COMPUTE!'s

VIC-20 and Commodore 64

Tool Kit: BASIC

Dan Heeb

A step-by-step explanation of the BASIC ROM routines of the VIC-20 and Commodore 64 home computers. Includes information on how to use the routines in your own programs.

A COMPUTE! Books Publication

$16.95

COMPUTE!'s
VIC-20 and Commodore 64
Tool Kit: BASIC

Dan Heeb

COMPUTE!™ Publications,Inc. abc
One of the ABC Publishing Companies
Greensboro, North Carolina

Printed in the United States of America

ISBN 0-942386-32-9

10 9 8 7 6 5 4 3 2 1

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is one of the ABC Publishing Companies, and is not associated with any manufacturer of personal computers. Commodore 64 and VIC-20 are trademarks of Commodore Electronics Limited.

Contents

Preface v

Part One: Overview and Applications 1
Introduction 3
SYS and USR 5
Direct Use of Floating Point Routines 19
Modifying BASIC 41
Mixing BASIC and Machine Language 84

Part Two: Detailed Descriptions 87
BASIC Initialization 89
Entry Phase 96
CHRGET/CHRGOT 99
Tokenization and Program Storage 103
Memory Allocations and Moves 120
Pointer Resets 132
Expression Evaluation 135
Variables and Arrays 156
Floating Point Operations 189
String Operations 256
Statement Execution 279
Statement Execution and Program Flow 287
PRINT 311
LIST 320
DEF and FN 326
Function Invocation 333
INPUT/GET/READ 338
I/O Routines 350
Comparison of Operands 364
PEEK/POKE/WAIT 368
POS 371
RND 372
SYS and USR 375
Check for ASCII A–Z in Accumulator 380
Check If in Direct Mode 381
Get Expression at Next Text Pointer Location 382
Issue IN Line Number Message 383

Syntax Check for Parentheses, Comma, or ASCII Character in Accumulator 384
Error Messages and READY 385
Polynomial Computation 388
Sine, Cosine, and Tangent 393
ATN 399
EXP 401
LOG 403

Appendices 405
A: ASCII Codes 407
B: Screen Codes 411
C: Screen Location Tables (64 and VIC) 413
D: Keycodes (64 and VIC) 414

Index 416

Preface

Before starting on this book, I saw BASIC as just a black box into which you put something and out popped the result. Exactly what went on inside that little box was unknown. Now after finishing the book, not only is it obvious what's going on inside BASIC, but it's also easy to use the same routines that already exist in BASIC in your machine language programs.

In the sense that the BASIC routines are tools, this book tells how to build a tool kit from BASIC. One group of BASIC tools that can be used are the floating point math routines, and an entire section is devoted to how to use these. Other tools from BASIC, such as moving a block of memory, may be found by examining the various detailed descriptions of routines. By picking only the routines that you need from BASIC and avoiding the interpretation of each statement, you can create a machine language program to do the same thing as a BASIC program but much more quickly.

In reading other people's programs I prefer to find a nicely formatted listing that is conducive to easy reading. It seems that many listings of BASIC programs try to cram as many statements as possible into one line without any spaces or remarks. Sure it saves space, but avoiding eyestrain is my preference.

Many programmers, especially those who use languages other than BASIC, prefer the more structured approach to programming; I include myself in this group. Finally, I prefer the way you can trace what is happening in a program to a much finer detail in machine language. With BASIC you have to assume that the definitions of the language are correct, and if they aren't, it's impossible to know whether your program or the language itself is in error.

For an example of how blind faith in a high-level language can lead you astray, enter and run the following short program on a Commodore 64 or VIC-20:

```
10 A = GO("X",5)
```

This statement should produce a syntax error since there is no GO function in Microsoft BASIC. No syntax error is produced, however. Instead, either a warm start of BASIC is done (similar to pressing the STOP and RESTORE keys at the same time) or you hang the system. When you read the section on function invocation, you will see what happens with this statement. This example points out that the rules BASIC supposedly lives by don't always hold true.

Why This Book?

After the above comments about BASIC, you may be wondering what I am doing writing a book about how BASIC works. Well, when I approached BASIC as just a large machine language program, I found it very interesting. Also, I have come to appreciate the immediate feedback from BASIC, and I recognize that BASIC is widely used and very easy to learn.

This book is to be accompanied by a book explaining the inner workings of the Kernal. The Commodore 64 and the VIC-20 are small enough that one person is able to understand all the major aspects of the built-in software. One of the more pleasant moments while doing the research for this book came when seemingly unrelated topics suddenly fell together to make sense and give the *Aha!* feeling. An example of this is when I was looking at function invocation for USR and realized there was no reason why USR could not operate on string functions.

A plethora of introductory books already exist about the VIC-20 and the Commodore 64. Once you get through these introductory books, where do you go? Besides magazine articles, there are few intermediate or advanced texts. I hope this book will find its niche in this uncrowded category.

This book assumes that the reader is already familiar with BASIC and 6502 machine language, so no attempt is made to teach these languages. I don't want to frighten away beginning BASIC or machine language programmers, though, for you too should find useful pointers here. A good method for learning how to write programs is to see how others have written them, and this book examines one such huge program, Microsoft BASIC as implemented on the Commodore 64 and VIC-20. Due to copyright restrictions, the source code for BASIC is not reproduced here. While other versions of Microsoft BASIC exist, this book treats only Microsoft version 2 as implemented on the Commodore 64 and VIC-20.

Following the Introduction are chapters on how to use SYS and USR; how to directly use floating point routines;

how to modify BASIC; and where to place machine language programs. The remainder of the book gives detailed descriptions of every BASIC routine. These descriptions of routines have been grouped into common topics.

This book has been written to apply to both the Commodore 64 and the VIC-20. When an address differs, the alternate address for each computer is indicated by a slash between two numbers, with the number on the left giving the Commodore 64 address, and on the right, the VIC-20. If only the leftmost digit differs, then just that one digit appears to the left of the slash (/). For example, E034/E023 refers to location E034 (hex) on the Commodore 64 and E023 on the VIC-20, while B/D02B refers to B02B on the 64 and D02B on the VIC. Also, when the book uses a hex number that represents an address, the number is written in this manner: B02B. But when the number refers to an absolute hex number used by the accumulator or one of the registers, it is preceded by a dollar sign: $25. Decimal numbers are indicated by the word *decimal* following in parentheses. The meaning of the few exceptions to these rules should be obvious from the context.

Another convention I use concerns the = sign. When I use = in explaining a routine, I imply that the values on the right side of the = are assigned to the variable on the left side of the =.

This book can best be used in conjunction with *Mapping the VIC* by G. Russ Davies or *Mapping the Commodore 64* by Sheldon Leemon. Since I generally just hop right into the details of the topic, consult the memory map to get a general summary of a routine, and then come back here for the details. The cross reference of routines in sequential address order to page numbers in the index should be useful in this respect.

Also, you will probably want to compare the descriptions of routines to the actual instructions comprising the routine. I don't include the disassembled instructions in the book, but I do provide the starting and ending addresses of every routine. You could use a machine language monitor such as Micromon64, Micromon, HESMON, VICMON, or any other monitor that has a disassembler capability to actually display the instructions. A couple of books, *Inside the Commodore 64,* by Milton Bathurst, and *The Anatomy of the Commodore 64,* from

Abacus Software, actually include the disassembled instructions for the Commodore 64. Although these disassembled listings are helpful, the brief comments in these two books aren't really adequate to gain an understanding of what the routines do.

I would like to thank G. Russ Davies for his help during this project. Russ provided much food for thought. His *Mapping the VIC* was a constant reference. I can vouch for its technical accuracy and high quality.

The editors at COMPUTE! were quite helpful and almost saw the meaning of infinity when considering the completion of this book. I thank them for giving me the time to do a thorough research into BASIC.

I would like to dedicate this book to Mom and in memory of Dad.

Part One

Overview and Applications

Introduction

BASIC is an interpreted language in which each BASIC statement is executed in turn. The general idea of how BASIC works is very simple. BASIC goes through a cycle. The main traffic cop of BASIC that directs the flow of action is termed the Main BASIC Loop in this book. This loop gains control after BASIC is warm or cold started, whenever an error occurs, or whenever a program or direct mode statement finishes execution.

The Main BASIC Loop waits for input from the current input device, typically the keyboard, and when a carriage return is received it reads the logical line from screen memory into a buffer area.

Once the line is in the buffer, the Main BASIC Loop determines if the logical line begins with a line number. Next the loop calls the tokenization routine to convert any recognizable BASIC keywords into their equivalent tokens and to crunch the buffer by removing the keywords.

Once tokenization is completed, one of two things will happen. If the line was entered without a line number, direct mode is in effect and the statement is executed. If the logical line contained a line number, the tokenized buffer is stored, along with the line number, in an area reserved for the tokenized program text.

After storing a line in the program text area, all lines in the text area are relinked, with each line pointing to the following line. To execute the program, a direct mode statement, the RUN command, is entered. RUN changes the pointer that points to where to get the next statement for execution. This pointer is changed from pointing to the input buffer to the start of the BASIC tokenized program area. Each statement in the program is then interpreted and the command in the statement executed. After a statement is executed, a routine determines whether any more statements need to be executed. If not, then the Main BASIC Loop is again reentered to await the next logical line from the input device.

BASIC executes only one statement at a time, and each time a statement is executed, it is checked for correct syntax. Any syntax errors cause control to return to the Main BASIC

Loop. This interpretation of each BASIC statement is one of the reasons why BASIC programs are so much slower than machine language programs.

Whenever a new program line is entered or edited, the variable area is cleared. Execution of each BASIC statement in the program text area then can create and reference variables.

There are very few differences between the BASIC on the VIC-20 and the BASIC on the Commodore 64. These few differences include the following: the warm start/cold start routines differ slightly; the method of handling comma to tab to screen locations during PRINT statements is handled differently because of the different screen sizes on the machines; certain patch areas in VIC-20 BASIC such as one that exists in the LOAD routine have been reintegrated into the main-line BASIC routine; and the RND function uses different timers.

Other than these differences, BASIC is essentially the same. The area where BASIC is located differs on the two computers. On the VIC-20, BASIC is located from C000 to E47A, while on the Commodore 64, BASIC is located from A000 to BFFF and from E000 to E4AC. The locations of routines in the first 8K area match exactly. Because a JMP is needed by the 64 to bridge the gap from BFFF to E000, most instructions in the remaining E000 area are offset by three bytes.

SYS and USR

The SYS and USR commands are discussed later in this book. This section gives some examples of how to use SYS and USR to access machine language programs. As these examples show, you can do virtually anything in a machine language routine as long as you return control to BASIC when the routine finishes.

SYS is more appropriate to use in some situations, and USR is more appropriate in others. SYS is a command, so it is executed as a statement, while USR is a function that must be on the right side of the equal sign in an expression.

SYS is well-suited for use when you wish to pass values to your machine language routine in the accumulator, X-register, Y-register, and status register, and then have the accumulator and these registers pass information back to BASIC.

USR is well-suited for use when you wish to pass information to the machine language routine in Floating Point Accumulator 1 (FAC1) and have the results returned in FAC1. The information that is passed in Floating Point Accumulator 1 is the value resulting from evaluating the expression within parentheses of the USR argument.

Below are three examples each for SYS and USR. The first example in each case is a relatively straightforward and common use of SYS and USR. The other examples are additional techniques that allow you to pass additional arguments to SYS and USR.

Each of these examples contains the addresses for Commodore 64 routines. It is easy to convert these addresses to the corresponding VIC-20 addresses—any address that starts with a B for the 64 starts with D on the VIC, and any address that starts with an A on the 64 starts with C on the VIC. For example, BBFC on the 64 = DBFC on the VIC, and AEFD on the 64 = CEFD on the VIC.

Locating your machine language program is not difficult on the Commodore 64. Most of the examples here locate the routines in the 4K block beginning at C000. On the VIC the best place to locate machine language is above or below BASIC programming RAM. For more information about how to do this, see the section called "Mixing BASIC and Machine Language" in Part 1.

SYS and USR Examples

The first example for SYS demonstrates passing values to your machine language program using the locations reserved for this purpose by BASIC. These locations at (decimal) 780, 781, and 782 are for passing values to the accumulator, X-register, and Y-register. Location 783 can also be used to pass a setting for the status register. Upon return from your machine language program, SYS loads these values back from the corresponding registers. This example asks for three numbers and POKEs these numbers into the locations that are passed to the registers. (In this example it really isn't important which values are POKEd to which locations.) Then the machine language program sorts the three values, returning the highest value in the accumulator, the next highest in the X-register, and the smallest in the Y-register. Upon return, SYS reloads 780, 781, and 782, and then PRINT displays the sorted numbers.

Example 1

BASIC

```
10 INPUT A
20 INPUT B
30 INPUT C
40 POKE 780,A
50 POKE 781,B
60 POKE 782,C
70 SYS 49152
80 PRINT"NUMBERS FROM LARGEST TO SMALLEST ARE:"
90 PRINTPEEK(780);PEEK(781);PEEK(782)
```

Assembly Listing

```
.OPT NOERRORS,LIST,GENERATE
;SYSUSR EXAMPLE #1
       * = $C000
       STA FIRST   ;STORE ENTRY VALUES OF REGISTERS
```

```
      STX SECOND
      STY THIRD
      LDY #$Ø2   ;AT MOST TWO PASSES ARE NEEDED
;                  FOR SORT
LOOP   LDA FIRST
      CMP SECOND ;COMPARE FIRST TO SECOND
      BCS NOSW1
      LDX SECOND
      STX FIRST  ;SWAP IF SECOND IS GREATER
      STA SECOND
NOSW1  LDA SECOND
      CMP THIRD  ;COMPARE SECOND TO THIRD
      BCS NOSW2
      LDX THIRD
      STX SECOND ;SWAP IF THIRD IS GREATER
      STA THIRD
NOSW2  DEY
      BNE LOOP
      LDA FIRST  ;RESTORE REGS FOR RETURN TO SYS
      LDX SECOND
      LDY THIRD
      RTS
FIRST  .BYTE Ø
SECOND .BYTE Ø
THIRD  .BYTE Ø
      .END
```

Example 2 for SYS shows that your machine language program does not necessarily have to return to the statement following the SYS, as might be expected, since SYS resembles a GOSUB. This example uses the previous example to determine the largest of three numbers. Then the machine language program adds the largest number to the current line number, sets the (14) address for GOTO, and calls the GOTO routine to return to BASIC at this new line number.

Example 2

BASIC

```
1Ø A = 3
2Ø B = 7
3Ø C = 8
35 PRINT"A = " A;"B = " B;"C = " C
4Ø POKE 78Ø,A
5Ø POKE 781,B
55 POKE 782,C
6Ø SYS 49152
```

```
61 PRINT"61"
62 PRINT"62"
63 PRINT"63"
64 PRINT"64"
65 PRINT"65"
66 PRINT"66"
67 PRINT"67"
68 PRINT"68"
69 PRINT"69"
```

Assembly Listing

```
.OPT NOERRORS,LIST,GENERATE
;SYSUSR EXAMPLE #2
      * = $CØØØ
      STA FIRST  ;STORE ENTRY VALUES OF REGS
      STX SECOND
      STY THIRD
      LDY #$Ø2   ;AT MOST TWO PASSES ARE
;                  NEEDED FOR SORT
LOOP   LDA FIRST
      CMP SECOND ;COMPARE FIRST TO SECOND
      BCS NOSW1
      LDX SECOND
      STX FIRST  ;SWAP IF SECOND IS GREATER
      STA SECOND
NOSW1  LDA SECOND
      CMP THIRD  ;COMPARE SECOND TO THIRD
      BCS NOSW2
      LDX THIRD
      STX SECOND ;SWAP IF THIRD IS GREATER
      STA THIRD
NOSW2  DEY
      BNE LOOP
      LDA FIRST  ;GET LARGEST NUMBER
      CLC
      ADC $39    ;ADD TO CURRENT LINE NUMBER
      STA $14    ;STORE IN LINE NUMBER
;                  FOR GOTO
      LDA $4Ø
      ADC #$ØØ
      STA $15
      JSR $A8A3  ;CALL GOTO
      RTS
FIRST  .BYTE Ø
SECOND .BYTE Ø
THIRD  .BYTE Ø
      .END
```

Example 3 demonstrates passing parameters to the SYS machine language program. Articles by J.C. Johnson in *COMPUTE!* magazine in November and December of 1982 showed how to do this for PETs and were the inspiration for this example. These parameters can be used to pass values to the program or receive values back from the program. The order in which the parameters are passed is unimportant as long as you pass them in the order your machine language program expects.

This example is rather frivolous. It just performs addition or subtraction on two expressions and returns a value in a variable. The first extra parameter to SYS specifies the type of operation, the next two arguments are the expressions, and the final argument is the variable. This example was used so we could easily verify that SYS is working as expected by comparing it to the BASIC addition and subtraction results.

Whenever control is passed to SYS, the text pointer (7A) is left pointing to the first nonspace character following the first argument to SYS. Normally this first nonblank is the end-of-line $00 byte or the end-of-statement colon. However, in this example, (7A) points to the following comma.

The first thing the machine language program does is to syntax-check for a comma by doing a JSR AEFD. This syntax check displays SYNTAX ERROR if it doesn't find a comma. If it finds a comma, CHRGET is called to retrieve the next nonblank character and return it to the accumulator, leaving (7A) pointing to this character.

The routine then calls CHRGOT to reload the accumulator with the character pointed to by (7A) and saves this character, which specifies whether addition or subtraction is requested. Then again the syntax check for a comma is done. Next, the expression evaluation routine at AEFD is called to evaluate the left operand expression and return the result in FAC1. Then the result returned is checked to see if it was a string expression, and if so the TYPE MISMATCH error is displayed. If it was a numeric expression, FAC1 is copied to 0057–005B because FAC1 will be overwritten during the next expression evaluation.

By using this location at 0057–005B, the erroneous addition for LOG is obtained since LOG uses this same location as a work area. After handling the left operand, the same is done for the right operand, leaving the result in FAC1. Then it checks to see if the token for + or − was specified by reloading the earlier saved value. The check is for a token and not the ASCII character value since tokenization has converted − or + to its equivalent tokens. If neither of these is found, SYNTAX ERROR is displayed.

For addition, JSR B867 to add the variable at 0057 to FAC1 and return the result in FAC1. For subtraction, copy 0057 to Floating Point Accumulator 2 (FAC2) and then JSR B853 to subtract FAC2 from FAC1 and leave the result in FAC1. Now copy FAC1 to FAC2 because we're going to overlay FAC1 in the next step. Check for another comma, and then JSR B08B to locate or create the variable that follows this final comma. Upon return from B08B, the X-register and Y-register point to the data area in the variable descriptor. Store these registers in (47) and then JSR BBD4 to copy FAC1 to the data area pointed to by (47).

Finally, call A8F8 to reset the text pointer to the final end-of-line and end-of-statement byte, thus ignoring any additional arguments that might be on the SYS line.

Example 3

BASIC

```
2Ø A = 3*5
3Ø B = 2Ø-7
4Ø C = A*3 + B-2
5Ø PRINT"NORMAL + =" C
6Ø SYS 49152,+,A*3,B-2,X
7Ø PRINT"SYS PLUS = " X
8Ø D = A*2 - 5
9Ø PRINT"NORMAL - = " D
1ØØ SYS 49152,-,A*2,5,Z
11Ø PRINT"SYS MINUS = " Z
12Ø REM FOLLOWING STATEMENTS SHOW DIFFERENT RESULT
    S FOR + AND SYS PLUS
125 REM SYS ROUTINE USES THE SAME WORK AREA AS LOG
     THUS CAUSING THIS DIFFERENCE
13Ø Z = 2Ø + LOG(5)
14Ø PRINT"NORMAL + = " Z
15Ø SYS 49152,+,2Ø,LOG(5),T
16Ø PRINT"SYS PLUS = " T
22Ø REM FOLLOWING STATEMENTS SHOULD CAUSE SYNTAX E
    RRORS
23Ø SYS 49152,*,3,7,X
```

```
240 SYS 49152,+,3
250 REM FOLLOWING STATEMENTS SHOULD CAUSE TYPE MIS
    MATCH ERROR
260 SYS 49152,+,"X",1,Z
270 SYS 49152,-,1,"X",T
```

Assembly Listing

```
; SYSUSR EXAMPLE #3
          *=$C000
          JSR $AEFD   ;SYNTAX CHECK FOR COMMA,
;                      LEAVE (7A) AT NEXT
;                      NON-BLANK
          JSR $0079   ;CHRGOT TO RETRIEVE
;                      THE OPERATOR TOKEN
          STA $030C   ;SAVE THE TOKEN
          JSR $0073   ;GET NEXT NON-BLANK
          JSR $AEFD   ;SHOULD BE A COMMA
          JSR $AD9E   ;EVALUATE FIRST
;                      EXPRESSION AND
;                      LEAVE IN FAC1
          LDA $0D
          BEQ NOERR1
          LDX #$16    ;IF A STRING EXPRESSION
          JMP $A437   ;DISPLAY TYPE MISMATCH
NOERR1     JSR $BBCA   ;COPY FAC1 TO 0057
          JSR $AEFD   ;CHECK FOR COMMA
;                      AFTER FIRST EXPRESSION
          JSR $AD9E   ;EVALUATE SECOND
;                      EXPRESSION AND
;                      LEAVE IN FAC1
          LDA $0D
          BEQ NOERR2
          LDX #$16    ;IF A STRING EXPRESSION
          JMP $A437   ;DISPLAY TYPE MISMATCH
NOERR2     LDA $030C   ;RETRIEVE THE OPERATOR
          CMP #$AA    ;TOKEN FOR "+"?
          BEQ PLUS
          CMP #$AB    ;TOKEN FOR "-"?
          BEQ MINUS
          JMP $AF08   ;IF NEITHER THEN
;                      SYNTAX ERROR
PLUS       LDA #$57
          LDY #$00
          JSR $B867   ;FLOATING POINT ADDITION:
;                      FAC1 = FAC1 + FVAR
          CLC
          BCC BYSUB   ;BRANCH AROUND SUBTRACTION
```

```
MINUS     LDA #$57
          LDY #$ØØ
          JSR $BA8C  ;COPY ØØ57 TO FAC2
          JSR $B853  ;FLOATING POINT SUBTRACTION:
;                      FAC1 = FAC1 - FAC2
BYSUB     JSR $BCØC  ;COPY FAC1 TO FAC2
          JSR $AEFD  ;CHECK FOR COMMA
;                      AFTER SECOND EXPRESSION
          JSR $BØ8B  ;LOCATE OR CREATE
;                      THE FOLLOWING VARIABLE
          JSR $BBFC  ;COPY FAC2 TO FAC1
          LDX $47
          LDY $48
          JSR $BBD4  ;COPY FAC1 TO DATA
;                      AREA OF VARIABLE
          JSR $A8F8  ;MOVE TEXT POINTER
;                      TO END-OF-LINE
;                      OR END-OF-STATEMENT
          RTS
```

The first example for USR passes a value to the USR machine language program in FAC1. The machine language program compares the value passed to the square root of two, and returns −1 if the value passed in FAC1 is larger or returns 0 if the value passed in FAC1 was smaller than or equal to the square root of two.

Example 4

BASIC

```
1Ø A = 49152
2Ø B = A/256
3Ø C = A - B*256
4Ø POKE 784,76
5Ø POKE 785,C
6Ø POKE 786,B
65 X = .3
7Ø PRINT"X =" X
75 Z = USR(X)
8Ø PRINTZ
```

Assembly Listing

```
; SYSUSR EXAMPLE #4
      BIT $ØD
      BPL NUMARG
      LDX #$16   ;IF STRING ARGUMENT
;                  DISPLAY TYPE MISMATCH
```

```
NUMARG LDA #$Ø4
       STA $12    ;SET COMPARISON FOR SQR(2)
;                   LESS THAN FAC1
       LDA #$DB
       LDY #$B9   ;POINT TO SQR(2)
       JSR $BA8C  ;COPY SQR(2) TO FAC2
      JSR $BØ1B   ;DO NUMERIC COMPARISON
      RTS
```

The next example for USR shows that you don't have to have a JMP opcode at location 00 (VIC) or 0310 (64). Since a JMP 0000 or JMP 0310 is made when a USR is encountered, any machine language instructions at that location will be executed. On the VIC you have room for seven bytes at 0000 before you start messing up the page 0 variables, while on the 64 you have only four bytes. Thus the example differs for the VIC and the 64. On the VIC, two JSR instructions and an RTS are located starting at 0000, while on the 64 a JMP to the machine language program at 033C in the tape buffer area is done, and there the two JSRs are performed. The call to B/D7F7 converts FAC1 to a two-byte integer in (14). The call to A/C8A3 then does a GOTO this address in (14).

