then starting assembly of the second program at the byte following the end of the first, thus overcoming any problems you may have with the limitation of a single file to 255 lines. Note that variables and labels from the first program must be redeclared for the second — they are not carried over.
2) Assemble without placing in memory: the file is assembled, with a full listing of all addresses and their contents but memory is unchanged.
3) Error only listing: only those instructions which contain errors will be printed, together with an indication of the nature of the error.
4) Full listing: the full listing of the program is printed including indications of any errors. Note that if there are two errors on the same line,

only one will be indicated on any one assembly. Subsequent assemblies will flag any remaining errors once the first batch have been corrected.

The Assembler provides seven 'directives' which do not appear in the machine code program but modify the manner of assembly:

1) ORG < address> : This directive indicates that the following assembly language instruction is to be assembled at the address specified — subsequent instructions will follow on from that address. A single assembly language program may contain several ORG directives indicating sections of the program which may be placed in entirely different areas of memory.

2) PRT: Following this directive output of the assembled program is diverted from the screen to the printer.

3) SYM: This indicates that the 'symbol table' containing values of variables and addresses at which labelled lines are assembled is to be appended to the listing.

4) END: Whenever encountered, this directive terminates assembly — it does not have to be placed at the end of a program. When END is used as the last line of a program, its address signifies the first free byte of memory which will follow the assembled program.

5) BYT: This directive allows a series of one byte value to be specified in a line, separated by full-stops. The values will be entered directly into memory.

6) DBY: Similar to BYT except that the value specified may be up to two byte range (0-65535). The two bytes will be placed into memory with the high byte first. WRD is the same as DBY except that the two bytes are placed into memory with the low byte first. Note that assembled programs may be saved as machine code files via the Monitor, provided that they have been placed into memory.

APPENDIX C
Mastercode: Table Of Variables

AD	Current address in memory
AM	(Assemble to memory) flag used in assembler
BASE	Current number base for conversions
CO	COntinue in monitor/COmmand in file editor
DEV	Indicates device for load/save
E$	Used in file editor to record empty lines
EA	(End Address) used in monitor
EC	(Error Count) during assembly
EN	(Error Number) used to indicate type of error during assembly
EO	(Error Only listing) flag used in assembler
ERR	Used to flag error conditions
ERR$	Error messages for assembler
EXIT	Set if END directive encountered by assembler
FALSE	Logical value (=0)
FI$	Main file array in file editor
FL	Line to finish list or delete in file editor
FM	Number of lines in FI$
FNDEC	Converts decimal digit to hex ASCII
FNHEX	Converts hex digit to decimal
FP	(Finish Pointer) used by list and delete in file editor
H	Used in conversion routines - H$ converted to decimal
H$	General string for input and output of hex numbers
IN$	General variable used for input
LN	(Line Number) used in file editor
PTR	Pointer used in scanning assembly language instruction
PTR$	Holds order of items in FI$

OP	Operand type: assembler and disassembler
O$	General output string
O1$	Output string used in dump of memory contents to screen
O2$	Output string used in dump of memory contents to screen and disassembler
O3$	Output string used in dump of memory contents to screen
PASS	Current pass of two pass assembler
PO	Pointer to mnemonic type
Q	Loop variable used in assembler
Q1	Start address of line being assembled
Q3	Loop variable used in assembler
Q1$	Temporary variable used in formatting assembler output
RESULT	Output of expression evaluator
SA	(Start Address) used by several routines
SE	Current number of symbols during assembly
SL	(Start Line) used in list and delete in file editor
SM	Maximum number of symbols in the symbol table
SP	Start pointer for list and delete in file editor
ST	System variable in BASIC
ST$	(Symbol Table) used in assembler
SY	Used to indicate dump of symbol table in assembler T,TA,TB,T0,T1,T2 etc. Temporary numeric variable used in several modules
T$	Temporary string variable used in several modules
TA$	Decoder tables for assembler/disassembler
T1$	Temporary variable used in several modules
TERM	Temporary result in expression evaluator
TRUE	Logical value (= -1)
X1	Loop variable used in Hex Loader
XY	Loop variable used in file editor
XZ	Loop variable used in file editor

APPENDIX D
Table of Subroutine Functions in Mastercode Program

10000	General initialisation
10100	Monitor control routine
11000	Convert decimal to hexadecimal
11100	Get byte from memory
11200	Input finish address
11250	Input file name
11850	Ask continue
11950	Convert hexadecimal to decimal
12050	Input start address
12200	Load hex characters into packed string
13000	Get 1 byte from user
13100	Memory modify
13300	Dump memory to screen
13500	Machine code execute
14100	Machine code save
14300	Machine code load
15300	Format operand
15450	Format operand for accumulator addressing mode
15500	Format operand for implied addressing mode
15550	Format operand for immediate addressing mode
15600	Format operand for relative addressing mode
15700	Disassemble instruction
15800	Disassemble memory area
19000	Initialise decoder tables
20000	Assembler control routine
23020	Find line number in file
23100	Add line to file
23300	Delete line from file
23400	List line from file
23500	Get start and finish pointers
23600	Load file from device
23700	Save file to device
23900	Remove leading spaces