Example 5

Commodore 64

```
2 REM ASSEMBLY LANGUAGE CODE POKED IS:
3 REM Ø31Ø JMP $Ø33C
4 REM Ø33C JSR $B7F7
5 REM Ø33F JSR $A8A3
6 REM Ø342 RTS
1Ø POKE 784,76
2Ø POKE 785,6Ø
3Ø POKE 786,3
4Ø POKE 828,32
5Ø POKE 829,247
6Ø POKE 83Ø,183
7Ø POKE 831,32
8Ø POKE 832,163
9Ø POKE 833,168
95 POKE 834,96
97 X = 1Ø5
98 PRINT"X=" X
1ØØ Y = USR(X)
1Ø1 PRINT"1Ø1"
1Ø2 PRINT"1Ø2"
```

```
1Ø3 PRINT"1Ø3"
1Ø4 PRINT"1Ø4"
1Ø5 PRINT"1Ø5"
1Ø6 PRINT"1Ø6"
1Ø7 PRINT"1Ø7"
1Ø8 PRINT"1Ø8"
1Ø9 PRINT"1Ø9"
```

VIC-20

```
2 REM ASSEMBLY LANGUAGE FOR VIC 2Ø
3 REM ØØØØ JSR $D7F7
4 REM ØØØ3 JSR $C8A3
5 REM ØØØ6 RTS
1Ø POKE Ø,32
2Ø POKE 1,247
3Ø POKE 2,215
4Ø POKE 3,32
5Ø POKE 4,163
6Ø POKE 5,2ØØ
7Ø POKE 6,96
8Ø X = 1Ø7
85 PRINT"X =" X
9Ø Y = USR(X)
1ØØ PRINT"1ØØ"
1Ø1 PRINT"1Ø1"
1Ø2 PRINT"1Ø2"
1Ø3 PRINT"1Ø3"
1Ø4 PRINT"1Ø4"
1Ø5 PRINT"1Ø5"
1Ø6 PRINT"1Ø6"
1Ø7 PRINT"1Ø7"
1Ø8 PRINT"1Ø8"
1Ø9 PRINT"1Ø9"
```

The final example shows that, contrary to most of the available literature about USR on the Commodore BASIC version, you can indeed handle a string expression in USR. This example takes three arguments such as X$ = USR(A$),B$,X.

The USR machine language program inserts string B$ into string A$ starting at position X in string A$. The resulting string is returned from the USR function and assigned to X$. Because USR evaluates the expression within parentheses before passing control, you have no chance to insert additional arguments inside the parentheses. Thus additional arguments follow the parentheses and the same general technique discussed in the SYS example before is used to scan for commas and evaluate expressions.

When your machine language program receives control, (64) points to the string descriptor for the string within parentheses. This descriptor is saved at (6F). The next string expression returns a string descriptor that is saved at (FD), and also a type check is done to display TYPE MISMATCH if this is a numeric expression. Then B79E is called to evaluate the next expression. B79E displays ILLEGAL QUANTITY if the expression does not evaluate to 0–255. The 0–255 value is returned in the X-register.

The program reserves room in the active string area to hold the combined lengths of the two strings, then copies the first string up to the starting insertion location to this reserved area. Next the second string to be inserted is copied continuing from where the previous copy stopped. Finally, the remainder of the first string is copied to the tail end. At this point two very critical PLAs are performed to remove the return address to function invocation. If this address is not removed, a TYPE MISMATCH error will occur. Finally, JMP B4CA to create a string descriptor for this new string on the temporary string dispatcher stack. Leave a pointer to the new string in (64), which is part of FAC1 that is then passed back to be assigned to the string variable on the left side of the equal sign.

Example 6

BASIC

```
5 A = 49152
6 B = A/256
7 C = A - B*256
8 POKE 784,76
9 POKE 785,C
10 POKE 786,B
20 A$ = "HAT"
30 B$ = "UTAH"
40 C$ = USR(A$),B$,4
50 PRINTC$
60 D$ = USR(C$),",",4
70 PRINTD$
80 E$ = "YOU HERE"
90 F$ = "'ALL"
100 G$ = "IN ARK"
```

```
11Ø H$ = "ANSAS"
12Ø I$ = USR(E$),F$+G$+H$,4
13Ø PRINTI$
132 XX$ = "NOTCOMPLEX"
133 YY$ = USR(XX$)," TOO ",4
134 PRINTYY$
```

Assembly Listing

```
; SYSUSR EXAMPLE #6 COMMODORE 64
        *=$CØØØ
       LDA $65   ;SAVE STRING DESCRIPTOR
;                  OF STRING
       PHA       ;EXPRESSION WITHIN
;                  PARENTHESES
       LDA $64
       PHA
       LDA $64
       LDY $65
       JSR $B6DB ;CLEAN-UP TEMPORARY
;                  STRING DESCRIPTOR STACK
       JSR $AEFD ;CHECK FOR FOLLOWING
;                  COMMA
       JSR $AD9E ;EVALUATE NEXT EXPRESSION
;                   - THE SUBSTRING
      LDA $ØD
      BNE NOERR1
      LDX #$16
      JMP $A437  ;DISPLAY TYPE MISMATCH
;                  IF NUMERIC EXPRESSION
NOERR1 LDA $64
      STA $FD    ;SAVE SUBSTRING
;                  DESCRIPTOR IN (FD)
      LDA $65
      STA $FE
      LDA $64
      LDY $65
      JSR $B6DB  ;CLEAN-UP TEMPORARY
;                  STRING DESCRIPTOR STACK
      JSR $AEFD  ;CHECK FOR COMMA
;                  FOLLOWING SUBSTRING
      JSR $B79E  ;EVALUATE EXPRESSION
;                  TO VALUE FROM Ø-255
;                  INTO X-REG
      DEX        ;SET INSERTION POSITION
;                  OFFSET FROM Ø
      STX $FF
      PLA        ;RESTORE FIRST STRING
;                  DESCRIPTOR INTO (6F)
```

```
       STA $6F
       PLA
       STA $7Ø
       LDY #$ØØ
       LDA ($6F),Y ;GET LENGTH OF FIRST
;                    STRING
       CLC
       ADC ($FD),Y ;ADD LENGTH OF
;                    SUBSTRING
       BCC NOERR2
       LDX #$17     ;IF COMBINED LENGTH
;                    GREATER THAN 255
       JMP $A437    ;DISPLAY STRING TOO
;                    LONG ERROR
NOERR2 STA $FC       ;SAVE COMBINED LENGTH
       JSR $B475    ;ALLOCATE AREA FOR
;                    COMBINED STRING IN
;                    ACTIVE STRING
;                    LEAVE LENGTH IN 61
;                    AND POINTER IN (62)
       LDY #$Ø1
       LDA ($6F),Y
       STA $22      ;PUT POINTER TO FIRST
;                    STRING IN (22)
       INY
       LDA ($6F),Y
       STA $23
       LDA $FF      ;RESTORE THE INSERTION
;                    POSITION TO STOP
;                    THIS COPY
       JSR $B68C    ;COPY STRING TO AREA
;                    ALLOCATED
       LDY #$ØØ
       LDA ($FD),Y ;GET LENGTH OF
;                    SUBSTRING
       PHA
       INY
       LDA ($FD),Y
       STA $22      ;PUT POINTER TO SECOND
;                    STRING IN (22)
       INY
       LDA ($FD),Y
       STA $23
       PLA          ;RESTORE LENGTH OF
;                    SUBSTRING FOR COPY
       JSR $B68C    ;COPY SUBSTRING
;                    TO INSERTION POSITION
       LDY #$ØØ
       LDA ($6F),Y ;DETERMINE AMOUNT
```

```
;                         REMAINING TO COPY
;                         FROM FIRST STRING
          SEC
          SBC $FF
          PHA
          INY
          LDA ($6F),Y
          CLC
          ADC $FF
          STA $22       ;SET POINTER FROM
;                         WHERE TO RESUME
;                         COPY OF FIRST STRING
          INY
          LDA ($6F),Y
          ADC #$ØØ
          STA $23
          PLA
          JSR $B68C     ;COPY TAIL END OF
;                         FIRST STRING
          PLA           ;REMOVE RETURN
;                         ADDRESS TO
;                         FUNCTION INVOCATION
          PLA           ;MUST BE DONE TO
;                         PREVENT TYPE
;                         MISMATCH
          JMP $B4CA     ;CREATE DESCRIPTOR
;                         FOR NEW STRING IN
;                         IN THE TEMPORARY
;                         STRING DESCRIPTOR
;                         STACK AND ALSO
;                         POINT (64)  TO
;                         THIS DESCRIPTOR
          BRK           ;USE THE RTS FROM
;                         LAST JMP
```

Direct Use of Floating Point Routines

Writing routines to do complicated math calculations such as division, exponentiation, square root, and multiplication can be very challenging. However, there is no need to write these machine language routines. It's best to use what's already waiting for you on the Commodore 64 and the VIC-20, namely the built-in floating point routines that are used by BASIC. There are many reasons for using these math routines from machine language rather than BASIC. One is that your machine language program will execute much faster than the corresponding BASIC program.

To use the floating point routines, you need to know where they are located, what conditions they expect to receive, and what they return. Myriad details about the floating point routines are found later in this book. For the answer to any particular question you may have about what a particular routine requires, try those sections.

Certain floating point routines, but not all, make use of the BASIC CHRGET routine at 0073–008A. Since that routine is used by some floating point routines, it is safe to go ahead and create it in BASIC. To do this, merely call the routine which copies CHRGET to page 0. On the VIC-20, do a JSR E3A4; on the Commodore 64 do a JSR E3BF. This E3BF/E3A4 routine also assigns values to the following page 0 locations: 00, 01, 02, 03, 04, 05, 06, 13, 16, 18, 2B, 2C, 33, 34, 37, 38, 53, 54, 68, 8B, 8C, 8D, 8E, 8F. Don't use these locations if your machine language program resides in page 0. Also, the floating point routines make use of two floating point accumulators, Floating Point Accumulator 1 (FAC1) at 61–66 and Floating Point Accumulator 2 (FAC2) at 69–6E. Work areas also exist at 57–60.

It is safest to keep your machine language program out of page 0 entirely, but if you do use page 0, at least keep away from locations 57–8F and the other locations listed above. FAC1 is the crucial floating point accumulator since all of the routines use it to store their final result. FAC2 is used to hold arguments that are used in a floating point routine. For example, you can add a floating point variable to FAC1. When this addition occurs, what really happens is that the floating point variable is copied to FAC2 and then FAC2 is added to FAC1 and the result is left in FAC1.

The following list gives the most frequently used floating point routines. Certain routines such as normalization are done for you, so it would be rare for you to have to use that routine directly. Also, the various trig and math functions such as LOG and EXP are not listed here, but they could be used in the same manner as the floating point routines described as long as you set up the correct calling environment. In this list, FVAR refers to a floating point variable in memory and the addresses are listed in the Commodore 64/VIC-20 format discussed above.

The required arguments are given where needed, although certain routines have unusual requirements. For example, multiplying FAC1 by FAC2 requires that the accumulator contain the exponent of FAC1 at entry. Generally, the examples that follow demonstrate any special requirements.

Floating Point Copies—Variable to Accumulator, Accumulator to Variable, and Accumulator to Accumulator

B/DBA2 Copy FVAR to FAC1; accumulator = low order, Y-register = high order address of FVAR.
B/DA8C Copy FVAR to FAC2; accumulator = low order, Y-register = high order address of FVAR.
B/DBD0 Copy FAC1 to (49); (49) typically points to the data area for a variable descriptor when called from BASIC.
B/DBD4 Copy FAC1 to address specified by X-register (low order) and Y-register (high order).
B/DBC7 Copy FAC1 to 5C–60.
B/DBCA Copy FAC1 to 57–5B.
B/DBFC Copy FAC2 to FAC1.
B/DC0C Copy FAC1 to FAC2.

Floating Point Addition and Subtraction

B/D86A FAC1 = FAC2 + FAC1.
B/DD7E FAC1 = FAC1 + accumulator (the 6502 accumulator with a value that you should set from $00–$09).
B/D867 FAC1 = FVAR + FAC1; accumulator = low order, Y-register = high order address of FVAR.
B/D853 FAC1 = FAC2 − FAC1.
B/D850 FAC1 = FVAR − FAC1; accumulator = low order, Y-register = high order address of FVAR.
B/D849 FAC1 = FAC1 + 0.5

Floating Point Multiplication and Division

B/DA2B FAC1 = FAC1 * FAC2.
B/DA28 FAC1 = FAC1 * FVAR; accumulator = low order, Y-register = high order address of FVAR.
B/DAE2 FAC1 = FAC1 * 10.
B/DB12 FAC1 = FAC2 / FAC1.
B/DB0F FAC1 = FVAR / FAC1; accumulator = low order, Y-register = high order address of FVAR.
B/DAFE FAC1 = FAC1 / 10.

Other Math and Logic Routines

B/DC1B Round FAC1.
B/DF7B Exponentiation: FAC1 = FAC1 raised to the power FAC2.
B/DF71 SQR: FAC1 = SQR(FAC1).
A/CFE9 AND: FAC1 = FAC1 AND FAC2.
A/CFE6 OR: FAC1 = FAC1 OR FAC2.
B/DCCC INT: FAC1 = INT(FAC1).
B/D947 Two's Comp: FAC1 = Two's Comp of FAC1.
B/DC39 SGN: FAC1 = SGN(FAC1).
B/DC58 ABS: FAC1 = ABS(FAC1).
B/DC5B Comparison: FAC1 = FAC1 comparison to FVAR; accumulator = low order, Y-register = high order address of FVAR.

Integer to Floating Point Conversions

B/DC44 Convert two-byte integer in 62 and 63 (of FAC1) to floating point in FAC1.

B/DC3C Convert accumulator to floating point in FAC1.
B/DDCD Convert accumulator (low order) and X-register (high order) to floating point and then convert the floating point to an ASCII string and send this ASCII string to the current output device.
B/D3A2 Convert Y-register to floating point in FAC1.

ASCII Decimal Number to Floating Point

B/DCF3 Convert ASCII decimal number (such as −23E+07) to floating point in FAC1.

Floating Point to ASCII

B/DDDD Convert FAC1 floating point to ASCII string at 0100 with accumulator (low order) and Y-register (high order) pointing to 0100 at exit.

Floating Point to Fixed Point Integer

B/DC9B Convert floating point FAC1 to fixed point four-byte integer in FAC1.
B/D1BF Convert floating point FAC1 to fixed point integer in the range −32,768 to 32,767 and store in 64 and 65.
B/D1D2 Convert numeric expression from floating point to fixed point integer in 64 and 65.
B/D1AA Convert floating point FAC1 to two-byte integer with low order in Y-register and high order in accumulator.
B/D7A1 Convert floating point FAC1 to integer in the range 0–255 and leave integer in X-register.
B/D79E Evaluate expression and convert FAC1 floating point expression value to integer in the 0–255 range left in X-register.

Helpful I/O Routines

A/CB24 PRINT a string pointed to by (22) and with the length in the accumulator.
A/CBF9 INPUT characters from the current input device and fill the buffer at 0200.

Now come the actual examples of using floating point routines. The actual display of the examples in execution was made by printing the Commodore 64 screen using the

HESMON 64 monitor. The G C300 is the HESMON command to execute a program. These examples all use the Commodore 64 addresses; consult the list above if you need to convert to the VIC. These routines could be packaged into a set of subroutines to handle most of your math needs from machine language.

Floating Point Examples

Example 1 converts the integer values in the accumulator and X-register to floating point, then to an ASCII string that is displayed on the screen. However, since only a two-byte integer can be displayed by using this routine, Example 2 will show a way to display the entire FAC1 value up to its maximum.

Guide to Labels Used in Floating Point Examples

```
.OPT NOERRORS,LIST,GENERATE
; FLOATING POINT EXAMPLES COMMODORE 64
;
; EQUATES FOR ROUTINE ADDRESSES
;
; CONVERT ACCUM AND X-REG TO
;  ASCII STRING
; AND SEND TO CURRENT OUTPUT DEVICE
 AXOUT   = $BDCD
; INITIALIZE CHRGET
 INTCGT = $E3BF
; PROMPT FOR ? FILL BUFFER BY READING
;   INPUT DEVICE
; - X,Y POINT TO Ø1FF
 PROMPT = $ABF9
 CHRGET = $ØØ73
 CHRGOT = $ØØ79
; CONVERT ASCII STRING AT Ø2ØØ TO FAC1
;    FLOATING POINT
ASCFLT = $BCF3
; CONVERT FAC1 FLOATING POINT TO ASCII
;    STRING AT Ø1ØØ
FACASC = $BDDD
; COPY FLOATING POINT VARIABLE POINTED
;    TO BY A AND Y  TO FAC1
   FVTF1 = $BBA2
; PRINT STRING AT (22) TO CURRENT
;   OUTPUT DEVICE
STROUT = $AB24
; COPY FAC1 TO ØØ57
F1T57   = $BBCA
```

```
; FAC1 = FAC1 X FVAR
F1XFV  = $BA28
; COPY FAC1 TO MEMORY POINTED TO BY
;    X-REG AND Y-REG
F1TMEM = $BBD4
; FAC1 = FAC1 X 1Ø
F1X1Ø  = $BAE2
; FAC1 = FAC1 / 1Ø
F1D1Ø  = $BAFE
; CHECK SIGN OF FAC1
SGNF1  = $BC2B
; COPY FAC1 TO FAC2
F1TF2  = $BCØC
; FAC1 = FAC1 X FAC2
F1XF2  = $BA2B
; FAC1 = FVAR/FAC1
FVDF1  = $BBØF
; COPY MEMORY POINTED TO BY A Y TO FAC2
MEMTF2 = $BA8C
; FAC1 = FAC2/FAC1
F2DF1 = $BB12
; FAC1 = FAC1 + FVAR
F1PFV = $B867
; FAC1 = FAC2 - FAC1
F2SF1 = $B853
; FAC1 = FVAR - FAC1
FVSF1 = $B85Ø
; FAC1 = FAC1 + ACCUM
F1PACC = $BD7E
; FAC1 = FAC1 + FAC2
F1PF2 = $B86A
; EXPONENTIATION
EXPON = $BF7B
; FAC1 = FAC1 AND FAC2
FPAND = $AFE9
; FAC1 = FAC1 OR FAC2
OR   = $AFE6
; FAC1 = SQR(FAC1)
SQR = $BF71
; FAC1 = FVAR/FAC1
FVDF1 = $BBØF
; FAC1 = INT(FAC1)
INT    = $BCCC
; COPY FAC1 TO ØØ5C
 F1T5C = $BBC7
; COMPARE FAC1 TO FVAR
COMP = $BC5B
; CONVERT FLOATING POINT TO X-REG
FLTX = $B7A1
```

```
; CONVERT FLOATING POINT TO ACCUM
;     AND Y-REG
FLTAY = $B1AA
; CONVERT Y-REG TO FLOATING POINT
YFLT = $B3A2
; POLYNOMIAL EVALUATION
POLY = $E059
      *=$C500
```

Example 1

```
; FLOATING POINT EXAMPLE #1
;
;CONVERT ACCUM AND X-REG TO ASCII STRING
; AND SEND TO CURRENT OUTPUT DEVICE
;
      LDA #$05
      LDX #$03
      JSR AXOUT
      RTS
```

Example 2 deals with two problems of machine language routines that use floating point routines. One problem is how to input a floating point into FAC1. The second problem is how to display or store the result of the floating point routine.

This example handles both problems. The result is displayed on the screen. Example 5 shows how to store the resulting FAC1 value into a floating point variable. This example uses CHRGET so the JSR E3BF is needed to initialize CHRGET in page 0. Next the input buffer is cleared. Then JSR ABF9 to send the ? prompt to the screen, which also receives characters from the current input device using CHRIN (assumes you will enter a number in standard or scientific format) and fills the BASIC input buffer at 0200.

Upon return, the X-register contains $FF and the Y-register contains $01. Then CHRGET and BCF3 are used to convert the ASCII string in the buffer to a floating point number in FAC1.

Now that a value has been input into FAC1, the rest of the routine is used to convert it back to an ASCII string and display it on the current output device. BDDD converts FAC1 to an ASCII string at 0100. The following instructions are used to determine the length of the string by searching for the $00 end-of-string byte, and then storing this length in the accumulator. Then (22) is set to point to this string at 0100 and the routine to PRINT a string is called, which sends the string pointed to by (22) to the current output device.

Example 2

```
; FLOATING POINT EXAMPLE #2
;
; INPUT ASCII NUMBER FROM SCREEN TO
; FLOATING POINT.
; RE-CONVERT FROM FLOATING POINT
; TO ASCII STRING AND PRINT TO SCREEN
;
      JSR INTCGT   ;INITIALIZE CHRGET
      LDA #$ØØ
      LDY #$59
CLEAR  STA $Ø2ØØ,Y ;CLEAR BASIC INPUT
;                    BUFFER
      DEY
      BNE CLEAR
      JSR PROMPT   ;PROMPT FOR ? AND
;                    FILL BUFFER BY
;                    READING INPUT DEVICE
      STX $7A      ;X AND Y POINT TO
;                    Ø1FF ON RETURN
      STY $7B
      JSR CHRGET
      JSR ASCFLT   ;CONVERT ASCII STRING
;                    AT Ø2ØØ TO FAC1
;                    FLOATING POINT
      JSR FACASC   ;CONVERT FAC1
;                    FLOATING POINT TO
;                    ASCII STRING AT Ø1ØØ
      LDY #$FF
FNDEND INY          ;DETERMINE END OF
;                    STRING BY SEARCHING
      LDA $Ø1ØØ,Y ;FOR THE END-OF-
      BNE FNDEND   ; STRING $ØØ BYTE
      INY
      TYA
      PHA
      LDA #$ØØ
      STA $22      ;SET (22) TO POINT
;                   ;TO STRING AT Ø1ØØ
      LDA #$Ø1
      STA $23
      PLA
      JSR STROUT   ;PRINT STRING AT
;                    (22) TO CURRENT
;                    OUTPUT DEVICE
      RTS
```

Since the previous example handled input so well, the following example makes the input portion into a subroutine, INSUB. Many of the remaining examples use this subroutine to input numbers by doing a JSR INSUB.

Example 3

```
; FLOAT EXAMPLE #3
;
; SUBROUTINE TO HANDLE INPUT OF A NUMBER
; FROM THE SCREEN AND CONVERT TO
;  FLOATING POINT IN FAC1
;
INSUB  JSR INTCGT    ;INITIALIZE CHRGET
      LDA #$00
      LDY #$59
CBUFF  STA $0200,Y   ;CLEAR BASIC INPUT
      DEY           ;  BUFFER
      BNE CBUFF
      JSR PROMPT    ;PROMPT FOR ? -
;                     FILL INPUT BUFFER
      STX $7A       ;X AND Y POINT TO
;                     01FF ON RETURN
      STY $7B
      JSR CHRGET
      JSR ASCFLT    ;CONVERT ASCII
;                     STRING AT 0200
;                     TO FAC1 FLOATING
;                     POINT
      RTS
```

Example 4 also handles input of a floating point number. However, in this case a floating point number is read from a location in memory. You could use this example as a basis for reading floating point variable data areas of your program. The example then converts this floating point number to an ASCII string and displays it on the screen.

Example 4

```
; FLOAT EXAMPLE #4
;
; READ FLOATING POINT VARIABLE FROM
; MEMORY
;
      JSR INTCGT    ;INITIALIZE CHRGET
      LDX #$83      ;SET POINTER TO
;                     FVAR AS LOCATION
;                            FOR FVAR MINUS ONE
      LDY #$C5
      STX $7A
      STY $7B
      JSR CHRGET
      JSR ASCFLT         ;CONVERT ASCII
;                          STRING TO FLOATING
;                          POINT IN FAC1
      JSR FACASC         ;CONVERT FAC1
;                          FLOATING POINT
;                          TO ASCII STRING
;                          AT Ø1ØØ
      LDY #$FF
SBUFX  INY                ;DETERMINE LENGTH
;                          OF STRING BY
;                          SEARCHING FOR
      LDA $Ø1ØØ,Y        ;THE END-OF-STRING
      BNE SBUFX          ;$ØØ BYTE
      INY
      TYA
      PHA
      LDA #$ØØ           ;SET (22) TO POINT
;                          TO STRING AT Ø1ØØ
      STA $22
      LDA #$Ø1
      STA $23
      PLA                ;RESTORE LENGTH OF
;                          OF STRING
      JSR STROUT         ;PRINT STRING AT
;                          (22) TO CURRENT
;                          OUTPUT DEVICE
      RTS
      .BYTE '25.35E3'
      .BYTE Ø
```

Example 5 uses the previous input routine, which is now a subroutine, INSUB, to get two numbers from the current input device. The numbers are multiplied and the result is stored into a memory area for a floating point variable rather than sending the result to the screen.

Example 5

```
; FLOAT EXAMPLE #5
;
; STORING FLOATING POINT ACCUMULATOR
;  INTO MEMORY
;
```

```
      JSR INSUB   ;INPUT 1ST NUMBER
      JSR F1T57   ;COPY FAC1 TO 57-5B
      JSR INSUB   ;INPUT 2ND NUMBER
      LDA #$57
      LDY #$ØØ    ;POINT TO 1ST NUMBER
      JSR F1XFV   ;FAC1 = FAC1 X FVAR
      LDX #$A4    ;SET POINTER TO AREA
;                   TO COPY RESULT TO
      LDY #$C5
      JSR F1TMEM ;COPY FAC1 TO MEMORY
      RTS
      .BYTE Ø,Ø,Ø,Ø,Ø,Ø
```

Example 6 also prompts for two numbers and multiplies them. In this case the result is converted to an ASCII string and displayed on the screen. Rather than again writing the code to convert FAC1 to a string, determine the length of the string at 0100, set the pointer to the string, and call the routine to print the string; a subroutine is created that handles just this function. This subroutine, OUTSUB, is located at C5C1. Any further references to OUTSUB in the following examples are also calling this subroutine.

Example 6

```
; FLOAT EXAMPLE #6
;
; FLOATING POINT MULTIPLICATION
;
      JSR INSUB    ;GET FIRST NUMBER
      JSR F1T57    ;COPY FAC1 TO ØØ57
      JSR INSUB    ;GET SECOND NUMBER
      LDA #$57
      LDY #$ØØ     ;SET POINTER TO FVAR
      JSR F1XFV    ;FAC1 = FAC1 X FVAR
      JSR FACASC   ;CONVERT FAC1 TO
;                    ASCII STRING AT Ø1ØØ
      JSR OUTSUB   ;PRINT STRING AT
;                    Ø1ØØ TO OUTPUT DEVICE
      RTS
; COMMON SUBROUTINE TO PRINT STRING
; AT Ø1ØØ TO CURRENT OUTPUT DEVICE
;
;
OUTSUB LDY #$FF
NXTCHR INY            ;DETERMINE LENGTH
;                      OF STRING BY
;                      SEARCHING FOR THE
;                          $ØØ END-OF-STRING
;                          BYTE
      LDA $Ø1ØØ,Y
      BNE NXTCHR
      INY
      TYA
      PHA
      LDA #$ØØ           ;PUT POINTER TO
;                          STRING IN (22)
      STA $22
      LDA #$Ø1
      STA $23
      PLA                ;GET LENGTH OF
;                          STRING
      JSR STROUT         ;PRINT STRING TO
;                          CURRENT OUTPUT
;                          DEVICE
      RTS
```

Unless otherwise stated, in the following examples the common method of input is to call INSUB, and the common method of output is to call OUTSUB.

Example 7 multiplies FAC1 by 10.

Example 7

```
; FLOAT EXAMPLE #7
;
; MULTIPLY FAC1 BY 1Ø
;
;
      JSR INSUB          ;GET NUMBER
      JSR F1X1Ø          ;FAC1 = FAC1 X 1Ø
      JSR FACASC         ;CONVERT FAC1 TO
;                          ASCII STRING AT Ø1ØØ
      JSR OUTSUB         ;PRINT STRING AT
;                          Ø1ØØ TO CURRENT
;                          OUTPUT DEVICE
      RTS
```

Example 8 divides FAC1 by 10. You might notice that the signs are incorrect if a negative number appears in either the dividend or divisor. The divide FAC1 by 10 routine assumes both numbers are positive. The multiply by FAC1 does not have the problem with the signs.

Example 8

```
; FLOAT EXAMPLE #8
;
; DIVIDE FAC1 BY 10
;
; (SIGN IS LOST IN RESULT)
;
      JSR INSUB    ;GET NUMBER
      JSR F1D10    ;FAC1 = FAC1/10
      JSR FACASC   ;CONVERT FAC1 TO
;                   ASCII STRING AT 0100
      JSR OUTSUB   ;PRINT STRING AT
;                   0100 TO CURRENT
;                   OUTPUT DEVICE
      RTS
```

Example 9 fixes the problem with the sign in the divide FAC1 by 10 routine by using BC2B to check the sign of FAC1.

Example 9

```
; FLOAT EXAMPLE #9
;
; DIVIDE FAC1 BY 10 WITH SIGN
;  CORRECTION
;
      JSR INSUB    ;GET NUMBER
      JSR SGNF1    ;CHECK SIGN OF FAC1
      PHA
      JSR F1D10    ;FAC1 = FAC1/10
      PLA
      TAX
      INX
      BNE NOTNEG
      LDA #$80     ;IF NEGATIVE
;                   ORIGINAL NUMBER
;                   IN FAC1
      STA $66      ;THEN MAKE THE
;                   QUOTIENT NEGATIVE
NOTNEG JSR FACASC   ;CONVERT FAC1
;                   FLOATING POINT TO
;                   ASCII STRING AT 100
      JSR OUTSUB
      RTS
```

Example 10 multiplies FAC1 by FAC2. The accumulator has to contain the value of the exponent of FAC1, 61, before calling the multiplication.

Example 10

```
; FLOAT EXAMPLE #1Ø
;
; FAC1 SQUARED
;
      JSR INSUB
      JSR F1TF2     ;COPY FAC1 TO FAC2
      LDA $61
      JSR F1XF2     ;FAC1 = FAC1 X FAC2
      JSR FACASC
      JSR OUTSUB
      RTS
```

Example 11 divides two numbers. If you try to divide by zero, you'll get the division by zero error and be thrown into BASIC.

Example 11

```
; FLOAT EXAMPLE #11
;
; FLOATING POINT DIVISION
; FAC1 = FVAR/FAC1
      JSR INSUB     ;GET DIVIDEND
      JSR F1T57     ;COPY FAC1 TO ØØ57
      JSR INSUB     ;GET DIVISOR
      LDA #$57
      LDY #$ØØ
      JSR FVDF1     ;FAC1 = FVAR/FAC1
      JSR FACASC
      JSR OUTSUB
      RTS
```

Example 12 divides FAC2 by FAC1. Again, the accumulator must contain 61.

Example 12

```
; FLOAT EXAMPLE #12
;
; FLOATING POINT DIVISION
;
; FAC1 = FAC2/FAC1
      JSR INSUB     ;GET DIVIDEND
      JSR F1T57     ;COPY FAC1 TO ØØ57
      JSR INSUB     ;GET DIVISOR
      LDA #$57
      LDY #$ØØ
```

```
        JSR MEMTF2   ;COPY 0057 TO FAC2
        LDA $61      ;GET EXPONENT OF FAC1
        JSR F2DF1    ;FAC1 = FAC2/FAC1
        JSR FACASC
        JSR OUTSUB
        RTS
```

Example 13 adds a FVAR to FAC1.

Example 13

```
; FLOAT EXAMPLE #13
;
; FLOATING POINT ADDITION
;
; FAC1 = FAC1 + FVAR
;
        JSR INSUB
        JSR F1T57
        JSR INSUB
        LDA #$57
        LDY #$00
        JSR F1PFV     ;FAC1=FAC1+FVAR
        JSR FACASC
        JSR OUTSUB
        RTS
```

Example 14 subtracts FAC1 from FAC2. Floating point subtraction does not require you to LDA 61 before using the routine.

Example 14

```
; FLOAT EXAMPLE #14
;
; FLOATING POINT SUBTRACTION
;
;  FAC1 = FAC2 - FAC1
;
         JSR INSUB
         JSR F1T57
         JSR INSUB
         LDA #$57
         LDY #$00
         JSR MEMTF2   ;COPY 0057 TO FAC2
         JSR F2SF1    ;FAC1=FAC2-FAC1
         JSR FACASC
         JSR OUTSUB
         RTS
```

Example 15 subtracts FAC1 from a floating point variable.

Example 15

```
; FLOAT EXAMPLE #15
;
; FLOATING POINT SUBTRACTION
;
;  FAC1 = FVAR - FAC1
;
       JSR INSUB
       JSR F1T57
       JSR INSUB
       LDA #$57
       LDY #$ØØ
       JSR FVSF1    ;FAC1=FVAR-FAC1
       JSR FACASC
       JSR OUTSUB
       RTS
```

Example 16 adds the accumulator to FAC1.

Example 16

```
; FLOAT EXAMPLE #16
;   ADD ACCUM TO FAC1
;
       JSR INSUB
       LDA #$Ø5
       JSR F1PACC    ;FAC1=FAC1+ACCUM
       JSR FACASC
       JSR OUTSUB
       RTS
```

Example 17 adds FAC2 to FAC1. Again, the accumulator must contain 61 at entry.

Example 17

```
; FLOAT EXAMPLE #17
;
;
; FLOATING POINT ADDITION
;
; FAC1 = FAC1 + FAC2
;
;
       JSR INSUB
       JSR F1T57   COPY FAC1 TO ØØ57
       JSR INSUB
       LDA #$57
       LDY #$ØØ
       JSR MEMTF2 ;COPY ØØ57 TO FAC2
       LDA $61    ;GET EXPONENT OF
;                   FAC1
       JSR F1PF2  ;FAC1=FAC1+FAC2
       JSR FACASC
       JSR OUTSUB
       RTS
```

Example 18 demonstrates exponentiation.

Example 18

```
;  FLOAT EXAMPLE #18
;
;   EXPONENTIATION
;
;
       JSR INSUB  ;GET POWER TO BE
;                   RAISED TO
       JSR F1T57  ;COPY FAC1 TO ØØ57
       JSR INSUB  ;GET NUMBER
       LDA #$57
       LDY #$ØØ
       JSR MEMTF2 ;COPY ØØ57 TO FAC2
       LDA $61
       JSR EXPON  ;RAISE FAC1 TO THE
;                   POWER FAC2
       JSR FACASC
       JSR OUTSUB
       RTS
```

Example 19 does a logical AND of FAC1 and FAC2.

Example 19

```
; FLOAT EXAMPLE #19
;
;
; LOGICAL AND
;
       JSR INSUB
       JSR F1T57
       JSR INSUB
       LDA #$57
       LDY #$ØØ
       JSR MEMTF2
       JSR FPAND     ;FAC1 = FAC1 AND
;                      FAC2
       JSR FACASC
       JSR OUTSUB
       RTS
```

Example 20 does a logical OR of FAC1 and FAC2.

Example 20

```
; FLOAT EXAMPLE #2Ø
;
; LOGICAL OR
;
       JSR INSUB
       JSR F1T57
       JSR INSUB
       LDA #$57
       LDY #$ØØ
       JSR MEMTF2
       JSR OR        ;FAC1 = FAC1 OR FAC2
                     ;FAC2
       JSR FACASC
       JSR OUTSUB
       RTS
```

Example 21 computes the square root of FAC1.

Example 21

```
; FLOAT EXAMPLE #21
;
; SQUARE ROOT
       JSR INSUB
       JSR SQR       ;FAC1 = SQR(FAC1)
       JSR FACASC
       JSR OUTSUB
       RTS
```

Example 22 performs a division and then uses INT to return only the integer result of the division.

Example 22

```
; FLOAT EXAMPLE #22
;
; RETURN INTEGER PORTION OF QUOTIENT
; FROM FLOATING POINT DIVISION
;
;
       JSR INSUB
       JSR F1T57
       JSR INSUB
       LDA #$57
       LDY #$00
       JSR FVDF1    ;FAC1=FVAR/FAC1
       JSR F1T57    ;COPY FAC1 TO 0057
       JSR INT      ;FAC1 = INT(FAC1)
       JSR F1T5C    ;COPY FAC1 TO 005C
       JSR FACASC
       JSR OUTSUB
       RTS
```

Example 23 uses the comparison routine to compare two numbers and display the larger number.

Example 23

```
; FLOAT EXAMPLE #23
;
; COMPARE TWO NUMBERS AND DISPLAY
;
       JSR INSUB    ;GET 1ST NUMBER
       JSR F1T57    ;COPY FAC1 TO 0057
       JSR INSUB    ;GET 2ND NUMBER
       JSR F1T5C    ;COPY FAC1 TO 005C
       LDA #$57
       LDY #$00
       JSR COMP    ;COMPARE FAC1 TO
;                    FVAR AT 0057
       TAX         ;SAVE RESULT OF
;                    COMPARE
       INX
       BNE MLESS   ;IF MEMORY <= FAC1
;                    BRANCH
       LDA #$57
       LDY #$00
       CLC
       BCC MGREAT  ;IF MEMORY > FAC1
MLESS   LDA #$5C
       LDY #$00
MGREAT  JSR FVTF1   ;COPY FVAR POINTED
;                    TO BY A Y TO FAC1
       JSR FACASC
       JSR OUTSUB
       RTS
```

Example 24 converts a floating point number to an integer from 0 to 255 in the X-register. The display from HESMON shows the contents of the X-register. The G 00FA is the HESMON command to convert a hex number to its decimal value.

Example 24

```
;    FLOAT EXAMPLE #24
;
;   CONVERT  FLOATING POINT TO X-REG
;
       JSR INSUB
       LDA #$ØØ
       STA $ØD
       JSR FLTX
       RTS
```

Example 25 converts a floating point number to a two-byte integer in the accumulator and Y-register.

Example 25

```
;   FLOAT EXAMPLE #25
;
;  CONVERT FLOATING POINT TO ACCUM
;   AND Y-REG
;
       JSR INSUB
       JSR FLTAY
       RTS
```

Example 26 converts a two-byte integer in the accumulator and X-register to a floating point value, then to a string, and then displays the string on the current output device. CHRGET is not needed by this routine.

Example 26

```
;   FLOAT EXAMPLE #26
;
;  CONVERT INTEGER TO FLOATING POINT
;  TO ASCII STRING AND PRINT STRING
;    DOES NOT NEED CHRGET
;
       LDA #$25
       LDX #$Ø7
       JSR AXOUT
       RTS
```

Example 27 converts the Y-register to a floating point number in FAC1.

Example 27

```
;   FLOAT EXAMPLE #27
;
;   CONVERT Y-REG TO FLOATING POINT
;
        LDY #$FE
        JSR YFLT
        JSR FACASC
        JSR OUTSUB
        RTS
```

Example 28 uses the routine at E059 which computes polynomials using Horner's method. A series of floating point variables is input. These FVARs are stored in consecutive memory locations, preceded by a one-byte value that gives the order of the series. This routine allows you to dynamically generate a series of order two and to compute the series based on the value that is input after having read all the series values. The first two executions used the polynomial $3*X^2+2*X+5$. The last execution used the polynomial $25*X^2-9*X+66$. By changing the number of series terms and the order, you could use this routine to dynamically generate any order series.

Example 28

```
;   FLOAT EXAMPLE #28
;
;
;  COMPUTE POLYNOMIAL OF ORDER 2
;
        LDA #$9A   :THE SERIES WILL BE
;                    STORED AT $C29A--
        STA $FD    ;ONE BYTE PAST ORDER
        LDA #$C2
        STA $FE
        LDA #$Ø3   ;ASK FOR THREE TERMS
;                    IN THE SERIES
        STA $FF
NEXSER   JSR INSUB
        LDX $FD
        LDY $FE
        JSR F1TMEM   ;COPY FAC1 TO
;                      MEMORY LOCATION
;                      POINTED TO BY
;                      X AND Y
        CLC
        LDA #$Ø5      ;MOVE POINTER TO
;                      NEXT LOCATION
;                      FOR STORING
;                      SERIES VALUE
        ADC $FD
        STA $FD
        DEC $FF
        BNE NEXSER
        LDA #$Ø2      ;SET ORDER FOR
;                      SERIES
        STA $C299
        JSR INSUB
        LDA #$99
        LDY #$C2
        JSR POLY
        JSR FACASC
        JSR OUTSUB
        RTS
```

Modifying BASIC

Several different methods can be used to add new commands or to modify BASIC. This section explores some of those methods.

For new commands to behave like the normal BASIC keywords—to be tokenized, detokenized and listed, and executed—you must intercept the vectors at 0300, error routine vector; 0304, tokenization vector; 0306, list tokens vector; 0308, command vector; and 030A, function vector. Two programs provided in this section demonstrate this interception-of-vectors approach to adding new commands to BASIC. BASIC uses the tokens from $80 through $CB for its keywords. One program is shown that allows you to use the tokens from $CC through $FE to implement up to 51 new commands and functions. $FF is reserved for the value of PI by BASIC. A second program allows up to 255 new tokens to be implemented by using two-byte tokens to represent the new commands and functions. With a two-byte token format, each two-byte token contains the same first-byte value of $60. This unique value allows easy detection of the new commands and functions during execution and during listing. Some commercial extensions to BASIC. such as *Simons' BASIC* make use of two-byte tokens.

You can also modify the CHRGET routine to intercept each character received from the input device. A program is included that illustrates one approach to modifying CHRGET.

Another technique is to intercept the vector to the IRQ interrupt handler. The NMI interrupt vector can also be intercepted, and a program that does just that is also included.

The four programs in this chapter which add new commands—one-byte tokenization, two-byte tokenization, CHRGET wedge, and NMI interrupt wedge—all have a few commands included to demonstrate how you can actually add the new commands. The programs provide a framework for adding your own commands. All four programs are coded to run on the Commodore 64. For the two-byte tokenization and CHRGET wedge, comments are provided that give the equates and other changes needed to run the programs on the VIC-20.

Bank Switching on the Commodore 64

One new technique for modifying BASIC is available on the Commodore 64 that involves bank switching of memory. The BASIC ROM contains a RAM hidden underneath. Whenever you write to a ROM location, the value is stored in the underlying RAM. Thus you can directly modify data or instructions by merely POKEing a new value to ROM. Then you just bank switch the ROM out and the RAM in, and you now have your modified BASIC. You need to POKE every location in ROM to the underlying RAM so that the unmodified portion of BASIC still exists after you do the switch. Through this technique you can insert a front-end or back-end wedge to any section of BASIC. The following example shows a BASIC program to do this modification and bank switch. The modification is to change the floating point variable that is used as the default step size in a FOR loop. The default step size is changed from one to two. The example then executes a FOR loop using the new default step size, and the PRINT statement illustrates that the change has indeed taken effect. This is a trivial example that just scratches the surface of what can be changed. The two-byte tokenization program also contains an example of using this bank switching to modify the normal tokenization routine.

Modifying BASIC

```
1Ø FOR X = 4Ø96Ø TO 49151
2Ø Y = PEEK(X)
25 POKE(X),Y
3Ø NEXT
4Ø POKE47548,13Ø
5Ø POKE1,PEEK(1)AND254
7Ø FOR X = 1 TO 2Ø
8Ø PRINTX
9Ø NEXT
```

The four programs at the end of this section should be used as a framework for building your own new command package. Below are comments about each of the programs that might not be obvious from looking at the program listings.

Two-Byte Tokenization

The two-byte tokenization program uses two-byte tokens with the first byte always being $60. This two-byte version has the obvious advantage of allowing many more new commands and functions to be implemented. If more than 255 new keywords are needed, you could use the program as a model and go to three-byte tokens.

A word of caution about the new keyword table (label TOKTAB): Normally you should not have keywords that contain embedded Commodore BASIC keywords. For example, AUTON contains the embedded TO. Since the program calls the normal tokenization routine before examining the new keywords, the normal tokenization routine would tokenize the TO in AUTON and thus AUTON would never be found. You may suggest that this would not be a problem if the new tokenization routine was executed first and the normal tokenization routine executed second. However, this presents a problem too, since normal tokenization throws away any bytes >=$80. Thus you could have new keywords only from $6000 through $607F rather than from $6000 through $60FF. The new keywords should be listed starting with commands, followed by functions that require only one argument, followed by functions that require two or more arguments. If you change the number of keywords in any of these categories, be sure to modify the FNCONE, TWOARG, and ENDFNC fields accordingly. Just as in the normal keyword table, the new keyword must have the final letter of the keyword OR'd with $80 so its high-order bit is on.

You should also notice that the COMADR table of command routine addresses and the FNCADR table of function routine addresses are in the same relative order as the keywords listed in the TOKTAB table. This relative order must be maintained since an index into one table can be used to index into the other tables.

The new tokenization routine is modeled after the existing tokenization routine, just changing the parts that are necessary. For example, since the normal tokenization routine has converted the Commodore BASIC keywords into tokens, the new tokenization routine does not discard bytes with values >=$80. To allow for more than 256 letters in the new keyword table, a pointer to the table is maintained, and when necessary, the page portion (low-order byte) of the pointer is incremented to point to the next set of 256 letters. The tests for quote mode, for DATA statements, and for REM statements are maintained in the new version.

The new LIST routine checks for a byte of $60, and if not found continues with the normal LIST routine. When a $60 is found, the second byte of the token is used to actually index into the table of keywords.

During the new command execution routine, a temporary modification of CHRGET is needed. Since CHRGET normally skips spaces in retrieving the next character, it would skip over the $20 in the $6020 two-byte token. CHRGET is modified to ignore the skip for spaces, and then is restored to its test for spaces once the CHRGET for the second byte of the two-byte token is complete.

The new error message handler only needs to handle error messages greater than $1E, since $1E is the highest normal error message number. Because the Commodore 64 normal error message handler is different from the VIC-20 normal error message handler, slightly different routines are used for the two computers. In the individual commands, the BRDR and SCREEN commands differ on the VIC-20 and the Commodore 64 since two registers are used on the 64 to control the screen and border colors while only one register controls these colors on the VIC.

The final section of the program that is headed by "Allow Embedded Keywords" gives a technique for allowing the normal Commodore keywords to be embedded in the new keywords, except at the very end of the keyword. This modification uses the same bank switching of RAM that was demonstrated previously in the FOR step size example. This time, though, the tokenization routine is modified to jump to this MODTOK routine that checks to see whether a space, $00, or colon follows a keyword before allowing that keyword to be tokenized. If you do use this modification, you must be sure to follow all keywords by a space, or syntax errors result. Also, this technique is not possible on the VIC-20 since the VIC does not permit the bank switching of the underlying RAM.

One-Byte Tokenization

The one-byte tokenization program provides a method for adding new keywords to implement commands and functions for the tokens from $CC through $FE. Although this program does not allow for as many new keywords as the two-byte tokenization routine does, it does provide for somewhat simpler and faster code.

CHRGET

Wedging into CHRGET is a technique used by many programs such as the DOS Wedge from Commodore. The JMPs to wedges from CHRGET are normally used from either $0073 or $007C. This program uses the method of placing a JMP to the wedge at $007C following the LDA. When the wedge does not find a new command, it completes CHRGET by jumping to the final unchanged portion of CHRGET in the ROM copy of CHRGET. One of the first things the wedge does is to see what routine called CHRGET by obtaining the return address from the stack. Because the two routines we are concerned with—the CHRGET call from direct mode and the CHRGET call from program mode—have either a first- or second-byte return address that is unique among all callers of CHRGET, we can just test that one byte to see if CHRGET was called by either of these two routines. If it wasn't, the wedge just restores the character retrieved by CHRGET and completes the normal execution of CHRGET. Also, to allow for execution of the wedge commands from a program, you must provide the token value for any character that can be tokenized. In this example, the / character is tokenized.

This wedge into CHRGET provides the following new commands: / to set the screen color, ! to set the border color, £ to define which keys repeat when held down, and # to remove the wedge from CHRGET. Notice the slight changes in the individual commands from the versions of the commands in the one- and two-byte tokenization programs. See Raeto Collin West's *Programming the PET/CBM* or Russ Davies' *Mapping the VIC* for more comments about wedging into CHRGET.

NMI

The NMI interrupt wedge makes use of the fact that the RESTORE key generates an NMI interrupt. Normally this interrupt is ignored unless the STOP key is also held down, in which case the BREAK or warm start function is executed. After you hit the RESTORE key and cause the NMI interrupt, the new NMI interrupt handler calls the Kernal GETIN routine to retrieve a character from the keyboard. I found that it was easiest to just get one character from the keyboard and not to try to retrieve any operands of a command. Thus the previous KEYS command was broken up into three separate commands.

The fact that the RESTORE key causes a GETIN from the keyboard allows you to use RESTORE to halt a listing or a program execution. To resume the listing or the program execution, just hit RETURN or any key that is not defined as a command. Hitting a key that is defined as a command will execute the command, and then it also continues the listing or program execution.