24000 Get line number from line input
24200 Input first and last lines
24300 Initialise file
24400 List lines
24500 Delete lines
24600 Input lines
24700 Renumber file
24800 File Editor control routine
25000 Add to file from memory
25500 Change device number
26000 Scan for symbol up to colon
26100 Determine operand type used
26300 Evaluate opcode
26400 Pass 1 control routine
26500 Get length of machine code instruction
26600 Calculate directive length
26900 Dump symbol table
27000 Evaluate operand
27200 Evaluate directive
27400 Evaluate immediate operand
27500 Evaluate relative operand
27600 Pass 2 control routine
28000 Assembler error routine
28100 Print IN$
28150 Scan for symbol up to non-letter/non-digit
28250 Find label in symbol table
28300 Evaluate label or number
28500 Evaluate term
28600 Evaluate expression
28700 Add symbol to symbol table
28850 Test for opcode mnemonics

APPENDIX E
Table Of ROM Routines Called

A38A	Obtain first return address on stack
A474	Print 'READY' and return to direct mode
A4A4	Insert line into BASIC file
A533	Rechain BASIC file
A613	Convert line number held in 14-15 hex to address of line start at 5F-60 hex
A65E	Perform CLR
A7F7	Convert value in floating point accumulator #1 to unsigned integer
A831	Perform END
A8E0	Print 'RETURN WITHOUT GOSUB' and return to direct mode
A8E3	Print 'UNDEFINED STATEMENT' and return to direct mode
AD8A	Get floating point number from BASIC program and place in F-P accumulator #1
AEF1	Evaluate expression within brackets
AEFD	Check next character in BASIC program is a comma, else print 'SYNTAX ERROR'
AEFF	Check next character in BASIC program is the same as that held in accumulator, else print 'SYNTAX ERROR'
B248	Print 'ILLEGAL QUANTITY' and return to direct mode 1
B391	Convert unsigned integer in 14-15 hex to floating point in F-P accumulator #1
B7F7	Convert floating point accumulator #1 to unsigned integer
B828	End of POKE routine
B86A	Add floating point numbers in F-P accumulators #1 and #2. Result in accumulator #1.
BA8C	Copy floating point number indicated by accumulator (low byte) and Y register (high byte) into F-P accumulator #2
E15F	Perform save from memory to device

E175	Perform load or verify from device
E1D4	Get parameters for load and save from BASIC program
FFF0	Kernal routine to put or get cursor position

APPENDIX F
Table of Control Characters

as Represented in the Mastercode Program

[CD]	— Cursor Down
[CU]	— Cursor Up
[CL]	— Cursor Left
[CR]	— Cursor Right
[CLR]	— Screen Clear
[HOME]	— Cursor Home
[GREEN]	— Control 6
[BLUE]	— Control 7

Other titles from Sunshine

THE WORKING SPECTRUM

David Lawrence 0 946408 00 9 £5.95

THE WORKING DRAGON 32

David Lawrence 0 946408 01 7 £5.95

THE WORKING COMMODORE 64

David Lawrence 0 946408 02 5 £5.95

DRAGON 32 GAMES MASTER

Keith Brain/Steven Brain 0 946408 03 03 £5.95

FUNCTIONAL FORTH for the BBC Computer

Boris Allan 0 946408 04 1 £5.95

Sunshine also publishes

POPULAR COMPUTING WEEKLY

The first weekly magazine for home computer users. Each copy contains Top 10 charts of the best-selling software and books and up-to-the-minute details of the latest games. Other features in the magazine include regular hardware and software reviews, programming hints, computer swap, adventure corner and pages of listings for the Spectrum, Dragon, BBC, Vic 20 and 64, ZX 81 and other popular micros. Only 35p a week, a year's subscription costs £19.95 (£9.98 for six months) in the UK and £37.40 (£18.70 for six months) overseas.

DRAGON USER

The monthly magazine for all users of Dragon microcomputers. Each issue contains reviews of software and peripherals, programming advice for beginners and advanced users, program listings, a technical advisory service and all the latest news related to the Dragon. A year's subscription (12 issues) costs £8.00 in the UK and £14.00 overseas.

For further information contact:
Sunshine
12-13 Little Newport Street
London WC2R 3LD

Printed in England by Commercial Colour Press, London E.7.

This extraordinary book opens up a new world for those interested in machine code programming on the Commodore 64.

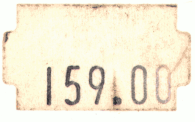
Part 1 provides a full listing and explanation of the Commodore 64 'Master Code Assembler', a sophisticated program with a host of features seldom found even on expensive commercial packages.

Part 2 contains a collection of tested machine code routines which extend the standard Commodore 64 BASIC with more than a dozen new commands. All the routines are fully explained, providing an introduction to a wide range of programming techniques and the ways in which the Commodore 64's ROM can be used to best advantage by the machine code programmer.

The Master Code Assembler will revolutionise your attitude to the Commodore 64 and its capabilities.

David Lawrence is the author of several books on home computing including the best selling The Working Commodore 64. He is also a regular contributor to Popular Computing Weekly.

Mark England is a student of Electronics Engineering at Southampton University.



ISBN 0 946408 05 X

£6.95 net

This was brought to you

from the archives of

http://retro-commodore.eu