The first part of the new NMI interrupt handler tests to see whether the handler is being called recursively, and if so, it exits. This was necessary to prevent having to hit a command key twice in response to what I thought was one entry of the RESTORE key. If you compare the new NMI interrupt handler to the normal one, you will also notice that the code that deals with RS-232 is deleted from the new version. Thus, don't use this program with RS-232, or include the RS-232 code and rewrite the new interrupt handler.

Each of these methods of adding new commands has its advantages. Pick the method that best serves your purposes.

New BASIC Commands—Two-Byte Tokens

```
; *********************************
;
;
;    NEW COMMANDS FOR BASIC
;
;      TWO-BYTE TOKEN VERSION
;      ALLOWING POSSIBLE 255 NEW
;       COMMANDS
;
;
;   COMMODORE 64 VERSION
;
;    COMMENTS INDICATE CHANGES
;    NEEDED FOR VIC-20 VERSION
;
;
;*****************************
; EQUATES FOR THE COMMODORE-64
;
;
NTOK = $A57C
NLST = $A724
OCHR = $AB47
LRET1 = $A6EF
LRET2 = $A6F3
```

```
NSTT = $A7AE
NCMD = $A7E7
EVALP = $AEF1
NCHK = $AD8D
NFNC = $AE8D
EVAL = $AD9E
F1TX = $B7A1
SYNTAX = $AFØ8
BREG = $DØ2Ø
SREG = $DØ21
KREG = $Ø28A
CSTRT = $E394
RVECT = $E453
F1T57 = $BBCA
INTF1 = $BCCC
SUBF1 = $B85Ø
CHKLP = $AEFA
CHKCM = $AEFD
CHKRP = $AEF7
F1DIV = $BBØF
NERR = $A43A
ERRCNT = $A447
READY = $A474
;
;
; EQUATES FOR THE VIC-2Ø
;
; NTOK = $C57C
; NLST = $C724
; OCHR = $CB47
; LRET1 = $C6EF
; LRET2 = $C6F3
; NSTT = $C7AE
; NCMD = $C7E7
; EVALP = $CEF1
; NCHK = $CD8D
; NFNC = $CE8D
; EVAL = $CD9E
; F1TX = $D7A1
; SYNTAX = $CFØ8
; BREG = $9ØØF
; SREG = $9ØØF
; KREG = $Ø28A
; CSTRT = $E378
; RVECT = $E45B
; F1T57 = $DBCA
; INTF1 = $DCCC
; SUBF1 = $D85Ø
; CHKLP = $CEFA
; CHKCM = $CEFD
; CHKRP = $CEF7
; F1DIV = $DBØF
; NERR = $C43A
; ERRCNT = $C447
; READY = $C474
;
; ******************************
;
; STARTING LOCATION OF COMMAND PACKAGE
; ON C-64 IS $CØØØ  (49152)
;
; ON VIC-2Ø IS $62ØØ
;
;  ORIGIN FOR COMMODORE 64:
;
       * = $CØØØ
;
;  ORIGIN FOR VIC-2Ø:
;      * = $62ØØ
;
;
; SET UP VECTORS TO NEW ROUTINES
; TO CHANGE VECTORS:
;
;  FOR COMMODORE 64 - SYS 49162
;  FOR VIC-2Ø       - SYS 25Ø98
;
; ******************************
;
;     ON THE VIC-2Ø ONLY
;  RUN FOLLOWING BASIC PROGRAM
;  TO RESERVE MEMORY FOR COMMAND
;   PACKAGE AT TOP OF MEMORY
;
;  IF YOU CHANGE THE STARTING
;  LOCATION OF YOUR VIC COMMAND
;  PACKAGE,THEN MODIFY THE POKES
;  TO 781 AND 782 ACCORDINGLY.
;  THE FOLLOWING PROGRAM RESETS
;  THE TOP OF MEMORY FOR BASIC
;   TO $5FFF
;
;  POKE 783,Ø    CLEAR THE CARRY
;  POKE 781,255  X-REG TO $FF
;  POKE 782,95   Y-REG TO $5F
;  SYS 65433     TO DO MEMTOP
;  SYS 58232     COLD START BASIC
;*******************************
```

```
;
ERRV .WORD ERRX
TOKV .WORD TOKX
LSTV .WORD LISTX
COMV .WORD COMX
FNCV .WORD FUNCX
;
      SEI
      LDA ERRV
      STA $0300   ; ($0300) =
      LDA ERRV+1  ; ERROR MESSAGE
      STA $0301   ; VECTOR
      LDA TOKV
      STA $0304   ; ($0304) =
      LDA TOKV+1  ; TOKENIZATION
      STA $0305   ;VECTOR
      LDA LSTV
      STA $0306   ; ($0306) =
      LDA LSTV+1  ; LIST TOKENS
      STA $0307   ; VECTOR
      LDA COMV
      STA $0308   ; ($0308) =
      LDA COMV+1  ; COMMAND VECTOR
      STA $0309
      LDA FNCV
      STA $030A   ; ($030A) =
      LDA FNCV+1   ;FUNCTION VECTOR
      STA $030B
      CLI
      RTS
;
; TABLE OF NEW KEYWORDS
;
;
;NO ABBREVIATIONS(WITH SHIFTED KEY)
; PERMITTED BECAUSE NORMAL TOKENIZATION
; THROWS AWAY BYTES >= #$80
;
;
; BASIC KEYWORDS EMBEDDED IN NEW
; COMMANDS ONLY PERMITTED ON C-64
; IF NORMAL TOKENIZATION ROUTINE
; IS MODIFIED - SEE "MODTOK"
; ROUTINE FOR METHOD.
;
;  WITH THE TWO BYTE TOKENS USED
;  THE FIRST TOKEN IS ALWAYS $60
;  WHILE THE VALUE FOR THE SECOND
;  IS SHOWN FOLLOWING THE COMMAND
;  OR FUNCTION NAME BELOW
```

```
;
;   MANY COMMANDS ARE NOT ACTUALLY
;   IMPLEMENTED - THEY ARE LISTED
;   HERE TO PROVIDE A FRAMEWORK
;   YOU CAN BUILD ON
;
;   THE NEW KEYWORDS MUST BE LISTED
;   IN THE ORDER OF:
;      NEW COMMANDS
;      NEW FUNCTIONS WITH ONE ARGUMENT
;      NEW FUNCTIONS WITH TWO OR
;            MORE ARGUMENTS
;
;   NOTE THAT TOKENS BEGIN WITH
;   $Ø1 - NOT WITH $ØØ
;
;
ADRTOK .WORD TOKTAB
TOKTAB .BYTE 'ADUM'
      .BYTE $DØ ; ADUMP  $Ø1
      .BYTE 'APPN'
      .BYTE $C4  ; APPND $Ø2
; 'AUTON' - EMBEDDED KEYWORD 'TO'
      .BYTE 'AUTO'
      .BYTE $CE  ; AUTON $Ø3
      .BYTE 'BO'
      .BYTE $D8 ; BOX  $Ø4
      .BYTE 'CAS'
      .BYTE $C5 ; CASE $Ø5
      .BYTE 'CHANG'
      .BYTE $C5  ; CHANGE $Ø6
      .BYTE 'CIRCL'
      .BYTE $C5   ; CIRCLE $Ø7
      .BYTE 'COLLID'
      .BYTE $C5  ; COLLIDE $Ø8
      .BYTE 'COMG'
      .BYTE $D4  ; COMGT $Ø9
      .BYTE 'COP'
      .BYTE $D9  ; COPY $ØA
      .BYTE 'DE'
      .BYTE $CC  ; DEL  $ØB
      .BYTE 'DLA'
      .BYTE $D9  ; DLAY $ØC
      .BYTE 'DRA'
      .BYTE $D7  ; DRAW $ØD
      .BYTE 'DUM'
      .BYTE $DØ  ; DUMP $ØE
      .BYTE 'ERAS'
      .BYTE $C5  ; ERASE $ØF
      .BYTE 'GFIN'
      .BYTE $C4  ; GFIND  $12
      .BYTE 'HEL'
      .BYTE $DØ ; HELP  $13
      .BYTE 'HIME'
      .BYTE $CD  ; HIMEM $14
      .BYTE 'HIRE'
      .BYTE $D3  ; HIRES $15
      .BYTE 'HPE'
      .BYTE $CE ; HPEN   $16
      .BYTE 'HTA'
      .BYTE $C2  ; HTAB $17
      .BYTE 'INSER'
      .BYTE $D4  ; INSERT $18
      .BYTE 'LABE'
      .BYTE $CC ; LABEL  $19
      .BYTE 'LIN'
      .BYTE $C5  ; LINE  $1A
      .BYTE 'LOME'
      .BYTE $CD  ; LOMEM $1B
      .BYTE 'MERG'
      .BYTE $C5  ; MERGE $1C
      .BYTE 'MO'
      .BYTE $C4  ; MOD $1D
      .BYTE 'MOV'
      .BYTE $C5  ; MOVE $1E
      .BYTE 'TAPE'
      .BYTE $CD  ; TAPEM $1F
      .BYTE 'MULTI'
      .BYTE $C3  ; MULTIC $2Ø
      .BYTE 'OL'
      .BYTE $C4  ; OLD $21
; 'PAINTR' - EMBEDDED KEYWORD 'INT'
      .BYTE 'PAINT'
      .BYTE $D2  ; PAINTR $22
      .BYTE 'PFKE'
      .BYTE $D9  ; PFKEY  $23
; 'POINTR' - EMBEDDED KEYWORD 'INT'
      .BYTE 'POINT'
      .BYTE $D2  ; POINTR $24
      .BYTE 'PO'
      .BYTE $DØ  ; POP $25
      .BYTE 'PRO'
      .BYTE $C3  ; PROC $26
      .BYTE 'PRTUSN'
      .BYTE $C7  ; PRTUSNG $27
      .BYTE 'RENU'
      .BYTE $CD  ; RENUM $28
      .BYTE 'REPEA'
```

```
.BYTE $D4   ; REPEAT $29
.BYTE 'RLS'
.BYTE $D4   ; RLST  $2A
.BYTE 'SE'
.BYTE $D4   ; SET $2B
.BYTE 'SINGL'
.BYTE $C5   ; SINGLE  $2C
.BYTE 'SPRITE'
.BYTE $C3     ; SPRITEC $2D
.BYTE 'SPRITE'
.BYTE $C4     ; SPRITED $2E
.BYTE 'SPRITE'
.BYTE $CD     ; SPRITEM $2F
.BYTE 'SR'
.BYTE $D4   ; SRT $3Ø
.BYTE 'TRAC'
.BYTE $C5   ; TRACE $31
.BYTE 'UNLS'
.BYTE $D4   ; UNLST $32
.BYTE 'UNN'
.BYTE $D7     ; UNNW $33
.BYTE 'VDUM'
.BYTE $DØ   ; VDUMP $34
.BYTE 'VPE'
.BYTE $CE     ; VPEN $35
.BYTE 'VTA'
.BYTE $C2   ; VTAB $36
.BYTE 'WHIL'
.BYTE $C5   ; WHILE $37
.BYTE 'COL'
.BYTE $C4     ; COLD $38
.BYTE 'BRD'
.BYTE $D2     ; BRDR $39
.BYTE 'SCREE'
.BYTE $CE     ;SCREEN $3A
.BYTE 'VBAC'
.BYTE $CB     ; VBACK $3B
.BYTE 'KEY'
.BYTE $D3     ; KEYS $3C
.BYTE 'FRA'
.BYTE $C3    ; FRAC $3D
.BYTE 'JO'
.BYTE $D9    ; JOY $3E
.BYTE 'PEN'
.BYTE $D8    ; PENX $3F
.BYTE 'PEN'
.BYTE $D9    ; PENY $4Ø
.BYTE 'DI'
.BYTE $D6     ; DIV  $41
```

```
      .BYTE $00
;
;
;
; EQUATES SPECIFY WHERE COMMANDS END,
; FUNCTIONS BEGIN AND END, AND WHICH
; FUNCTIONS ONLY HAVE ONE ARGUMENT
;
FNCONE .BYTE $3D  ; LAST COMMAND
;                 TOKEN+1 (OR 1ST
;                 FUNCTION TOKEN)
TWOARG .BYTE $3E  ; 1ST TOKEN FOR
;                 FUNCTION THAT
;                 NEEDS TWO OR MORE
;                 ARGUMENTS
ENDFNC .BYTE $42  ; LAST TOKEN PLUS ONE
;
; COMMAND ADDRESS TABLE
;
;
; ADDRESSES USED IN THE TABLE
; ARE ONE LESS THAN ACTUAL START
; OF COMMAND BECAUSE THE ADDRESS
; IS PUSHED ONTO THE STACK.
; RTS, WHICH ADDS ONE TO THE
; ADDRESS IT PULLS, IS USED TO
; JUMP TO THE ACTUAL ROUTINE.
;
COMADR .WORD ADUMP-1
      .WORD APPND-1
      .WORD AUTON-1
      .WORD BOX-1
      .WORD CASE-1
      .WORD CHANGE-1
      .WORD CIRCLE-1
      .WORD COLLID-1
      .WORD COMGT-1
      .WORD COPY-1
      .WORD DEL-1
      .WORD DLAY-1
      .WORD DRAW-1
      .WORD DUMP-1
      .WORD ERASE-1
      .WORD FIND-1
      .WORD GCHANG-1
      .WORD GFIND-1
      .WORD HELP-1
      .WORD HIMEM-1
      .WORD HIRES-1
```

```
       .WORD HPEN-1
       .WORD HTAB-1
       .WORD INSERT-1
       .WORD LABEL-1
       .WORD LINE-1
       .WORD LOMEM-1
       .WORD MERGE-1
       .WORD MOD-1
       .WORD MOVE-1
       .WORD TAPEM-1
       .WORD MULTIC-1
       .WORD OLD-1
       .WORD PAINTR-1
       .WORD PFKEY-1
       .WORD POINTR-1
       .WORD POP-1
       .WORD PROC-1
       .WORD PRTUSN-1
       .WORD RENUM-1
       .WORD REPEAT-1
       .WORD RLST-1
       .WORD SET-1
       .WORD SINGLE-1
       .WORD SPRITC-1
       .WORD SPRITD-1
       .WORD SPRITM-1
       .WORD SRT-1
       .WORD TRACE-1
       .WORD UNLST-1
       .WORD UNNW-1
       .WORD VDUMP-1
       .WORD VPEN-1
       .WORD VTAB-1
       .WORD WHILE-1
       .WORD COLD-1
       .WORD BRDR-1
       .WORD SCREEN-1
       .WORD VBACK-1
       .WORD KEYS-1
;
;
; FUNCTION ADDRESS TABLE
;
;
FNCADR  .WORD FRAC
       .WORD JOY
       .WORD PENX
       .WORD PENY
       .WORD DIV
```

```
;
; TABLE OF ADDRESS OF NEW ERROR
; MESSAGES
ERRTAB .WORD ERR1
;
;
; NEW ERROR MESSAGES
ERR1   .BYTE  'INVALID KEY OPTIO'
     .BYTE  $CE   ; INVALID KEY OPTION
;
;
;
;  TOKENIZATION ROUTINE FOR
;     NEW COMMANDS
;
;
;
;
TOKX    JSR NTOK    ; HANDLE NORMAL TOKENS
      LDX #$ØØ   ; SET INPUT INDEX
      LDY #$Ø4   ; SET OUTPUT INDEX
      STY $ØF    ; FLAG FOR DATA TOKEN
CHARLP LDA $Ø2ØØ,X ; NEXT CHAR FROM BUFFER
      CMP #$8Ø   ;  IF ALREADY A TOKEN
      BCS STORE ;  THEN SKIP
      CMP #$2Ø   ; TEST FOR SPACE
      BEQ STORE
      STA $Ø8    ; SAVE FOR POSSIBLE QUOTE
      CMP #$22   ;  IS IT QUOTE?
      BEQ QUOTE
      BIT $ØF    ;  SEE IF INSIDE DATA
      BVS STORE ;  YES - BRANCH
      CMP #$3Ø   ;  SEE IF NUMERIC
      BCC NOTNUM
      CMP #$3C   ;  BRANCH IF NUMBER ; :
      BCC STORE
NOTNUM LDA ADRTOK+1
      STA $FD
      STA $FF
      STY $71    ; SAVE OUTPUT INDEX
      LDA ADRTOK ; SET POINTERS
      TAY         ; TO TOKEN TABLE
      DEY         ; IN ($FC)
      STY $FE     ; AND ($FE)
      DEY
      STY $FC
      LDY #$Ø1
      STY $ØB    ; INIT TOKEN INDEX
      DEY
      STX $7A    ; SAVE INPUT INDEX
      DEX
NEXTUP INY         ; NEXT CHARACTER IN TABLE
      BNE NOTINC
      INC $FF    ; MOVE TO NEXT TABLE
      INC $FD    ; PAGE AFTER 256 BYTES
NOTINC INX         ; NEXT CHARACTER IN BUFFER
NOTEND LDA $Ø2ØØ,X
      SEC
      SBC ($FE),Y
      BEQ NEXTUP ; IF BUFFER AND TABLE MATCH
      CMP #$8Ø   ; IF ONLY HIGH BIT OFF
      BNE NOMTCH ; NO MATCH
      LDA #$6Ø   ; TWO-BYTE TOKEN-1ST BYTE FIXED
      LDY $71    ; OUTPUT INDEX
      INY
      STA $Ø1FB,Y ; STORE THE $6Ø TOKEN
      INC $71
      LDA $ØB    ; 2ND BYTE TOKEN
REPEET LDY $71     ; RESET OUTPUT INDEX
STORE  INX         ; INPUT  INDEX
      INY        ; OUTPUT INDEX
      STA $Ø1FB,Y ; STORE CHAR
      LDA $Ø1FB,Y
      BEQ DONE   ;IF END OF BUFFER
      SEC
      SBC #$3A   ; SEE IF COLON
      BEQ COLON
      CMP #$49   ; SEE IF "DATA" TOKEN
      BNE NOTDAT
COLON  STA $ØF     ; SET DATA FLAG TO $49 FOR BVS
      BNE CHARLP
NOTDAT SEC
      SBC #$55   ; SEE IF REM
      BNE CHARLP
      STA $Ø8    ; IF REM SET $ØØ TERMINATOR
TSTEND LDA $Ø2ØØ,X   ; NEXT CHAR FROM INPUT
      BEQ STORE
      CMP $Ø8   ; =  TERMINATOR ?
      BEQ STORE
;
;
;
QUOTE  INY         ; OUTPUT INDEX
      STA $Ø1FB,Y
      INX
      BNE TSTEND ; UNCONDITIONAL
NOMTCH LDX $7A
      INC $ØB    ; TOKEN COUNT
TOKADV INY
```

```
      BNE NOTOVR
      INC $FF    ; INCREMENT TOKEN
      INC $FD    ;  TABLE PAGE
NOTOVR LDA ($FC),Y
      BPL TOKADV ;  LOOP UNTIL END-OF-TOKEN BYTE
      LDA ($FE),Y
      BNE NOTEND   ; FALL THROUGH IF $ØØ END
      LDA $Ø2ØØ,X   ; END-OF-TABLE
      BPL REPEET    ; TRY NEXT CHAR IN BUFFER
DONE   STA $Ø1FD,Y
      LDA #$Ø1
      STA $7B
      LDA #$FF
      STA $7A
      RTS
;
;
;
;
;     LIST FRONT-END TO
;     HANDLE LISTING OF
;     NEW KEYWORDS THROUGH
;     DETOKENIZATION
;
;
;
LISTX  BMI ONETOK ; ONE BYTE TOKEN
      CMP #$6Ø    ; DOUBLE TOKEN?
      BEQ LSTTOK ; YES - BRANCH
      BNE PRTONE ; NOT A TOKEN
ONETOK BIT $ØF    ; IN QUOTES?
      BMI PRTONE
TSTNLS CMP #$FF    ; PI?
      BEQ PRTONE ; NO - DO NEW TOKENS
      JMP NLST    ; NO - DO TOKEN
LSTTOK LDA ADRTOK+1 ; SET POINTER
      STA $FF    ; TO TOKEN TABLE
      DEC $FF
      LDA ADRTOK ; IN (FE)
      STA $FE
      INY         ; GET 2ND BYTE OF TOKEN
      LDA ($5F),Y
      STY $49   ; SAVE INDEX INTO LINE
      TAX
      LDY #$FF    ; SET INDEX INTO KEYWORD TABLE
LLOOP1 DEX         ; IF X=Ø THEN
      BEQ MATCH   ; KEYWORD MATCHES TOKEN
LLOOP2 INY
      BNE CONT1
      INC $FF
CONT1  LDA ($FE),Y   ; NEXT KEYWORD CHAR
      BPL LLOOP2    ; SAME KEYWORD
      BMI LLOOP1    ; END OF KEYWORD
MATCH  INY
      BNE CONT2
      INC $FF
CONT2  LDA ($FE),Y   ; GET KEYWORD CHAR
      BMI ENDTOK    ; LAST CHAR OF KEYWORD
      JSR OCHR      ; OUTPUT ALL CHARS EXCEPT LAST
      BNE MATCH     ; UNCONDITIONAL
ENDTOK JMP LRET1     ; BACK TO LIST
PRTONE JMP LRET2     ; BACK TO LIST
;
;
;   NEW COMMAND EXECUTION
;
;
COMX   JSR $ØØ73    ; CHRGET
      PHP
      CMP #$6Ø     ; TWO-BYTE TOKEN?
      BNE NRMXEQ
      LDA #$EA   ; REMOVE CHECK FOR
      STA $82    ; SPACE IN CHRGET
      STA $83    ; DUE TO $6Ø2Ø TOKEN
      JSR $ØØ73    ; GET SECOND TOKEN
      PHA
      LDA #$FØ   ; RESTORE CHECK
      STA $82    ; FOR SPACE
      LDA #$EF   ; IN CHRGET
      STA $83
      PLA
      CMP FNCONE ; TOKEN > COMMANDS ?
      BCS NRMXEQ  ; IF YES EXIT
      PLP         ; IF NOT
      JSR CMDNEW  ; DO THE COMMAND
      JMP NSTT    ; PREPARE FOR NEXT
;                   BASIC STATEMENT
;
CMDNEW SEC
      SBC #$Ø1
      ASL A
      TAY
      LDA COMADR+1,Y
      PHA
      LDA COMADR,Y
      PHA
      JMP $ØØ73
;
```

```
;
NRMXEQ PLP
       JMP NCMD     ; NORMAL COMMANDS
;
;
; NEW ERROR MESSAGE HANDLER
;
;  VIC VERSION DOES NOT NEED THE
;   BMI RDYMSG AND THE JMP READY
;
ERRX    TXA
       BMI RDYMSG
       CPX #$1F   ; NEW ERROR MESG?
       BCS NEWERR
       JMP NERR   ; NO
RDYMSG JMP READY
NEWERR TXA
       SEC
       SBC #$1F   ; GET INDEX INTO
       ASL A      ; NEW ERROR ADDRESS
       TAX        ; TABLE
       LDA ERRTAB,X
       STA $22
       LDA ERRTAB+1,X
       STA $23
       JMP ERRCNT ; CONTINUE WITH
;                   NORMAL ERROR
;                   MESSAGE HANDLER
;
;
;
;   NEW FUNCTION EXECUTION
;
FUNCX  LDA #$ØØ
       STA $ØD    ; INDICATE NUMERIC RESULT
       JSR $ØØ73
       PHP
       CMP #$6Ø   ; SEE IF TWO BYTE TOKEN
       BNE NRMFNC ; IF NOT-NORMAL FUNCTION
       JSR $ØØ73 ; GET 2ND BYTE
       CMP FNCONE ; CHECK FOR
       BCC ERRFNC ; VALID RANGE
       CMP ENDFNC ; FOR 2ND BYTE
       BCS ERRFNC
       PLP
       PHA
       CMP TWOARG ;SEE IF ONE-ARG FUNCTION
       BCS MLTARG ; NO - BRANCH
       JSR $ØØ73  ; CHRGET THE "("
      JSR EVALP  ; EVAL EXPRESSION IN PARENS
MLTARG PLA        ; GET INDEX INTO
      SEC        ; FUNCTION
      SBC FNCONE ; ADDRESS TABLE
      ASL A
      TAY
      LDA FNCADR,Y   ; SET UP TO
      STA $55        ; EXECUTE LIKE
      LDA FNCADR+1,Y  ; NORMAL FUNCTION
      STA $56
      JSR $ØØ54
      JMP NCHK   ; CHECK FOR NUMERIC RESULT
;
;
NRMFNC PLP
      JMP NFNC    ; NORMAL FUNCTIONS
ERRFNC JMP SYNTAX ; IF 2ND BYTE NOT VALID
;
;
;
; SCREEN (C-64 VERSION)
; SCREEN (C-64 VERSION)
;
SCREEN JSR EVAL  ; EVAL EXPRESSION
      JSR F1TX   ; CONVERT FAC1 TO X-REG Ø-255
      CPX #$1Ø ; VALID COLOR?
      BCS SYNER1
      STX SREG
      RTS
SYNER1 JMP SYNTAX; SYNTAX ERROR
;
; SCREEN (VIC-2Ø VERSION)
;
;SCREEN JSR EVAL ; EVAL EXPRESSION
;       JSR F1TX  ; CONVERT TO X-REG Ø-255
;       CPX #$1Ø  ; VALID COLOR?
;       BCS SYNV1 ; NO
;       LDA SREG
;       AND #$ØF
;       STA SREG
;       TXA
;       ASL A
;       ASL A
;       ASL A
;       ASL A
;       ORA SREG
;       STA SREG
;       RTS
;SYNV1 JMP SYNTAX
```

```
;
; COLD
;
COLD   JMP CSTRT
;
; BRDR - SET BORDER COLOR (C-64)
;
BRDR   JSR EVAL    ; EVAL EXPRESSION
      JSR F1TX   ; CONVERT TO X-REG Ø-255
      CPX #$1Ø
      BCS SYNER2
      STX BREG
      RTS
SYNER2 JMP SYNTAX
;
; BRDR - SET BORDER COLOR (VIC-2Ø)
;
;BRDR JSR EVAL ; EVAL EXPRESSION
;     JSR F1TX ; CONVERT X-REG TO Ø-255
;     CPX #$Ø8 ; VALID BORDER COLOR ?
;     BCS SYVER2
;     LDA BREG
;     AND #$F8
;     STA BREG
;     TXA
;     ORA BREG
;     STA BREG
;     RTS
;SYVER2 JMP SYNTAX
;
; VBACK  - RESET VECTORS TO NORMAL SETTING
;
VBACK  SEI
      JSR RVECT  ; INIT VECTORS
      CLI
      RTS
;
;
;    KEYS
;         A - ALL TYPOMATIC
;         N - NONE TYPOMATIC
;         S - NORMAL TYPOMATIC
;
KEYS   CMP #$41  ; IS IT "A"?
      BNE NOTA
      LDA #$8Ø
      BNE COMKEY
NOTA   CMP #$4E  ; IS IT "N"?
      BNE NOTN
```

```
       LDA #$4Ø
       BNE COMKEY
NOTN    CMP #$53   ; IS IT "S"?
       BNE SYNER3
       LDA #$ØØ
COMKEY STA KREG
       JSR $ØØ73 ; CHRGET END-OF-STATEMENT
       RTS
;
SYNER3 LDX #$1F   ; SET ERROR NUMBER
       JMP ($Ø3ØØ) ; ERROR VECTOR
;
; FRAC - RETURN FRACTIONAL PART
;        OF NUMBER
;
FRAC    JSR F1T57 ; COPY FAC1 TO ØØ57
       JSR INTF1 ; FAC1 = INT(FAC1)
       LDA #$57
       LDY #$ØØ
       JSR SUBF1 ; FAC1=ØØ57-FAC1
       RTS
;
;
; DIV - RETURN INTEGER QUOTIENT
;
;
;
DIV     JSR $ØØ73 ; CHRGET THE "("
       JSR CHKLP ; CHECK FOR "("
       JSR EVAL  ; EVAL 1ST EXPRESSION
       JSR F1T57 ; COPY FAC1 TO ØØ57
       JSR CHKCM ; CHECK FOR COMMA
       JSR EVAL  ; EVAL 2ND EXPRESSION
       JSR CHKRP ; CHECK FOR ")"
       LDA #$57
       LDY #$ØØ
       JSR F1DIV ; FAC1=ØØ57/FAC1
       JSR INTF1 ; FAC1=INT(FAC1)
       RTS
;
;
;   COMMANDS AND FUNCTIONS NOT
;    YET IMPLEMENTED
;
;
ADUMP   RTS
APPND   RTS
AUTON   RTS
BOX     RTS
```

```
CASE    RTS
CHANGE  RTS
CIRCLE  RTS
COLLID  RTS
COMGT   RTS
COPY    RTS
DEL     RTS
DLAY    RTS
DRAW    RTS
DUMP    RTS
ERASE   RTS
FIND    RTS
GCHANG  RTS
GFIND   RTS
HELP    RTS
HIMEM   RTS
HIRES   RTS
HPEN    RTS
HTAB    RTS
INSERT  RTS
LABEL   RTS
LINE    RTS
LOMEM   RTS
MERGE   RTS
MOD     RTS
MOVE    RTS
TAPEM   RTS
MULTIC  RTS
OLD     RTS
PAINTR  RTS
PFKEY   RTS
POINTR  RTS
POP     RTS
PROC    RTS
PRTUSN  RTS
RENUM   RTS
REPEAT  RTS
RLST    RTS
SET     RTS
SINGLE  RTS
SPRITC  RTS
SPRITD  RTS
SPRITM  RTS
SRT     RTS
TRACE   RTS
UNLST   RTS
UNNW    RTS
VDUMP   RTS
VPEN    RTS
```

```
VTAB   RTS
WHILE  RTS
JOY    RTS
PENX   RTS
PENY   RTS
;
;    ALLOW EMBEDDED KEYWORDS
;
;
;
; ROUTINE USED WHEN COMMODORE 64
; TOKENIZATION ROUTINE IS MODIFIED
; TO CHECK FOR A FOLLOWING SPACE
; , COLON, OR $00 END-OF-LINE BYTE
; BEFORE A KEYWORD IS TOKENIZED
; TO ALLOW THE NEW KEYWORDS TO
; CONTAIN EMBEDDED OLD KEYWORDS
; (EXCEPT AT THE END OF A NEW
; KEYWORD)
;
;  IF THIS FEATURE IS USED THEN
;  YOU MUST FOLLOW ALL KEYWORDS
;  WITH A SPACE
;
;
;  BASIC PROGRAM TO MODIFY THE C-64
;  TOKENIZATION ROUTINE IS:
;  10 FOR X=40960 TO 49151
;  20 Y=PEEK(X)
;  30 POKE X,Y
;  40 NEXT
;  50 POKE 42433,76
;  60 POKE 42434,105
;  70 POKE 42435,196
;  80 POKE 1,PEEK(1) AND 254
;
;
;
MODTOK CMP #$80
       BNE POSIN
       PHA        ; IF POSSIBLE TOKEN
       LDA $0201,X  ; GET NEXT CHAR
       CMP #$20   ; IS IT A SPACE?
       BEQ ENDW
       CMP #$00   ; END-OF-LINE?
       BEQ ENDW
       CMP #$3A   ; ":" ?
       BEQ ENDW
       PLA
```

```
POSIN   JMP $A5F5   ; DON'T TOKENIZE
ENDW    PLA
      JMP $A5C5 ; DO TOKENIZE
```

New BASIC Commands—One-Byte Tokens

```
;**********************************
;
;
;     NEW COMMANDS FOR BASIC
;
;
;     ONE-BYTE TOKEN VERSION
;
;     ALLOWING NEW TOKENS $CC - $CE
;
;
;
;     COMMODORE 64 VERSION
;
;
;
;*********************************
;
; EQUATES FOR THE COMMODORE-64
;
;
NTOK = $A57C
NLST = $A724
OCHR = $AB47
LRET1 = $A6EF
LRET2 = $A6F3
NSTT = $A7AE
NCMD = $A7E7
EVALP = $AEF1
NCHK = $AD8D
NFNC = $AE8D
EVAL = $AD9E
F1TX = $B7A1
SYNTAX = $AFØ8
BREG = $DØ2Ø
SREG = $DØ21
KREG = $Ø28A
CSTRT = $E394
RVECT = $E453
F1T57 = $BBCA
INTF1 = $BCCC
SUBF1 = $B85Ø
CHKLP = $AEFA
```

```
CHKCM = $AEFD
CHKRP = $AEF7
F1DIV = $BBØF
;
        * = $CØØØ
;
; SET UP VECTORS TO NEW ROUTINES
;
; SYS 49152  TO CHANGE VECTORS
;
;
      SEI
      LDA TOKV
      STA $Ø3Ø4
      LDA TOKV+1
      STA $Ø3Ø5
      LDA LSTV
      STA $Ø3Ø6
      LDA LSTV+1
      STA $Ø3Ø7
      LDA COMV
      STA $Ø3Ø8
      LDA COMV+1
      STA $Ø3Ø9
      LDA FNCV
      STA $Ø3ØA
      LDA FNCV+1
      STA $Ø3ØB
      CLI
      RTS
;
;
;
;
TOKV   .WORD  NEWTOK
LSTV   .WORD  NEWLST
COMV   .WORD  NEWCOM
FNCV   .WORD  NEWFNC
;
;
;NO ABBREVIATIONS(WITH SHIFTED KEY)
; PERMITTED BECAUSE NORMAL TOKENIZATION
; THROWS AWAY BYTES >= #$8Ø
;
; NOTE:  AS CURRENTLY CONSTRUCTED
;        THE PROGRAM ALLOWS A
;  MAXIMUM NUMBER OF CHARACTERS
;  IN TOKEN TABLE OF 255 SINCE
;  8-BIT REGISTER USED TO INDEX
;    THROUGH TABLE
;
TOKTAB .BYTE 'COL'
      .BYTE $C4     ; COLD   $CC
      .BYTE 'BRD'
      .BYTE $D2     ; BRDR   $CE
      .BYTE 'SCREE'
      .BYTE $CE     ; SCREEN   $CD
      .BYTE 'VBAC'
      .BYTE $CB     ; VBACK   $CF
      .BYTE 'KEY'
      .BYTE $D3     ; KEYS $DØ
      .BYTE 'FRA'
      .BYTE $C3     ; FRAC   $D1
      .BYTE 'DI'
      .BYTE $D6     ; DIV    $D2
      .BYTE $ØØ
;
; EQUATES SPECIFY WHERE COMMANDS END,
; FUNCTIONS BEGIN AND END, AND WHICH
; FUNCTIONS ONLY HAVE ONE ARGUMENT
;
FNCONE .BYTE $D1
TWOARG .BYTE $D2
ENDFNC .BYTE $D3
;
; COMMAND ADDRESS TABLE
;
COMADR  .WORD COLD-1
       .WORD BRDR-1
       .WORD SCREEN-1
       .WORD VBACK-1
       .WORD KEYS-1
;
;
; FUNCTION ADDRESS TABLE
;
;
FNCADR  .WORD FRAC
       .WORD DIV
;
;
;  TOKENIZATION ROUTINE FOR
;     NEW COMMANDS
;
;
;
;
;
```

```
;
NEWTOK JSR NTOK    ; HANDLE NORMAL TOKENS
       LDX #$ØØ   ; SET INPUT INDEX
       LDY #$Ø4   ; SET OUTPUT INDEX
       STY $ØF    ; FLAG FOR DATA TOKEN
CHARLP LDA $Ø2ØØ,X ; NEXT CHAR FROM BUFFER
       CMP #$8Ø   ;  IF ALREADY A TOKEN
       BCS STORE
       CMP #$2Ø   ; TEST FOR SPACE
       BEQ STORE
       STA $Ø8    ; SAVE FOR POSSIBLE QUOTE
       CMP #$22   ;  IS IT QUOTE?
       BEQ QUOTE
       BIT $ØF    ;  SEE IF INSIDE DATA
       BVS STORE ;  YES - BRANCH
       CMP #$3Ø   ;  SEE IF NUMERIC
       BCC NOTNUM
       CMP #$3C   ;  BRANCH IF NUMBER ; :
       BCC STORE
NOTNUM STY $71     ; SAVE OUTPUT INDEX
       LDY #$ØØ
       STY $ØB    ; INIT TOKEN INDEX
       DEY
       STX $7A    ; SAVE INPUT INDEX
       DEX
NEXTUP INY         ; NEXT CHARACTER IN TABLE
       INX        ; NEXT CHARACTER IN BUFFER
NOTEND LDA $Ø2ØØ,X
       SEC
       SBC TOKTAB,Y
       BEQ NEXTUP
       CMP #$8Ø   ; IF ONLY HIGH BIT OFF
       BNE NOMTCH ; NO MATCH
       LDA $ØB    ; TOKEN VALUE
       CLC
       ADC #$CC   ; ADD START OF EXPANDED TOKENS
REPEAT LDY $71     ; RESET OUTPUT INDEX
STORE  INX         ; INPUT  INDEX
       INY        ; OUTPUT INDEX
       STA $Ø1FB,Y ; STORE CHAR
       LDA $Ø1FB,Y
       BEQ DONE   ; IF END OF BUFFER
       SEC
       SBC #$3A   ; SEE IF COLON
       BEQ COLON
       CMP #$49   ; SEE IF "DATA" TOKEN
       BNE NOTDAT
COLON  STA $ØF
       BNE CHARLP
```

```
NOTDAT SEC
       SBC #$55   ; SEE IF REM
       BNE CHARLP
       STA $Ø8    ; IF REM SET $ØØ TERMINATOR
TSTEND LDA $Ø2ØØ,X   ; NEXT CHAR FROM INPUT
       BEQ STORE
       CMP $Ø8   ; =  TERMINATOR ?
       BEQ STORE
;
;
;
QUOTE   INY        ; OUTPUT INDEX
       STA $Ø1FB,Y
       INX
       BNE TSTEND
NOMTCH LDX $7A
       INC $ØB   ; TOKEN COUNT
TOKADV  INY
       LDA TOKTAB-1,Y
       BPL TOKADV
       LDA TOKTAB,Y
       BNE NOTEND
       LDA $Ø2ØØ,X   ; END-OF-TABLE
       BPL REPEAT
DONE    STA $Ø1FD,Y
       LDA #$Ø1
       STA $7B
       LDA #$FF
       STA $7A
       RTS
;
;
;
;
;     LIST FRONT-END TO
;     HANDLE LISTING OF
;     NEW KEYWORDS THROUGH
;     DETOKENIZATION
;
;
;
;
NEWLST BPL PRTONE ; CAN'T BE A TOKEN IF
;                   HIGH ORDER BIT IS Ø
       CMP #$FF    ; SEE IF PI
       BEQ PRTONE
       BIT $ØF     ; IN QUOTES?
       BMI PRTONE
       CMP #$CC    ; NEW TOKEN?
```

```
      BCS LSTTOK ; YES IF >= $CC
      JMP NLST   ; NO - DO NORMAL LIST
;                  DETOKENIZATION
LSTTOK SEC
      SBC #$CB   ; #$CC = 1, #CD = 2, ETC.
      TAX
      STY $49    ; SAVE INDEX INTO
;                  LINE BEING LISTED
      LDY #$FF   ; SET INDEX INTO KEYWORD TABLE
LLOOP1 DEX        ; IF X=Ø THEN
      BEQ MATCH  ; KEYWORD MATCHES TOKEN
LLOOP2 INY
      LDA TOKTAB,Y ; NEXT KEYWORD CHAR
      BPL LLOOP2   ; SAME KEYWORD
      BMI LLOOP1   ; END OF KEYWORD
MATCH  INY
      LDA TOKTAB,Y ; GET KEYWORD CHAR
      BMI ENDTOK   ; LAST CHAR OF KEYWORD
      JSR OCHR     ; OUTPUT ALL CHARS EXCEPT LAST
      BNE MATCH    ; UNCONDITIONAL
ENDTOK JMP LRET1    ; BACK TO LIST
PRTONE JMP LRET2    ; BACK TO LIST
;
;
;   NEW COMMAND EXECUTION
;
;
NEWCOM JSR $ØØ73   ; CHRGET
      PHP
      CMP #$CC    ; LESS THAN COMMANDS?
      BCC NRMXEQ
      CMP FNCONE  ; GREATER THAN COMMANDS?
      BCS NRMXEQ
      PLP
      JSR CMDNEW
      JMP NSTT
;
;
CMDNEW SEC
      SBC #$CC
      ASL A
      TAY
      LDA COMADR+1,Y
      PHA
      LDA COMADR,Y
      PHA
      JMP $ØØ73
;
;
```

```
NRMXEQ PLP
       JMP NCMD     ; NORMAL COMMANDS
;
;
;
;
;
;    NEW FUNCTION EXECUTION
;
;
NEWFNC LDA #$ØØ
       STA $ØD
       JSR $ØØ73
       PHP
       CMP FNCONE ; VALID FUNCTION
       BCC NRMFNC ; TOKEN ?
       CMP ENDFNC
       BCS NRMFNC
       PLP
       PHA
       CMP TWOARG ;SEE IF ONE-ARG FUNCTION
       BCS MLTARG ; NO - BRANCH
       JSR $ØØ73  ; CHRGET THE "("
       JSR EVALP  ; EVAL EXPRESSION IN PARENS
MLTARG PLA
       SEC
       SBC FNCONE
       ASL A
       TAY
       LDA FNCADR,Y
       STA $55
       LDA FNCADR+1,Y
       STA $56
       JSR $ØØ54
       JMP NCHK   ; CHECK FOR NUMERIC RESULT
;
;
NRMFNC PLP
       JMP NFNC
;
;
;
; SCREEN
;
SCREEN JSR EVAL   ; EVAL EXPRESSION
       JSR F1TX   ; CONVERT FAC1 TO X-REG Ø-255
       CPX #$1Ø ; VALID COLOR?
       BCS SYNER1
       STX SREG
```

```
        RTS
SYNER1 JMP SYNTAX; SYNTAX ERROR
;
;
; COLD
;
COLD    JMP CSTRT
;
; BRDR - SET BORDER COLOR
;
BRDR    JSR EVAL    ; EVAL EXPRESSION
        JSR $B7A1 ; CONVERT TO X-REG Ø-255
        CPX #$1Ø
        BCS SYNER2
        STX BREG
        RTS
SYNER2 JMP SYNTAX; SYNTAX ERROR
;
;
; VBACK  - RESET VECTORS
;
VBACK  SEI
        JSR RVECT
        CLI
        RTS
;
;
;     KEYS
;          A - ALL TYPOMATIC
;          N - NONE TYPOMATIC
;          S - NORMAL TYPOMATIC
;
KEYS    CMP #$41   ; IS IT "A"?
        BNE NOTA
        LDA #$8Ø
        BNE COMKEY
NOTA    CMP #$4E   ; IS IT "N"?
        BNE NOTN
        LDA #$4Ø
        BNE COMKEY
NOTN    CMP #$53   ; IS IT "S"?
        BNE SYNER3
        LDA #$ØØ
COMKEY STA KREG
        JSR $ØØ73 ; CHRGET END-OF-STATEMENT
        RTS
;
SYNER3 JMP SYNTAX
;
```

```
; FRAC - RETURN FRACTIONAL PART
;          OF NUMBER
;
FRAC    JSR F1T57 ; COPY FAC1 TO 0057
       JSR INTF1; FAC1 = INT(FAC1)
       LDA #$57
       LDY #$00
       JSR SUBF1 ; FAC1=0057-FAC1
       RTS
;
;  DIV - INTEGER QUOTIENT DIVISION
;
;
DIV     JSR $0073 ; CHRGET THE "("
       JSR CHKLP ; CHECK FOR "("
       JSR EVAL  ; EVAL 1ST EXPRESSION
       JSR F1T57 ; COPY FAC1 TO 0057
       JSR CHKCM ; CHECK FOR COMMA
       JSR EVAL  ; EVAL 2ND EXPRESSION
       JSR CHKRP ; CHECK FOR ")"
       LDA #$57
       LDY #$00
       JSR F1DIV ; FAC1=0057/FAC1
       JSR INTF1 ; FAC1=INT(FAC1)
       RTS
```

New BASIC Commands—NMI

```
;***********************************
;
;
;    NEW COMMANDS FOR BASIC
;
;
;    USING RESTORE KEY
;
;    AND NMI INTERRUPT WEDGE
;
;
;
;    COMMODORE 64 VERSION
;
;
;
;**********************************
;
; EQUATES FOR THE COMMODORE-64
;
;
```

```
GETIN = $FFE4
BREAK  = $FE66
BREG = $DØ2Ø
SREG = $DØ21
KREG = $Ø28A
CSTRT = $E394
OUTNMI = $FEBC
;
        * = $CØØØ
;
; SET UP VECTORS TO NEW ROUTINES
;
; SYS 49152  TO CHANGE VECTORS
;
;
      LDA #$ØØ     ; NO RECURSION
      STA RECURS   ; INITIALLY
      SEI
      LDA NMIV
      STA $Ø318    ; (Ø318) RESET
      LDA NMIV+1
      STA $Ø319
      CLI
      RTS
;
;
;
;
NMIV   .WORD  NEWNMI
;
;
RECURS .BYTE $ØØ
;
;
;
;     NEW NMI INTERRUPT HANDLER
;
;
NEWNMI PHA  ; SAVE ACCUM, X-REG,
      TXA  ;  AND Y-REG
      PHA  ; AS IN NORMAL NMI
      TYA
      PHA
      LDA RECURS   ; CHECK FOR
      BEQ NOTAGN   ; RECURSIVE
      DEC RECURS   ; ENTRY
      JMP OUTNMI
NOTAGN INC RECURS
      JSR $F6BC   ; KEYBOARD SCAN
```

```
;               AND JIFFY UPDATE
      JSR $FFE1  ; TEST FOR STOP KEY
      BNE NOTSTP
      JMP BREAK  ; IF STOP KEY
;
; IF STOP KEY NOT DEPRESSED
;  THEN PERFORM OWN NMI CODE
;
NOTSTP CLI
KEYLP  JSR $FFE4  ; KERNAL GETIN
      CMP #$00   ; ANY KEY?
      BEQ KEYLP  ; NO - TRY AGAIN
      SEI
      CMP #$0D  ; RETURN KEY?
      BEQ SKIP
      CMP #$30  ; LESS THAN 0?
      BCC SKIP
      CMP #$5B  ; GREATER THAN Z?
      BCS SKIP
      JSR DOKEY
SKIP   JMP OUTNMI
;
;
;  GET ADDRESS OF NEW COMMAND
;  AND GO TO IT
;
DOKEY  SEC
      SBC #$30
      ASL A
      TAY
      LDA COMADR+1,Y
      PHA
      LDA COMADR,Y
      PHA
      RTS
;
;
;
;
;
; ENTRY FOR UNDEFINED KEYS
;
;
NULL   RTS
;
;
;   ADDRESS TABLE FOR KEY COMMANDS
;                         KEY
;
```

```
COMADR .WORD NULL-1   ; ØØ   Ø
       .WORD NULL-1   ; Ø1   1
       .WORD NULL-1   ; Ø2   2
       .WORD NULL-1   ; Ø3   3
       .WORD NULL-1   ; Ø4   4
       .WORD NULL-1   ; Ø5   5
       .WORD NULL-1   ; Ø6   6
       .WORD NULL-1   ; Ø7   7
       .WORD NULL-1   ; Ø8   8
       .WORD NULL-1   ; Ø9   9
       .WORD NULL-1   ; ØA   :
       .WORD NULL-1   ; ØB   ;
       .WORD NULL-1   ; ØC   <
       .WORD NULL-1   ; ØD   =
       .WORD NULL-1   ; ØE   >
       .WORD NULL-1   ; ØF   ?
       .WORD NULL-1   ; 1Ø   @
       .WORD ALL-1    ; 11   A
       .WORD BRDR-1   ; 12   B
       .WORD COLD-1   ; 13   C
       .WORD NULL-1   ; 14   D
       .WORD NULL-1   ; 15   E
       .WORD NULL-1   ; 16   F
       .WORD NULL-1   ; 17   G
       .WORD NULL-1   ; 18   H
       .WORD NULL-1   ; 19   I
       .WORD NULL-1   ; 1A   J
       .WORD NULL-1   ; 1B   K
       .WORD NULL-1   ; 1C   L
       .WORD NULL-1   ; 1D   M
       .WORD NONE-1   ; 1E   N
       .WORD NULL-1   ; 1F   O
       .WORD NULL-1   ; 2Ø   P
       .WORD NULL-1   ; 21   Q
       .WORD REG-1    ; 22   R
       .WORD SCREEN-1 ;23    S
       .WORD NULL-1   ; 24   T
       .WORD NULL-1   ; 25   U
       .WORD VBACK-1  ; 26   V
       .WORD NULL-1   ; 27   W
       .WORD NULL-1   ; 28   X
       .WORD NULL-1   ; 29   Y
       .WORD NULL-1   ; 2A   Z
;
;
;
;
; SCREEN  - CHANGE SCREEN COLOR
;
```

```
SCREEN INC SCRN
       LDA SCRN
       CMP #$11
       BCC OKSK
       LDA #$ØØ
       STA SCRN
OKSK    STA SREG
       RTS
SCRN    .BYTE $ØØ
;
;
; COLD
;
COLD    JMP CSTRT
;
; BRDR - SET BORDER COLOR
;
BRDR    INC BORD
       LDA BORD
       CMP #$11
       BCC OKBR
       LDA #$ØØ
       STA BORD
OKBR    STA BREG
       RTS
BORD    .BYTE $ØØ
;
; VBACK
;
VBACK  LDA #$47
       STA $Ø318
       LDA #$FE
       STA $Ø319
       CLI
       RTS
;
;
; ALL TYPOMATIC KEYS
;
ALL     LDA #$8Ø
       STA KREG
       RTS
;
;  REGULAR TYPOMATIC KEYS
;
REG     LDA #$ØØ
       STA KREG
       RTS
;
```

```
;  NO TYPOMATIC KEYS
;
NONE    LDA #$40
       STA KREG
NOKEY   RTS
```

Wedging into CHRGET

```
;*******************************
;
;
;  WEDGING INTO CHRGET
;
;*******************************
;
;
;  SYS 50176 TO ACTIVATE WEDGE
;   ON THE COMMODORE 64
;
;
      * = $C400
;
;
;  ON THE VIC-20 ONE POSSIBLE
;  ORIGIN IS $6200 IF YOU HAVE
;  ENOUGH MEMORY EXPANSION
;  THE SYS FOR $6200 IS SYS 25088
;
;  OF COURSE YOU CAN MAKE THE VIC
;   ORIGIN ANY RAM LOCATION AS
;   LONG AS YOU PROTECT IT FROM
;   BASIC
;
;        * = $6200  ; VIC ORIGIN
;
; *************************
;
; MODIFY CHRGET TO INITIALIZE WEDGE
;
;
INITWD  LDX #$02
NEXTI   LDA FIX,X
       STA $7C,X
       DEX
       BPL NEXTI
       RTS
FIX     JMP WEDGE
;
;
```

```
; *************************
;
;  ADDRESS (MINUS ONE) OF THE
;     WEDGE COMMANDS
;
;
COMADR   .WORD BRDR-1     ; FOR !
        .WORD SCREEN-1 ; FOR /
        .WORD SCREEN-1 ; FOR / TOKEN
        .WORD KEYS-1   ; FOR £
        .WORD QUIT-1   ; FOR #
NEWCMD   .BYTE '!/'
        .BYTE $AD      ; / TOKEN
        .BYTE '£#'
ENDCMD   .BYTE $Ø5     ; NUMBER OF COMMANDS
HOLDX    .BYTE $ØØ
HOLDA    .BYTE $ØØ
;
;
; EQUATES COMMON TO VIC-2Ø AND C-64
;
;
        CHRGET = $ØØ73
        CHRGOT = $ØØ79
        KREG   = $Ø28A
;
;
; EQUATES FOR COMMODORE-64
;
        EVALE = $AE86
        F1TX = $B7A1
        SYNTAX = $AFØ8
        BREG = $DØ2Ø
        SREG = $DØ21
        ROMGET = $E3AB
        READY = $A474
        EMSG = $A437
;
;
;  EQUATES FOR THE VIC-2Ø
;       EVALE = $CE86
;       F1TX = $D7A1
;       SYNTAX = $CFØ8
;       BREG = $9ØØF
;       SREG = $9ØØF
;       ROMGET = $E39Ø
;       READY = $C474
;       EMSG = $C437
;
```

```
;*****************************
;
; START OF THE WEDGE
;
WEDGE    STX HOLDX
       STA HOLDA  ;CHAR FROM CHRGET
       TSX
       LDA $Ø1Ø1,X ; RETURN ADDRESS LO BYTE
       CMP #$E6   ; CALL FROM $A7E6?
       BEQ CMDDIR ; YES - PROGRAM MODE
       LDA $Ø1Ø2,X
       CMP #$A4   ; CALL FROM $A48C?
       BNE NOTNEW  ; IF YES - DIRECT MODE
;                  IGNORE ALL OTHER
;                  CHRGET CALLS
;
;
; DETERMINE IF CHARACTER IS IN
;  THE COMMAND TABLE
;
;
CMDDIR  LDX #$ØØ
       LDA HOLDA     ; CHRGET CHARACTER
CHECKN  CMP NEWCMD,X  ; MATCH TABLE ENTRY?
       BEQ FOUND ; YES- X OFFSET INTO TABLE
       INX
       CPX ENDCMD
       BNE CHECKN
;
;  IF NOT A NEW COMMAND THEN RESTORE
;  CHARACTER RETRIEVED BY CHRGET
;  AND SET FLAGS BY JUMPING TO
;  REMAINDER OF CHRGET IN ROM COPY
;
;
NOTNEW  LDX HOLDX
       LDA HOLDA   ; RESTORE CHRGET CHARACTER
       JMP ROMGET  ; CONTINUE WITH CHRGET
;
;
; IF COMMAND FOUND THEN PUSH ADDRESS
; OF COMMAND ONTO STACK AND RTS TO
;  THE COMMAND CODE
;
;
FOUND    TXA
       ASL A
       TAY
       LDA COMADR+1,Y
       PHA
       LDA COMADR,Y
       PHA
       RTS
;
;
;
;  ! COMMAND TO SET THE BORDER COLOR
;
;
BRDR    JSR EVALE ;EVAL EXPRESSION
       JSR F1TX ; CONVERT X-REG Ø-255
       CPX #$1Ø
       BCS SYNER2
       STX BREG
       JMP CHRGOT
SYNER2  JMP SYNTAX
;
;
; ! BORDER COMMAND FOR VIC
;
;BRDR   JSR EVALE
;       JSR F1TX
;       CPX #$Ø8
;       BCS SYNER2
;       LDA BREG
;       AND #$F8
;       STA BREG
;       TXA
;       ORA BREG
;       STA BREG
;       RTS
;SYNER2 JMP SYNTAX
;
;
;
; / COMMAND TO SET THE SCREEN COLOR
;
;
;
SCREEN  JSR EVALE ; EVAL EXPRESSION
       JSR F1TX  ; CONVERT X-REG Ø-255
       CPX #$1Ø
       BCS SYNER1
       STX SREG
       JMP CHRGOT
SYNER1  JMP SYNTAX
;
;
```

```
; / COMMAND TO SET SCREEN COLOR
;     VIC VERSION
;
;SCREEN   JSR EVALE
;         JSR F1TX
;         CPX #$1Ø
;         BCS SYNER1
;         LDA SREG
;         AND #$ØF
;         STA SREG
;         TXA
;         ASL A
;         ASL A
;         ASL A
;         ASL A
;         ORA SREG
;         STA SREG
;        RTS
;SYNER1 JMP SYNTAX
;
;
;
;
;
; £ COMMAND TO SET REPEATING KEYS
;
;  FOR DEMONSTRATION ONLY THIS
; COMMAND IS CODED TO PREVENT ENTRY
;  OF THE £ COMMAND FROM DIRECT MODE
;
;  A - ALL KEYS TYPOMATIC
;  N - NO KEYS TYPOMATIC
;  S - REGULAR TYPOMATIC KEYS
;
;
;
;
KEYS     JSR CHRGET
       LDX $7B    ; SEE IF DIRECT MODE
       CPX #$Ø2
       BNE NOTDIR
       LDX #$15   ; ILLEGAL DIRECT MSG
       JMP EMSG
NOTDIR  CMP #$41
       BNE NOTA
       LDA #$8Ø
       BNE COMKEY
NOTA     CMP #$4E
       BNE NOTIN
```

```
         LDA #$40
         BNE COMKEY
NOTIN     CMP #$53
         BNE SYNER3
         LDA #$00
COMKEY    STA KREG
         JMP CHRGET
SYNER3    JMP SYNTAX
;
;
;
; # COMMAND TO REMOVE THE WEDGE
;       FROM CHRGET
;
;
QUIT      LDX #$02
NEXTQ     LDA ROMGET,X
         STA $7C,X
         DEX
         BPL NEXTQ
         JMP READY
;
;
```

Mixing BASIC and Machine Language

In order for a BASIC program and a machine language program to coexist in memory, each must stay in the area reserved for it. The area reserved for each can vary widely.

Where should you place your machine language programs? The areas above or below BASIC are safe territories for placing machine language programs. The top of BASIC is pointed to by (37), and the bottom of BASIC is pointed to by (2B). On the Commodore 64, the top of BASIC is normally 9FFF, so a good area on the Commodore 64 is C000–CFFF. On the VIC (you can also do this on the 64), you can reserve an area for BASIC by calling the Kernal MEMTOP or MEMBOT routines to modify the top or bottom of memory and then call the routine used during cold start that sets the pointers in (37) and (2B). This routine is located at E3BF/E3A4 (58303/58276 decimal). Or you can just cold start BASIC at E394/E378 (58260/58232 decimal). To call MEMTOP, SYS to 65433. To call MEMBOT, SYS to 65436. Since SYS can pass X-register and Y-register and the status register, that is how you pass the new setting to MEMBOT or MEMTOP. The following short program resets the unexpanded VIC-20 top of memory, changing the BYTES FREE from 3583 to 2559.

POKE 783,0	**Clears the carry**
POKE 781,0	**Sets the X-register value to $00**
POKE 782,26	**Sets the Y-register value to $1A**
SYS 65433	**Go to MEMTOP**
SYS 58232	**Cold start BASIC**

Another area to place your machine language program is in the cassette tape buffer at 033C–03FC (actually, you can use through 03FF).

You can also embed machine language programs in a BASIC program. Often programmers put machine language programs in BASIC program DATA statements, and then have the BASIC program POKE the data values into memory. Another technique is to insert REMs into your BASIC program, followed by dummy characters that just occupy space in the tokenized program area. Then you can insert machine language statements into the dummy characters, being careful not to modify the line number, link address, REM token, or end-of-line—$00. It's easier if you include all your dummy REMs at the start of your BASIC program so that you don't have to worry about the REM line being moved in the BASIC text area.

Part Two

Detailed Descriptions

BASIC Initialization

BASIC is initialized through either a cold start or a warm start, depending on whether the Kernal uses the vector at A/C000 for the cold start or A/C002 for the warm start. A cold start occurs when the computer is powered on or when a reset occurs. A warm start occurs when the STOP and RESTORE keys are held down or when a BRK instruction occurs. If an autostart cartridge is activated, the cold or warm start passes control to the autostart cartridge rather than to BASIC. However, if a BRK instruction is executed while in BASIC (for example, POKE 1088,0:SYS 1088), control returns to BASIC's warm start even with an autostart cartridge in place.

A cold start performs several functions such as initializing page 0 variables; setting the pointers to the start of the tokenized program area, the top of available free space, and the end of BASIC memory; copying CHRGET to page 0; copying vectors to 0300–030B; displaying the initial power-up message; and executing the NEW routine. Any program that existed prior to a reset is no longer accessible through the normal BASIC pointers.

A warm start only resets the I/O channels, the stack pointer, and the string descriptor temporary stack pointer; disallows CONT; and enables IRQ interrupts. Normally, any BASIC program that was in memory before the warm start should still be there after the warm start, and the pointers to the program should still be valid.

If you are writing your own machine language programs to use BASIC features, you should consider doing a JSR to E453/E45B to create the vectors at 0300–030B and a JSR to E3BF/E3A4 to initialize CHRGET and various zero page pointers. If you are only interested in creating a copy of CHRGET in page 0, look at the instructions from E3E0/E3C2 to E3E8/E3CB. You could create your own subroutine by copying these instructions to a RAM location, then changing the initial value of X-register to $17 (the $1C includes copying the RND seed value following CHRGET), and putting an RTS at the end of your RAM routine.

Cold Start Routine
E394/E378–E3A1/E386

Whenever the system is powered on or reset, this routine is executed. The Commodore 64 and VIC-20 versions differ slightly—see step 5.

Called by:
JMP(A000)/(C000) at FCFF/FD3C system power-up routine.

Entry conditions:
System power-up routine has initialized the CIA/VIA registers, VIC Chip Registers, and the Kernal Vectors at 0314–0332.

Operation:
1. JSR E453/E45B to copy vectors to 0300–030B.
2. JSR E3BF/E3A4 to initialize certain zero page locations and to copy the CHRGET routine to page 0.
3. JSR E422/E404 to display:

   (For the Commodore 64)
   **** COMMODORE 64 BASIC V2 ****
   64K RAM SYSTEM 38911 BASIC BYTES FREE

   (For the VIC-20)
   **** CBM BASIC V2 ****
   xxxx BYTES FREE
   The xxxx is the end of BASIC memory minus the start of the tokenized BASIC program area. Then E422/E404 calls NEW.
4. Initialize the stack pointer to $FB.

5a. Commodore 64: Branch E386 to set the X-register to $80 then JMP(0300), where 0300 is the error message vector with a default of E38B. At E38B, if the X-register is no longer $80 (for example, if you have modified the vector at $0300 to go to your routine where the X-register is changed), JMP A43A to display the error message indexed by the X-register. Then fall through to A474 to display the READY message and continue at A480 with the Main BASIC Loop. At E38B, if the X-register is still $80, JMP A474.

5b. VIC: JMP C474 to display the READY message and then continue at C480 with the Main BASIC Loop.

Initialize Vectors E453/E45B-E45E/E466

Called by: JSR at E394/E378.

Exit conditions:
(0300) = E38B/C43A Error Message Vector.
(0302) = A/C483 Main BASIC Routine Vector.
(0304) = A/C57C Tokenization Routine Vector.
(0306) = A/C71A Expand and Print Tokens Routine Vector.
(0308) = A/C734 Statement Execution Vector.
(030A) = A/CE86 Evaluate Operand or Handle Monadic Operator Vector.

Operation:
Copy the six vectors at E447/E44F to 0300–030B.

Set Zero Page Variables, Restore CHRGET to Page 0, and Set Memory Pointers E3BF/E3A4-E421/E403

Called by: JSR at E397/E37B.

Exit Conditions:
00–02 (VIC); 0310–0312 (64)—JMP B/D248—USR function default JMP to display ILLEGAL QUANTITY error message.
(03) = vector to floating point to integer conversion routine, B/D1AA.
(05) = vector to integer to floating point conversion routine, B/D391.
13 = $00 I/O Channel number—keyboard.
16 = $19 pointer to available slot in the temporary string descriptor stack.
18 = $00 high byte of pointer to last string descriptor in temporary string descriptor stack.
(2B) = pointer to the start of the area for the tokenized BASIC program.
(33) = pointer to the top of available free space, initialized to the end of contiguous RAM.
(37) = pointer to the end of BASIC memory, initialized to the end of contiguous RAM.

53 = $03 three-byte size of string descriptor for garbage collection.
54 = $4C, the JMP opcode for performing a direct jump to a function whose address has been stored at (55).
68 = $00 overflow area when normalizing FAC1.
73–8A = CHRGET routine.
8B–8F = RND initial seed value.

Called by: FF9C, FF99.

Operation:

1. Store the opcode for the direct JMP at 0310/00 and at 54. The JMP at 0310/00 is for the USR function; the JMP at 54 is for all other function routines. See the INVOKE FUNCTION routine at A/CFA7 for how the address in (55) is set.
2. Initialize the USR address at (0311)/(01) to B/D248 to display ILLEGAL QUANTITY if USR is used with this default address.
3. Initialize (05), the vector to the integer to floating point conversion routine, to B/D391. This vector is never used by BASIC or the Kernal.
4. Initialize (03), the vector to the floating point to integer conversion routine, to B/D1AA. This vector is never used by BASIC or the Kernal.
5. Copy the RND seed of .811635157 in floating point format of 80-4F-C7-52-58 to 8B through 8F.
6. Copy the CHRGET routine from E3A2/E387–E3B9/E39E to 73 through 8A.
7. Initialize 53 for three-byte string description use in garbage collection.
8. 68 = $00, the FAC1 overflow byte.
9. 13 = $00, setting initial I/O channel of keyboard.
10. 18 = $00, the pointer to the last string descriptor on the temporary string descriptor stack.
11. Put $01 in stack locations 01FC and 01FD.
12. 16 = $19, setting the pointer to an available slot on the temporary string descriptor stack to be the first slot of the three available slots.
13. JSR FF9C, the Kernal MEMBOT routine, with carry set to read the start of memory which is stored in (0281) during the Kernal initialize memory routine. For the VIC-20, the initialize memory test starts at location 0401. When the

first RAM location is found, the page of that location is stored in 0282, and 0281 remains zero. For the Commodore 64, (0281) should be 0800. For the VIC with no expansion, (0281) should be 1000; with 3K expansion, (0281) should be 0400; and if greater than 3K expansion, (0281) should be 1200. Store the start of memory at 2B.

14. JSR $FF99, the Kernal MEMTOP routine, to read the end of memory. Store the end of memory in the pointer (37) and also in (33). (37) points to the end of BASIC, while (33) points to the top of the available free space. On the Commodore 64, this pointer should be 9FFF; on the VIC-20 it varies, depending on your memory size. Store $00 at the location pointed to by (2B), and then increment (2B).

Initialization Message
E422/E404–E446/E428 Display BASIC version and number of free bytes, then do NEW

Called by: JSR at E39A/E37E.

Operation:

1. JSR A/C408 to see if the pointer to the start of the area reserved for the tokenized BASIC program, (2B), is <= the pointer to the top of free space, (33). If not, display OUT OF MEMORY error message.
2. JSR A/CB1E to display the message at E473/E436. For the Commodore 64, the message is:

   **** COMMODORE 64 BASIC V2 ****
   64K RAM SYSTEM
   For the VIC, the message is:
   **** CBM BASIC V2 ****

3. Subtract (2B), the start of the area reserved for the tokenized BASIC program, from (37), the pointer to the end of memory, leaving the low-order result in the X-register and the high-order in the accumulator.
4. JSR B/DDCD to convert the X-register and the accumulator containing the number of free bytes to a floating point number in FAC1. Then convert FAC1 to an ASCII string and send the ASCII string to the current output device, thus displaying the number of free bytes.

5. JSR A/CB1E to display the message at E460/E429—BASIC BYTES FREE for the Commodore 64 or BYTES FREE for the VIC.
6. JMP A/C644 to perform NEW.

Allow Error Vector and Message During Cold or Warm Start (Commodore 64 only) E386–E393

When a cold or warm start occurs, this routine allows execution of a routine pointed to by the vector at (0300) and a possible call to display an error message.

Called by: BNE at E3A0; fall through from E385.

Note: During a cold start, (0300) is set to E38B before this step is executed, preventing you from using your own vector routine. During a warm start, (0300) is not reset.

Operation:
1. X-register = $80.
2. JMP(0300), with the default of E38B.
3. E38B—If X-register is < $80, JMP A43A to display error message, display READY and continue with the Main BASIC Loop. If the X-register is >= $80, JMP A474 to display READY and continue with the Main BASIC Loop.

BASIC Warm Start E37B/E467–E385/E474

Called by:
JMP(A/C002) at FE6F/FEDB—When the STOP and RESTORE keys are held down and no autostart cartridge exists or when a BRK instruction occurs, this BASIC Warm Start routine is called.

Operation:
1. JSR FFCC, the Kernal CLRCHN routine to clear I/O channels and restore default values.
2. 13 = $00, setting the I/O CMD channel to the keyboard.
3. JSR A/C67A to reset stack pointer, string descriptor temporary stack pointer, disallow CONT, and clear 10, the FN or subscript flag.

4. Enable IRQ interrupts.
5. Commodore 64: Fall through to E386. See previous routine for details.
   VIC: JMP C474 to display READY and continue with the Main BASIC Loop.

Entry Phase

During the BASIC entry phase, the Kernal CHRIN routine is called to receive bytes from the current input device. Bytes received are stored in the 89-byte BASIC input buffer at 0200–0258. The Commodore 64 allows only 80 bytes in a logical input line from the screen, while the VIC permits 88 bytes. Once a carriage return ($0D) is received from the input device, $00 is stored in the buffer to signify the end of this BASIC line.

After the carriage return is received, the text pointer (7A) is set to $01FF, and CHRGET scans the buffer at 0200 for the first character. If the first character is a number, the line just entered contains a line number. Logical input lines that contain a line number are then tokenized and stored in the program text area. If a line with the same line number already exists in the program text area, this new line replaces the old one. Once the line has been placed in the program text area, the entry phase is restarted.

If the first character of a logical input line is not a number, the statement just entered is a direct mode statement. The statement is tokenized, with the tokenized format residing in the buffer at 0200. Then the routine to execute statements is called to immediately execute this direct mode line.

The READY message is displayed after a BASIC cold or warm start, after completion of direct mode statement execution, and after error situations. The READY message then falls through to the Main BASIC Loop that handles the entry phase.

The vector at (0302) with the default of C483 is called by the Main BASIC Loop. You can change this vector to point to your own routine and receive control before the routine to fill the BASIC input buffer is executed.

Display READY
A/C474-A/C47F

Called by:
JMP at A/C714 List Command; JMP at A/C854 End Command; BEQ at A/C46F Error Message; fall through from A/C473 Error Message; JMP at E384 VIC Cold Start; JMP at E391 Commodore 64 Warm/Cold Start; JMP at E472 VIC Warm Start.

Operation:
1. JSR A/CB1E to display the READY message at A/C376.
2. JSR FF90 to the Kernal SETMSG routine to ORA the message control flag, 9D, with $80, allowing control messages.
3. Fall through to A/C480, the Main BASIC Loop Routine.

Main BASIC Loop A/C480-A/C49B

Whenever entry of a new BASIC line is needed, this routine does the necessary work to fill the input buffer at 0200 and then calls the appropriate routines to handle direct mode or program mode lines.

Called by:
Falls through from C47F READY; BEQ at C4F6 Store/Replace BASIC line with line having only line number; JMP at A/C530 Store/Replace BASIC line.

Operation:
1. JMP(0302) with default of A/C483 to step 2.
2. A/C483: JSR A/C560 to put the character from the current input device into the BASIC input buffer at 0200 until a carriage return is received, when a $00 is then stored in the buffer. Exit with X and Y pointing to 01FF.
3. Set (7A), the text pointer for CHRGET, to 01FF from X and Y.
4. JSR 0073 to CHRGET to retrieve the first character in the buffer that is not a space. If the character is numeric, carry is clear.
5. If the byte returned = $00, only a carriage return was entered for this line. Branch to step 1.
6. Set 3A to $FF to indicate that a direct mode statement is being processed, until told different.
7. If carry is clear, indicating a numeric character started the buffer, branch to A/C49C to store/replace this line in the BASIC program text area.
8. If carry is set, this is a direct mode statement with no line number. JSR A/C579 to tokenize the line in the BASIC buffer at 0200.
9. JMP A/C7E1 to execute the direct mode statement.

Fill Input Buffer
A/C560–A/C578

The Kernal CHRIN routine is called to fill the BASIC buffer at 0200, ending when a carriage return is received.

Called by:
JSR at A/C483 Main BASIC Loop; JSR at A/CC03 INPUT command.

Operation:
1. X = $00 to set starting index for storing into buffer at 0200.
2. JSR E112/E10F to call the Kernal CHRIN routine to retrieve the character from the current input device.
3. Is the character = $0D (carriage return)? If yes, branch to step 7.
4. Store the character in the BASIC input buffer at 0200, using X as an index into the buffer, then INX.
5. If X < 89 (decimal), branch to step 2.
6. JMP A/C437 to display STRING TOO LONG if X >= 89 (decimal).
7. JMP A/CACA to store $00 at 0200,X. This puts the end-of-line byte in the buffer. Set X and Y to point to 01FF. If 13, the current channel for BASIC I/O, = 0, then send a carriage return to the current input device. Also, send a linefeed if 13 is > 127.

BASIC's CHRIN
E112/E10F–E117/E114

Called by: JSR at A/C562 Fill Input Buffer.

Operation:
1. JSR FFCF to the Kernal CHRIN routine to retrieve the next character from the current input device and return the character in A.
2. If carry is set, indicating an error during CHRIN, branch to E0F9/E0F6 to handle errors from BASIC I/O calls.

CHRGET/ CHRGOT

The CHRGET/CHRGOT routine is a very active routine—CHRGET is called 25 times and CHRGOT 23 times by other BASIC routines. CHRGET refers to the routine at 0073; CHRGOT to the entry point at 0079. A copy of CHRGET exists in BASIC ROM at E3A2/E387–E3B9/E39E and is copied to 0073–008A during a BASIC cold start.

CHRGET is used to scan the BASIC input buffer when tokenizing the buffer, to scan BASIC statements stored in the program text area, to scan the input buffer for INPUT or GET, and to scan for DATA statements for READ. Each time CHRGET is called, the next character in the buffer or area being scanned is returned in the accumulator. To correctly retrieve this next character, CHRGET modifies itself when it executes, incrementing the operand at 7A–7B. This operand is used in an LDA instruction to retrieve the next byte from the area being scanned.

CHRGOT does not increment the operand at 7A–7B. Rather, it just uses the current value of 7A–7B for the LDA.

Both CHRGET and CHRGOT set flags indicating what kind of character was retrieved. More flags are set than are actually used. The ones used are: carry set if the character is nonnumeric; carry clear if the character is numeric; Z = 1 (BEQ condition) if it is a colon ($3A) or end-of-line byte ($00). CHRGET and CHRGOT also skip over any spaces ($20) they encounter.

One bug in using CHRGET/CHRGOT occurs when this routine is called by the TI$ function. TI$ = "XYZABC" is valid, although I have always interpreted the *Programmer's Reference Guides* to the VIC and to the Commodore 64 to imply that a numeric string was required. However, the CHRGET/CHRGOT routine is entered at 0080 for TI$ and the carry flag is set, in this case, if the value in the accumulator is >=$30. Thus both numbers and characters are valid for assignment to TI$. Only ASCII values <$30 are invalid.

A few examples should help demonstrate how the flags are set for CHRGET/CHRGOT. If you wonder why $30 is subtracted and then $D0 is subtracted, the reason is that $30 + $D0 = $00 (with carry set) and thus zero is actually being subtracted, leaving the value unchanged in the accumulator.

Examples:

First, when subtracting $30 with the carry set, subtraction is the same as addition in two's complement form. The calculation to add $30 in two's complement form is:

$30	**=**	**0011 0000**
complement		**1100 1111**
add 1		**0000 0001**
		1101 0000

add complement of carry, which effectively adds 0.
Also, the calculation to subtract $D0 by adding its two's complement form is:

$D0	**=**	**1101 0000**
complement		**0010 1111**
add 1		**0000 0001**
		0011 0000

Okay, for the first example, see what flags are set for the lowest numeric value, $30.

$30	**=**	**0011 0000**
+ 2's comp of $30	**=**	**1101 0000**
Carry	**= 1**	**0000 0000**
+ 2's comp of $D0	**=**	**0011 0000**
Carry	**= 0**	**0011 0000**

Thus, for $30, the carry is clear at exit, indicating a digit, and the value in the accumulator is unchanged. For the next example, let's try $2F, the ASCII value for /.

$2F	**=**	**0010 1111**
+ 2's comp of $30	**=**	**1101 0000**
Carry	**= 0**	**1111 1111**
+ 2's comp of $D0	**=**	**0011 0000**
Carry	**= 1**	**0010 1111**

Thus, for $2F, the carry is set at exit, indicting a nonnumeric value, and the accumulator is unchanged. For values

>$39, the CMP $3A at 7C branches with the carry set for the RTS at 8A. For the final example, let's show the bug that occurs when entry at 0080 is made by TI$, using an example of **$41 for "A".**

```
$41                  =       0100 0001
+ 2's comp of $30    =       1101 0000
               Carry = 1     0001 0001
+ 2's comp of $D0    =       0011 0000
               Carry = 0     0100 0001
```

Thus, for $41, when called at 0080 by TI$, the carry is clear for a nonnumeric value, allowing the TI$ = "ABCDEF" bug. This could be fixed by having TI$ JSR $007E so that the CMP $3A and BCS $8A would pick off the values greater than $39 and return with the carry set.

CHRGET/CHRGOT
0073–008A

Called by:
JSR at A/C48A Main BASIC Loop; JSR at A/C6B3 LIST; JSR at A/C799 FOR; JSR at A/C7E4 Statement Execution Control; JMP at A/C801; JSR at A/C812 Execute the Current BASIC Statement; JSR at A/C95F ON; JSR at A/CB13 Handle Print for TAB, SPC, Comma, and Semicolon; JSR at A/CB82 GET and GET#; JSR at A/C99F Convert ASCII Decimal Number to Two-Byte Hex Value; JSR at A/CC51 READ; JSR at A/CD84 NEXT; JSR at A/CDD1 Main Expression Evaluation; JSR at A/CE8A Evaluate Operand or Handle Monadic Operator; JMP at A/CEA5 Evaluate Operand or Handle Monadic Operator; JSR at A/CF05 Syntax Check for Requested Character; JSR at A/CFAA Function Invocation; JSR at B/D0A5 Locate or Create Variable; JSR at B/D0B0 Locate or Create Variable; JSR at B/D0D8 Locate or Create Variable; JSR at B/D1B2 Convert Numeric Expression from Floating Point to Fixed Point Integer; JSR at B/D79B Get Expression at Next Text Pointer Location; JSR at B/DD0A, B/DD17, B/DD30 ASCII Decimal Number to Binary Floating Point Conversion.

Alternate 0079: JSR at A/C6AA LIST; JSR at A/C792 FOR; JSR at A/C897 Do GOTO for GOSUB or RUN when RUN Is Followed by Parameter; JSR at A/C92B IF; JSR at A/C940 IF; JSR at A/CA9D Print String from a String Expression of a PRINT Statement; JSR at A/CC2C, A/CC91, AS/CCAD, A/CCD4 READ; JSR at A/CD7D NEXT; JSR at A/CD8B Type Check Variable or Expression; JSR at B/D085 DIM; JSR at B/D08D, B/D094 Locate or Create Variable; JSR at B/D202 Locate or Create Array; JSR at B/D441 FN; JSR at B/D73B MID$; JMP at B/D7AA Convert Floating Point to Integer in the Range 0–255; JSR at B/D7D7 VAL; JSR at B/D834 WAIT; JSR at E206/E203 Check If End of Parameter List; JSR at E211/E20E Check for Commas Between Parameters.

Alternate 0080: JSR at A/CA1F Add Character for TI$.

Operation:

1. 0073: Increment (7A), the pointer to the buffer or area being scanned. This increment does a self-modification of this code.
2. 0079: LDA from the location pointed to by (7A).
3. Branch to step 7 with carry set if the accumulator is greater than or equal to $3A. If equal, then the : statement separator has been loaded and the Z flag = 1.
4. 0080: If the accumulator = $20, a space has been loaded. Skip over spaces by branching back to step 1.
5. Subtract $30 with carry set.
6. Subtract $D0 with carry set to restore the accumulator to its original value before step 5 and to give the flag the correct setting.
7. RTS.

Tokenization and Program Storage

After a line has been received from the current input device and stored in the BASIC input buffer at 0200, the line is converted to its tokenized format. In tokenized format, each BASIC keyword found in the input buffer is converted to its unique one-byte token. The resulting tokenized line is stored back into the buffer at 0200. Tokenization of keywords saves space when storing lines in the program text area. Also, when executing statements, the presence of bytes with high-order bits on (which is the format for tokens) allows an easy method of determining the keywords.

This tokenization routine is called both when direct mode lines are entered and when program mode lines with line numbers are entered. However, in the case of program mode lines, the line number is stored in (14) and the text pointer is moved past the line number before the tokenization routine is called.

During tokenization, keywords are matched by scanning the keyword table that is located at A/C09E–A/C19D. The keyword table is terminated by a final byte of $00. Tokens are not actually stored in the keyword table. Instead, each time the table is scanned, an index into the current keyword in the table being scanned is maintained. When a keyword match is found, the current index is OR'd with $80, and this token is placed in the BASIC buffer in place of the keyword. Thus all tokens have the high-order bit on.

Spaces are not removed during tokenization. Any bytes >=$80 are deleted from the BASIC input buffer during tokenization. With the exception of the ?, all other ASCII values <$80 are left unchanged in the BASIC buffer. For example, $5C, the ASCII value for /, is left unchanged. The ? value, $3F, is changed to the token for PRINT—$99.

If the DATA or REM keyword is found in the BASIC buffer, all following values in the buffer are stored back into the buffer unchanged. However, if a : signifying the end of the current statement is found after a DATA statement, tokenization is reenabled.

If a quotation mark is found, all following bytes in the input buffer up to another quotation mark are left unchanged.

Whenever the $00 (end-of-line stored when the carriage return was detected) is found, this $00 is stored back into the BASIC input buffer. Also, another $00 is stored two bytes past this byte in the buffer. This additional $00 is used after a direct mode statement is executed to signify that no following statement needs to be executed.

At exit from the tokenization routine, the Y-register = length of the tokenized line plus four, and (7A) points to $01FF.

Each keyword in the keyword table has the final letter in the keyword as the normal ASCII value for that letter, but also has the high-order bit on. During the scan for matching keywords to the current word in the BASIC buffer, if a letter from the buffer matches the keyword buffer except for the high-order bit, then the letters are considered a match. This feature is used for the second or later letter in a keyword and allows keywords to be abbreviated. For example, N shift-E and NE shift-X are both valid abbreviations for NEXT. The shifted ASCII value of the character evaluates to $C0–$DA for A–Z. If two keywords have the same starting letters, such as READ and RESTORE, the first entry in the keyword table is selected. For example, R shift-E gives the token for READ because READ is located first in the table.

The highest token value used is $CB.

Tokenize Input Buffer A/C579–A/C612

Called by:
JSR at A/C496 Main BASIC Loop—Direct Mode Statement;
JSR at A/C49F Store/Replace BASIC line in program text.

Operation:

1. JMP (0304) with the default vector in 0304 of A/C57C.
2. A/C57C: X-register = 7A: which is the index value into the buffer at 0200 to the first letter in the buffer if it's a direct mode line; or the first letter in the buffer past the line number if it's a program mode line.
3. Y-register = $04, setting the index value for storing characters back into the buffer off 01FB. Before the first character

is placed back into the output buffer, the Y-register is incremented to $05. Thus 01FB + 5 = 0200, which is where the storing of output characters into the buffer begins. The Y-register is set to 4 because the two-byte link field and the two-byte line number need space reserved when storing the program line. The Y-register at exit gives the length of the tokenized line plus four. This information is used when obtaining space for the link address and line number.

4. 0F = Y-register. 0F is checked later to see if a DATA token has been encountered.
5. Load the accumulator with the next character from the input buffer—0200,X.
6. If this character is <$80, branch to step 9.
7. If the character is $FF, the ASCII value for PI, branch to step 28.
8. For all other values > $80, INX and branch to step 5, thus deleting all values > $80 from the buffer.
9. If the character is a space, branch to step 28 as spaces are copied unchanged from the input buffer.
10. Save the character in 08.
11. If the character is a quote, branch to step 39.
12. If bit 6 of 0F = 1, which it is if a DATA keyword has been found, branch to step 28.
13. If the character is ?, set the accumulator = $99, the token for PRINT, and branch to step 28.
14. If the character is < $30, the ASCII code for 0, branch to step 16.
15. If the character is < $3C, branch to step 28. If it is < $3C, the character is a number, a colon, or a semicolon.
16. This step is reached if the character is $00–$2F or $3C–$7F (except for $3F, ?). 71 = Y-register, saving the index into the buffer at 0200 for output.
17. 0B = $00, initializing the index into the table of keywords.
18. DEY; Y-register now = $FF. Set to $FF because the loop for reading characters of the keyword table does INY before retrieving the character. The first INY will set the Y-register to $00.
19. 7A = X-register, saving the index into the buffer at 0200 for input.
20. DEX to point previous to the character from the input buffer because the loop for reading characters from the input buffer does an initial INX.

21. INY and INX, thus moving the index to the next character in the keyword table and the next character in the input buffer.
22. Accumulator = 0200 indexed by the X-register, which is the next character from the input buffer. When this step is branched to after an unsuccessful keyword match from step 43, the X-register has been reset from the 7A that was stored in step 19, thus pointing to the start of this sequence of letters again.
23. Subtract the next character from the keyword table, A/C09E indexed by Y-register, from the accumulator.
24. If the two characters are equal, the subtraction leaves $00 in the accumulator. Branch if $00 to step 21 to try the next two characters since these matched okay.
25. If the accumulator = $80 after subtraction, only the high-order bit differs. Continue with step 26 if true, or branch to step 40 if false. This is true if the buffer character has its high-order bit on and the table character does not, or if the table character has its high-order bit on and the buffer character does not. This check allows for abbreviated keywords and also provides a check for the end of a non-abbreviated keyword.
26. Accumulator = 0B, the current index into the table of keywords, ORA with $80, thus creating the token for this keyword with the high-order bit on.
27. Y-register = 71, the saved index into the buffer at 0200 for output. 71 is the same value as when the search for this keyword began, and by using this value, all the keyword letters in the input buffer are not copied to the buffer on output.
28. INX, the index into the buffer for input, and INY, the index into the buffer for output.
29. STA 01FB,Y. The original character in the input buffer or the token for a keyword in the input buffer is copied to the output buffer. For example, if the buffer at 0200 started with X, then this would now store at 01FB,5—at 0200. The INX in step 28 would have moved the input index to 0201. Thus the input index stays one byte ahead in the buffer of the output index.
30. Now see if this character stored was $00, the end-of-line byte, by reloading the accumulator from 01FB,Y. If it is $00, branch to step 45.

31. If the character is not $00, see if it is a colon by subtracting $3A, the ASCII value of a colon, from the accumulator and branching to step 33 if the result is $00, indicating a colon.
32. Compare the accumulator to $49 to check for the DATA token. $49 + $3A = $83, the token for DATA. If not equal, branch to step 34.
33. 0F = accumulator. If :, then $0F = 00, thus resetting any DATA setting from the just-ended statement. If it's a DATA token, set 0F to $49 to force the following characters in the input buffer for this statement to be copied unchanged to the output buffer.
34. Check for the REM token by subtracting $55. $55 + $3A = $8F, the token for REM. If it's not REM, branch to step 4.
35. If it's a token for REM, 08 = $00.
36. If it's a token for REM or a quotation mark, just copy the input buffer to the output buffer. Accumulator = 0200 indexed by the X-register.
37. If this byte just retrieved is $00, the end-of-line byte, branch to step 28.
38. Compare the accumulator to 08, which contains $22 for the ASCII quotation mark if an odd number of quotation marks have been found so far. If equal, branch to step 29 to store this ending quote.
39. Branch here if a quotation mark is detected from step 11; also fall through from step 38.

    INY, STA at 01FB indexed by the Y-register, then INX. Then branch to 36. This step does the actual copying of the input buffer to the output buffer for REM or quote.
40. If not a match of the input buffer character to the keyword table character, then branch here from step 25.

    X-register = 7A, which points to the character from the input buffer.
41. 0B, the index into the keyword table, is incremented since the keyword just examined is not a match.
42. Now advance the Y-register past this keyword by INY then loading the next byte of this keyword. Repeat this loop until a character is found with its high-order bit on, which indicates the end of this keyword.
43. If this next byte past the keyword is not $00, branch to step 22.

44. If this next byte past the keyword is $00, the end of the keyword table has been reached. In this case the letter from the buffer has no match and should be copied unchanged into the buffer on output. Reload the character into the accumulator from 0200 indexed by the X-register. Since letters that don't match keywords are copied unchanged into the buffer on output, words such as VERB or CHAR are copied unchanged. The words copied can serve as variables, or a SYNTAX ERROR occurs when the statement is executed.

    Unconditionally branch to step 27.
45. After the final $00 end-of-line byte has been stored to the buffer, branch here from step 30 with the accumulator = $00. Store the accumulator at 01FD indexed by the Y-register. Since the previous $00 was stored at 01FB indexed by the Y-register, this stores another $00 two bytes past the original $00. The Y-register now contains at exit the length of the tokenized line in the buffer at 0200 plus four for the link address and line number.
46. Set (7A) to $01FF to again point to the byte previous to the input buffer at 0200, then RTS.

Table of BASIC Keywords

A/C09E-A/C19D

This table of BASIC keywords is in the same order as the keyword dispatch vector table at A/C00C followed by the function dispatch vector table at A/C052. Thus the tokens derived from scanning the keyword table can be used to index into the dispatch tables. The table of BASIC keywords is used by the tokenization routine. Each final letter of a keyword has its ASCII value ORA with $80 to turn its high-order bit on. The final byte in the table is signified by a $00. Certain keywords include as the final symbol of the keyword #, $, or (. Since these are included as part of the keyword, these characters are removed from the input buffer when the keyword is tokenized. Keep this in mind when you read the routines for scanning and executing BASIC statements. Also note that GET does not have a GET# keyword and thus the GET# routine has to specifically check for a #.

BASIC Keywords and Their Associated Tokens:

END	$80	FOR	$81	NEXT	$82
DATA	$83	INPUT#	$84	INPUT	$85
DIM	$86	READ	$87	LET	$88
GOTO	$89	RUN	$8A	IF	$8B
RESTORE	$8C	GOSUB	$8D	RETURN	$8E
REM	$8F	STOP	$90	ON	$91
WAIT	$92	LOAD	$93	SAVE	$94
VERIFY	$95	DEF	$96	POKE	$97
PRINT#	$98	PRINT	$99	CONT	$9A
LIST	$9B	CLR	$9C	CMD	$9D
SYS	$9E	OPEN	$9F	CLOSE	$A0
GET	$A1	NEW	$A2	TAB(	$A3
TO	$A4	FN	$A5	SPC(	$A6
THEN	$A7	NOT	$A8	STEP	$A9
+	$AA	–	$AB	*	$AC
/	$AD	↑	$AE	AND	$AF
OR	$B0	>	$B1	=	$B2
<	$B3	SGN	$B4	INT	$B5
ABS	$B6	USR	$B7	FRE	$B8
POS	$B9	SQR	$BA	RND	$BB
LOG	$BC	EXP	$BD	COS	$BE
SIN	$BF	TAN	$C0	ATN	$C1
PEEK	$C2	LEN	$C3	STR$	$C4
VAL	$C5	ASC	$C6	CHR$	$C7
LEFT$	$C8	RIGHT$	$C9	MID$	$CA
GO	$CB				

Note: These keywords are in the same order as those in the keyword dispatch vector table at A/C00C.

Delete/Store Program Lines

If the line that has been received from the current input device and stored in the BASIC input buffer at 0200 starts with a number, then that number is the line number of a BASIC line. This routine is called to store a line into the BASIC program text area after tokenization. If a line with the same number already exists in the program text area, that line is deleted and this one is inserted in its place. If no line with this number exists in the program text area, room for this line is created and the line inserted.

If the line contains only a line number, this line is not inserted in the program text. However, if a line exists with the same line number, that line is deleted from the program text area.

Before the input buffer at 0200 is tokenized, the line number at the start of the buffer is converted to a two-byte line number format in 14 (low-order value) and 15 (high-order value). The largest line number possible is 63999 (decimal). The text pointer to the input buffer is also moved to the first nonnumeric character past the leading numeric characters so that the line number is not stored as part of the tokenized text.

After the line number has been scanned, the rest of the line in the input buffer is converted to its tokenized format.

Then a search is made for the line number in the BASIC program text area. After seaching the program text area, (5F) points to the location in the program text area of the line with this number if the line number was found or to the location in the program text area of the next higher numbered line if the line number is not found. The lowest-to-highest sequential storing of lines allows this next higher line number to be found.

BASIC lines are tokenized and stored in the program text area in the following format. The program text is preceded by a $00. Each line starts with a two-byte pointer or link to the next BASIC line. This pointer has the high-byte value followed by the low-byte value. Following this two-byte link is a two-byte line number, again in high byte/low byte sequence. Following the line number is the tokenized text, ended by a $00 end-of-line byte. The end of the program text is indicated by a link of $0000.

If the line number was found, then this line pointed to by (5F) is deleted by moving the program text area down in memory to where this line was located, starting the move from the following line.

After space has been created for the line to be inserted, the line is copied from the BASIC input buffer to the area reserved for the line. When this copy is done, two dummy link bytes are copied to the start of the line followed by the two valid bytes for the line number. The lines in the program text area are then relinked. The text pointer (7A) is reset to the start of the program area, CLR is done to erase any variables, and the Main BASIC Loop is reentered.

Delete/Store BASIC Line in Program Text Area A/C49C–A/C532

Called by: BCC at A/C494 Main BASIC Loop.

Operation:

1. JSR A/C96B to convert the line number operation at the start of the buffer into a two-byte line number in 14 (low-order) and 15 (high-order). Return with (7A) pointing to the first nonnumeric character past the line number and with the accumulator containing this first nonnumeric character.
2. JSR A/C579 to tokenize the rest of the BASIC input buffer past the line number.
3. 0B = Y-register, which contains the length of the tokenized line plus four.
4. JSR A/C613 to see if a line with this same line number already exists in the program text area. If one does, then return with carry set and (5F) pointing to the start of the next higher numbered line in the text area.
5. If carry is clear, indicating the line did not already exist, branch to step 27.
6. A/C4A9: *Note:* If you want to delete a line from the text area, enter here with (5F) pointing to the line to be deleted.

   Store the high-order link address of this line in 23.

   In order to delete a line, all of the program text that follows the line to be deleted needs to be moved down in memory (towards the start of the program text area). The program text to be moved must be moved an amount equal to the number of bytes in the line being deleted. To determine the number of bytes for this move, subtract the start of the next line from the pointer to the end plus one of the program text area. Another way is to subtract the start of the current line plus the length of the current line from the pointer to the end plus one of the program text area.

   When moving program text, the move starts with bytes at the line following the line to be deleted and moves bytes to the location of the line that is being deleted. The move works upward in memory from this start until all bytes have been moved. By moving in this direction, a nondestructive move is done where the source bytes don't overlay the destination bytes.

7. Store 2D, the low-order pointer to the end of the BASIC text area, into 22.
8. Store 60, the high-pointer byte, into 25.
9. Accumulator = 5F, the low-pointer byte.
10. DEY to back the Y-register up to index to low-order link address of this line.
11. Subtract the low-order link address of this line from the low-order pointer to this BASIC line, thus getting the low-order length of this line. Since a line can't be longer than 89 bytes, this one-byte field is sufficient to hold the length of the line. This length is a negative number in two's complement form.
12. Add 2D, the low-order pointer to the end plus one of the BASIC text area, and store the result back into 2D.
13. Also store in 24, the low-order byte of the destination move pointer.
14. If carry was set in step 12, leave 2E unchanged; otherwise, decrement 2E, which is the high-order pointer to the end plus one of the BASIC text area. Accumulator = 2E.
15. Subtract 60, the high-pointer byte to this BASIC line, from the accumulator.
16. The X-register now receives the accumulator, the number of pages of program text to be moved when this line is deleted.
17. Subtract 2D, the low-order pointer to the end of the program text area, from 5F, the low-order pointer to this line, to obtain the number of bytes in the first page that need to be moved. This could also be a negative number in two's complement form.
18. The Y-register now receives the number of bytes to move in the first page.
19. If 2D < 5F, branch to step 21.
20. If 2D >= 5F, then INX, the number of pages to move, and decrement 25, the high-order byte of the destination move pointer.
21. Add 22, the temporary value for the low-order pointer to the end of the BASIC text area, to the accumulator, the number of bytes to move. If carry is clear after add, branch to step 23.
22. If carry is set, then DEC 23 and CLC.
23. Move the byte from (22),Y to (24),Y. (22) points to the low

end of the text area to be moved; (24) points to the area that is to become the new low end.

24. Branch to step 23 until all bytes in this page of memory have been moved by INY until the Y-register = $00.
25. Once a page has been moved, increment 23 and 25, the low-order bytes of the pointer to the areas being moved.
26. DEX and continue the move by branching to step 23 until X becomes $00.

Steps 6–26 give the details of the deletion operation. Here is an example of how deletion actually works:

Assume the following BASIC program has been entered:

```
1 A = 1
2 B = 2
3 C = 3
4 D = 4
5 E = 5
```

This program is stored in tokenized format on a VIC with 3K expansion in the area at 0401 as follows:

```
Address  Link   Line#     Program Text   End
0401:    0B 04  01  00  41  20  B2  20  31  00
040B:    15 04  02  00  42  20  B2  20  32  00
0415:    1F 04  03  00  43  20  B2  20  33  00
041F:    29 04  04  00  44  20  B2  20  34  00
0429:    33 04  05  00  45  20  B2  20  35  00
0433:    00 00
```

(2D), the pointer to the end+1 of the program text area = $0435 (2D = $35 and 2E = $04)

If line 2 is to be deleted, then at entry to step 6, (5F) will point to $040B (5F = $0B and $60 = $04). Following are the values after the steps indicated have been executed:

Step six: 23 = $04.
Step seven: 22 = $35.
Step eight: 25 = $04.
Step nine: accumulator = $0B.
Step ten: Y-register = $00.
Step eleven: accumulator = $F6, the two's complement representation of −10, which results after $0B − $15.
Step twelve: accumulator = 2D + $F6 = $35 + $F6 = $2B. 2D = $2B.
Step thirteen: 24 = $2B.

Step fourteen: 2E = $04 as carry was set in step 12.
Step fifteen: accumulator = $04 − 60 = $04 − $04 = $00.
Step sixteen: X-register = $00.
Step seventeen: accumulator = 5F − 2D = $0B − $2B = −$20 or $E0 in two's complement form. These are the 32 bytes that need to be moved (lines 3, 4, and 5, 10 bytes each, and the two-byte final link).
Step eighteen: Y-register = $E0.
Step nineteen: 2D > 5F so INX and DEC 25. X-register = $01 and 25 = $03.
Step twenty: accumulator = $E0 + 22 = $E0 + $35 = $15 with carry set.
Step twenty-one: DEC 23. 23 = $03.
Now at *step twenty-two* it is time to start the move, moving from (22),Y to (24),Y. (22) = $0335 and (24) = $032B. Y-register = $E0. $0335 + $E0 = $0415. $032B + $E0 = $040B. Now if you look at the display of how the program text is stored above, you see that $040B is where the second line was located, and $0415 is where the third line is located. The move routine then moves the 32 bytes from the source program text area starting at line 3 to the destination area starting at line two.

27. A/4CED: This is a possible entry point if you want to insert a line into the program text area. (5F) = pointer to the location where the line is to be inserted, which is the next highest numbered line or past the last line. Also, 0B = length of tokenized line in buffer at 0200 plus four.
    JSR A/C659 to reset (7A), the text pointer, to the start of the tokenized program area minus one. Also do a CLR. Thus, entry of a line number causes all variables to be erased.
28. JSR A/C533 to reset the link fields in all the BASIC lines that are stored in the program text area.
29. If the first byte of the BASIC input buffer is $00, a line was entered that contained only a line number. In this case, branch to A/C480 to do the JMP to the Main BASIC Loop.
30. Accumulator = 2D, the low-order byte of the pointer to the end plus one of the BASIC program. Store in 5A.
31. Add 0B, the length of the tokenized line.

32. Store in 58, the low-order byte of the end location of the move.
33. Y-register = 2E, the high-order byte of the pointer to the end plus one of the BASIC program. Store in 5B. Thus, (5A) = the original pointer to the end plus one of the BASIC program.
34. If the add in step 31 caused a carry, then INY.
35. Store the Y-register in 59, the high-order byte of the end location of the move.
36. JSR A/C3B8 to allocate space for the line by checking that free space is available. Once space is available, move all lines from (5F) to the end of the BASIC program up in memory by the number of spaces needed for this line. The move starts from the end of the area to be moved and works downward.
37. Retrieve the two-byte line number from (14) and store at 01FA, so the stack is not disturbed.
38. Store (31), the pointer to the start of free area, into (2D), the pointer to the end plus one of the BASIC program area.
39. Y-register = 0B, which contains the length of the tokenized line in the buffer at 0200 plus four. The four extra bytes in combination with the following BPL allow the buffer plus the four preceding bytes 01FC–01FF to be copied to the area created in the program text area. The 01FC and 01FD are just dummy values as the link field is reset in step 43. The line number is taken from 01FE and 01FF.
40. Move bytes from 01FC indexed by the Y-register to (5F) indexed by the Y-register, decrementing Y-register after each move and ending the move when the Y-register becomes negative. (5F) points to the area in the BASIC program region created for this line.
41. JSR A/C659 to reset (7A), the text pointer, to the start of the program text area and to do CLR.
42. JSR A/C533 to relink the BASIC lines.
43. JMP A/C480 to return to the Main BASIC Loop to receive the next BASIC line from the current input device.

BASIC Line Numbers

Several other BASIC routines use this routine to convert a decimal ASCII line number to its two-byte hex equivalent. Decimal values > 63,999 produce a syntax error. This routine could be the basis for a general decimal-to-hex conversion routine. It can also be used as is for converting decimal values from 0 to 63,999 to a two-byte hex format. These decimal values to be converted must be in ASCII format.

The following short routine will do this conversion from a machine language program:

JSR $E3B4/$E3A4 to copy CHRGET to page 0.

LDA #$06 where this is the high-order pointer to your decimal ASCII string.

STA $7B.

LDA #$00 where this is the low-order pointer to your decimal ASCII string minus one.

STA $7A.

JSR $0073.

JSR $A/C96B to execute this ASCII decimal-to-hex conversion routine, with 14 and 15 at exit containing the hex value of your ASCII decimal string.

In this example, if at the location starting at 0601, I had $32 $33 $37 $38 for the ASCII decimal string 2378, then after this JSR to A/C96B then 14 = $4A and 15 = $09. $094A (hex) = 2378 (decimal).

This routine allows spaces between numbers since CHRGET skips over spaces. For example, 2 3 2 is a valid line number, the number 232.

Convert ASCII Decimal Number to Two-Byte Hex Value
A/C96B–A/C9A4

Called by:
JSR at A/C49C Store/Replace New BASIC Line; JSR at A/C8A0 GOTO; JSR at A/C6B6 LIST; JSR at A/C6A4 LIST; JSR at A/C962 ON.

Exit Conditions:
(14) = hex value of this ASCII decimal string in low-byte/high-byte format. The decimal string must be <64,000 or a SYNTAX ERROR results. (7A) points to

the first nonnumeric ASCII value past this ASCII decimal string, and the accumulator holds this first nonnumeric value.

Operation:

1. 14 and 15 = $00.
2. BCS to RTS if the last CHRGET retrieved a character that is nonnumeric.
3. SBC $2F from the ASCII numeric value in the accumulator. However, since the carry is clear, this SBC in effect subtracts $30.
4. Save the resulting 0–9 decimal value in 07.
5. If 15, the high-order byte of the hex result, is >= $19, then display SYNTAX ERROR and return to the Main BASIC Loop. $1900 = 6400 decimal. Since this check is made before the next multiplication by 10, this prevents any line number of 64,000 or higher.
6. Now multiply (14) by 10. To do this multiplication by 10, shift 14 left twice, which is (14) * 4. Then add 14 to this value, giving (14) * 5. Now shift this left once again, again multiplying by 2, giving the final result of 14 * 10.
7. After (14) has been multiplied by 10, add 07, which holds the digit just scanned.
8. JSR 0073 CHRGET to retrieve the next ASCII character.
9. Loop to step 2.

Search for a BASIC Line

The program text area is searched for the line number in (14). The carry is set on exit if the line is found, or the carry is cleared if the line is not found. If the line is found, (5F) contains the address of this line. If the line is not found, but a line number with a higher value is found, (5F) contains the address of this next higher numbered line. If the entire program text area is searched without finding the line, (5F) points to the address of the line with the two-byte $0000 link that indicates the end of the program.

A/C613-A/C641—Search for BASIC Line

Called by:

JSR at A/C4A4—Store/Replace BASIC Line; JSR A/C6A7 LIST; Alternate A/C617; JSR at A/C8C0 GOTO.

Operation:

1. Accumulator = 2B and X-register = 2C. (2B) pointer to the start of the tokenized BASIC program area.
2. A/C617: 5F = X-register, 60 = accumulator. (5F) points to the line to be compared.
3. Retrieve the high-order byte of the link address of this line.
4. If it's zero, this is the end-of-program $00 link byte as no program text would be stored in page 0. If it's 0, branch to step 13 to CLC, and RTS with (5F) pointing to this line with the end-of-program link address.
5. Accumulator = 15, the high-order byte of the target line number.
6. If it's 15, the high-order byte of the target line number, is < the high-order byte of the line number of this line, then BCC to step 14 with (5F) pointing to this line.
7. If the target line number 15 is > the high byte of this line number, branch to step 12.
8. If the target line number 15 is equal to the high byte of this line number, also check the low byte of target 14 to the low byte of this line number.
9. If 14 < this line number's low byte, the line number of this line is higher than the target line. BCC to step 14 with (5F) pointing to this line.
10. If 14 = this line number's low byte, the line number of this line is equal to the target line. BEQ to step 14 with carry set and (5F) pointing to this line.
11. If 14 > this line number's low byte, continue with step 12.
12. Load the accumulator and the X-register from the link address in this line and branch to step 2.
13. CLC.
14. RTS.

Relink BASIC Lines

After a line has been deleted or inserted in the program text area, the link addresses of all lines in the program text area are recalculated. All lines following the inserted one may have been shifted in memory, and the links to and from the new line also need to be inserted.

The link address forms the first two bytes of a line in low-order/high-order format, and a link address of $0000 indicates the end of the program.

A/C533-A/C55F Relink BASIC Lines

Called by:
JSR at C4F0—Store/Replace BASIC line; JSR at C52D—Store/Replace BASIC Line.

Operation:
1. Store (2B), the pointer to the start of the BASIC text area, into (22), a temporary pointer to the BASIC lines for this routine.
2. Load the second byte of the BASIC line, which is the high-order link address byte.
3. If this link byte is $00, the end-of-program link field has been reached, so branch to step 11.
4. Y-register = $04 to start count of characters in this line past the two-byte link address and the two-byte line number.
5. INY, incrementing the count of characters.
6. Accumulator = next character of line from (22) indexed by the Y-register.
7. If the accumulator is not $00, the end-of-line byte, branch to step 5. If $00, continue with step 8.
8. INY to make the Y-register = total number of bytes in the line, counting from 1, including the $00 end-of-line byte.
9. Add the Y-register to (22), storing the resulting address of the next line in the link field of this line and then back into (22) so that (22) also points to the next line. Branch to step 2.
10. RTS.

Memory Allocations and Moves

Several routines are used in checking for available memory or stack space and in copying sections of memory from one location to another. These routines are grouped in this section.

Check Available Free Space: Determines whether sufficient space exists in the free area to contain an area that has been requested to be allocated.

Check Available Stack Space: Expression evaluation, FOR, and GOSUB all check to see if enough stack space exists before they execute.

Create Space for Variable or Line: Whenever a variable or BASIC line needs to be added to the program text area or to the variable area, this routine checks to see that enough space in the free area exists.

Move Memory: This routine copies memory contents from one location to another memory location. The routines that call this all move memory from a lower address to a higher address. However, the routine also works for moving memory from a higher address to a lower address as long as there is no overlap in the move. This routine can be called from a machine language program to do memory moves.

Garbage Collection: Whenever routines detect no available free space or whenever FRE is executed, garbage collection moves all strings in the active string area that are actually active to the top of memory, and sets the pointer to the active strings to the bottom of the last currently active string. This garbage collection routine makes passes over the temporary string descriptor stack, the scalar variable area, and the array variable area. With each pass it takes the highest string that has not already been processed and moves it to the bottom of the active string area. When no more active strings exist, the garbage collection routine exits.

Allocate Space for String: This routine allocates an area below the end of the active strings, setting a pointer to this area in (35) to allow the copy routines to copy a string to this area.

Copy String from Descriptor to Allocated Area and Copy String from (22) to Allocated Area: These routines are used by concatenation, string variable creation, and the string operators LEFT$, MID$, RIGHT$, and CHAR$ to copy a string to the area allocated for it.

Check Available Free Space
A/C408–A/C434

Called by:
JSR at A/C3B8 Create Space for New Line or Variable; JSR at B/D264 Create Array Descriptor; JSR at B/D2B9 Create Array; JSR at E426/E408 Display Start-Up Message.

Entry Conditions:
(33) = pointer to the top of available free space.
Accumulator = low-order, Y-register = high-order, address of the end address of the area to be allocated.

Exit Conditions:
If there is no available free space, display OUT OF MEMORY.

Operation:
1. Compare the Y-register to 34, the high-order bytes of the area to be allocated compared to the end of available free space.
2. If the Y-register < 34, space is available, so branch to step 8.
3. If the Y-register > 34, branch to step 5.
4. If the Y-register = 34, compare the accumulator to 33, comparing the low-order bytes. If the accumulator < 33, space is available, so branch to step 8.
5. Since the accumulator and the Y-register >= (33), no available space exists.
6. JSR B/D526 to do garbage collection.
7. Again compare the accumulator and the Y-register to (33). If the accumulator and the Y-register < (33), garbage collection has reclaimed enough available free space, so branch to step 8. If not, display OUT OF MEMORY error message.
8. RTS.

Check Available Stack Space A/C3FB–A/C407

Called by:
JSR at A/CDAE Expression Evaluation; JSR at A/C885 GOSUB; JSR at A/C757 FOR.

Entry Conditions:
Accumulator = Number of bytes of stack space needed divided by two.

Exit Conditions:
OUT OF MEMORY error if not enough space.

Operation:

1. Multiply the accumulator by 2 by doing ASL.
2. Add 62. Besides the asked-for space, 62 bytes of space are also reserved.
3. If carry is set as a result of the add in step 2, display OUT OF MEMORY error. Carry is set if the accumulator >= $57 (decimal 87) at entry. The OUT OF MEMORY error message is not a very accurate description: OUT OF STACK SPACE would be more descriptive.
4. 22 = accumulator.
5. X-register = stack pointer.
6. If the X-register <= 22, display OUT OF MEMORY. For example, if the stack pointer = $40 and 22 = $44, not enough stack space exists to store the accumulator * 2 + 62 bytes. The stack occupies the area from 0100 to 01FF. When a new entry on the stack is made, it is placed at the location pointed to by the stack pointer and then the stack pointer is decremented, so new entries are made at successively lower memory locations. Entries on the stack are not actually moved; rather the stack pointer moves.
7. If the X-register > 22, adequate stack space exists. RTS with carry set.

Create Space for Variable or Line A/C3B8–A/C3BE

Called by:
JSR at B/D15D Create Variable; JSR at A/C50A Replace/Store BASIC Line.

Entry Conditions:
Accumulator = low-order, Y-register = high-order, address of the end address of the area to be allocated.

Operation:

1. JSR A/C408 to see if enough available free space exists for this variable or line. If not, display OUT OF MEMORY.
2. 31 = accumulator, the low-order byte of the end of the area to be allocated.
3. 32 = Y-register, the high-order byte of the end of the area to be allocated. (31) is the pointer to the start of the BASIC free area.
4. Fall through to A/C3BF to move memory to allow space for this line or variable.

Move Memory
A/C3BF-A/C3FA

Called by:
JSR at B/D628 Garbage Collect a String; Fall through from A/C3BD Create Space for Variable or Line.

Entry Conditions:
(5F) = pointer to start of area to be moved.
(5A) = pointer to the end of area to be moved plus one.
(58) = pointer to the end of the destination move area plus one.

Exit Conditions:
(58) = start of destination move.
This move can be used by your machine language routine as long as the move is in an upward direction. It will also work in a downward memory move as long as the source area and the destination area do not overlap.

Operation:

1. Subtract 5F, the low-order byte of the start of area to be moved, from 5A, the low-order byte of the end of the area to be moved. Store the result in the Y-register, the low-order number of bytes to be moved. Also store the result in 22, which denotes the number of bytes to move within a partial page.
2. Subtract 60, the high-order byte of the start of the area to be moved, from 5B, the high-order byte of the end of the area to be moved. Store the result in the X-register, the

high-order number of bytes to be moved (also termed *number of pages*). Also, INX to get the correct number of pages because of BNE test for end of move.

3. If the Y-register = $00, an exact number of pages are to be moved and the partial page move section is not needed, so branch to step 8.
4. Bytes within page move: (5A), the end of the source address minus 22, the number of bytes to move in the partial page.
5. (58) = (58), the end of the destination move area, minus 22, the number of bytes to move in the partial page.
6. LDA (5A),Y and STA (58),Y. The move thus proceeds from the end of this partial page to the beginning of the page.
7. DEY and loop to step 6 until the Y-register = $00.
8. Full page move: DEC 5B and DEC 59. After a page has been moved (or if there is no partial page to move), decrement the page bytes of the source and destination pointers.
9. Decrement the number of pages to be moved in the X-register.
10. If the X-register is not zero, branch to step 7.
11. RTS.

Garbage Collection
B/D526–B/D5BC

Called by:
JSR at A/C414 Check Available Free Space; JSR at B/D384 FRE; JSR at B/D51C Allocate Space for String; Alternate B/D52A; JMP at B/D63A Garbage Collect a String.

Operation:

1. X-register = 37 and accumulator = 38. (37) = pointer to the end of BASIC memory.
2. B/D52A: 33 = X-register and 34 = accumulator. (33) is the pointer to the bottom of the active strings.
3. (4E), the string descriptor for the highest string found so far, is set to $0000. If (4E) is still $0000 when this pass through the temporary string descriptor stack, scalar variables, and array variables finishes, then garbage collection is complete.
4. Accumulator = 31 and X-register = 32. (31) is the pointer to the start of the free area.

5. 5F = accumulator and 60 = X-register. (5F) = pointer to the string for the string descriptor.
6. (22) = $0019, pointing to the start of the temporary string descriptor stack.
7. Compare the accumulator value of $19 (or $1C or $1F or $22) to 16, the pointer to the next available slot in the temporary string descriptor stack. When 16 = $22, the temporary string descriptor stack is full. If the accumulator = 16, there are no temporary string descriptors, or all temporary string descriptors have been processed. Branch to step 9.
8. Since there are temporary string descriptors, JSR B/D5C7 to see if this temporary string descriptor points to a string which is the highest string in string memory. The first time through this routine, the string will be the highest unless the string is located in the program text area. Set (5F) as the pointer to the highest string. (22) = pointer to this string descriptor plus the length of this string descriptor, thus pointing to past this string descriptor. The accumulator also holds the pointer to the next string descriptor. Loop to step 7 until all temporary string descriptors have been processed.
9. All temporary string descriptors have now been processed, so reset 53, the string descriptor length, to $07 since the scalar string variables have a seven-byte descriptor.
10. Store (2D), the pointer to the start of variables, into (22), using the accumulator and the X-register.
11. Compare the accumulator and X-register to (2F), the pointer to the start of arrays. If equal, branch to step 13.
12. If not at the start of arrays, JSR B/D5BD to see if this is a string descriptor. If it is, see if the string pointed to is > (5F), the pointer to the currently highest string, and < (33), the pointer to the bottom of the active strings. This pointer to the bottom of the active strings moves downward as each garbage collected string has been moved back into the active string area. Thus, a string won't be moved to the active string area twice. If this string descriptor does point to a string that is > (5F) and < (33), reset (5F) to point to this highest string, and (4E) to point to the descriptor for this highest string.

    Also increase (22) by seven to point to the next descriptor in the variable area. Loop back to step 11.

13. 58 = accumulator and 59 = X-register. (58) is a movable pointer to the arrays.
14. Set 53, the string descriptor length, to $03, since array string variables have a string descriptor length of three.
15. Accumulator = 58 and X-register = 59, getting the pointer to the next array.
16. Compare the accumulator and X-register to (31), the pointer to the start of the free area. If equal, all arrays have been processed, so JMP A/D606 to move the string pointed to by the garbage collection routine, whose descriptor is (4E), to the bottom of the active string area. If (4E) is $0000, no more active strings have been found, and garbage collection ends.
17. If not equal, then 22 = accumulator and 23 = X-register.
18. X-register = first character of the array (first letter of array name) loaded from (22), 0.
19. The accumulator is loaded from the second character of the array (second letter of array name) and pushed onto the stack.
20. Add the length of this array to (58), and store the result back into (58), the pointer to the next array descriptor to be checked.
21. Array descriptors for string arrays have the first letter of the name with the high-order bit off, and the second letter of the name with the high-order bit on. If either of these conditions is not true for this array, it is not a string array and can be bypassed. Branch to step 15.
22. Accumulator = number of dimensions of the array from the array descriptor. Then compute the offset to the actual start of the array data by doing the following: Since each dimension has a two-byte number-of-elements field, do ASL to multiply the number of dimensions times two. Then add five to account for the two-byte name, the two-byte array length, and the one-byte number of dimensions. Add the offset to (22) and also leave the result in the accumulator and the X-register.
23. If the accumulator and the X-register are = (58), the pointer to the next array descriptor, this array has been completely processed, so branch to step 16.
24. If the end of this array has not yet been reached, JSR B/D5C7 to see if the string descriptor for this array string variable is < (33), the pointer to the bottom of active

strings, and >= (5F), the pointer to the highest current string. If this string descriptor does point to a string that is > (5F) and < (33), reset (5F) to point to this highest string, and (4E) to point to the descriptor for this highest string.

Also leave the accumulator and the X-register pointing to the next string descriptor in this string array. Then branch to step 23.

Check for Variable String Descriptor B/D5BD-B/D5C6

Called by:
JSR at B/D561 Garbage Collection.

Operation:
1. See if this seven-byte descriptor has the variable name for a string variable by testing the first two bytes of the descriptor. A string variable has the first byte with the high-order bit off and the second byte with the high-order bit on. If either of these tests for high-order bit status of these two bytes fails, branch to B/D5F6 to skip this descriptor. The next descriptor to check is obtained by adding 7 to the pointer to this variable descriptor.
2. If this is a string descriptor, fall through to B/D5C7 to see if the string pointed to is the highest eligible string.

Determine Highest Eligible String B/D5C7-B/D605

Called by:
JSR at B/D5B8 Garbage Collection; JSR at B/D548 Garbage Collection.

Operation:
1. If the length of this string from the string descriptor is zero, branch to step 7.
2. Accumulator and X-register = pointer to this string, loaded from the string descriptor.
3. If the pointer to the string is >= (33), the pointer to the bottom of the active strings, branch to step 7 since this string has already been processed for garbage collection and moved to the active string area.
4. If the pointer to this string is < (5F), the pointer to the

highest string found so far in this pass, branch to step 7 since this string is not the highest string this pass.

5. If this string pointer passes both tests (step 3 and step 4), this string is the highest string so far this pass. Reset (5E) to point to this string, and reset (4E) to point to the descriptor for this string.
6. Save the size of this string descriptor in 55.
7. B/D5F6: Add 53, which contains the length of this string descriptor, to (22), the pointer to the current string descriptor, thus pointing (22) to the next string descriptor.
8. Set the Y-register to $00 and RTS.

Move String Flagged by Garbage Collection to Active String Area
B/D606–B/D63C

Called by: JMP at B/D57A Garbage Collection.

Operation:

1. If (4E), the pointer to the descriptor for the highest string, is $0000, branch to B/D601 to exit garbage collection with the X-register = 23, the page of the last descriptor scanned, and the Y-register = $00.
2. Accumulator = 55, which holds the length of this string descriptor. This length is either 3 (0000 0011) or 7 (0000 0111).
3. AND $04 to leave the accumulator = $00 if three-byte descriptor.
4. LSR so accumulator = 2 if seven-byte descriptor.
5. Save the accumulator in the Y-register and in 55.
6. LDA (4E),Y to get the length of this string. Y = 0 if three-byte descriptor for temporary string of array string variables. Y = 2 if seven-byte descriptor for scalar string variables.
7. Add the length of the string to (5F), the pointer to the beginning of the string, and store the result in (5A), which now points to the end of the string + 1.
8. Copy (33), the pointer to the end of free space (the bottom of active strings −1), to (58), which is now set to be the end destination of the following move.
9. JSR A/C3BF to copy the string from (5A), the end of the string, to (58), the end of free space, working backward

from (5A) and (58) for the length of the string. At exit, (58) points to the start of the copied string.

10. Now store (58) into the pointer bytes of the string descriptor pointed to by (4E).
11. JMP to B/D52A, step 2 of Garbage Collection, to store (58) into (33), the pointer to the bottom of active strings.

Allocate Space for String
B/D4F4–B/D525

Called by: JSR at B/D47D Create Space for String.

Entry Conditions:
Accumulator = length of string to be allocated.

Operation:

1. LSR 0F to clear the error flag.
2. Push the length of the string to be allocated onto the stack.
3. Subtract the length of the string from the pointer to the bottom of the active strings, storing the low-order result in the Y-register and the high-order result in the accumulator.
4. Compare this result to the pointer to start of free space (31). If the result is < (31), thus overlapping into the area for arrays, branch to step 8.
5. If there's no overlap, reset (33), the pointer to the bottom of active strings, from the Y-register and the accumulator.
6. Also put this value into (35), the pointer to the most current string to be added or moved.
7. RTS with X-register = high-order byte of pointer, Y-register = low-order byte of pointer, and PLA to restore the length of the string in the accumulator.
8. If 0F has its high-order bit on, indicating garbage collection has already been done once, branch to B/D4D2 to display OUT OF MEMORY.
9. JSR B/D526 to perform garbage collection of strings.
10. Set the high-order bit on for 0F to indicate garbage collection is completed.
11. PLA to restore the length of this string and branch to step 2 to try one more time.

Copy String Pointed to by Descriptor to Allocated Area B/D67A–B/D68B

Called by:
JSR at B/D660 Concatenate String; JSR at A/CA61 LET; Alternate B/D688; JSR at B/D4C7 Set Up a String Variable.

Entry Conditions:
(6F) = pointer to string descriptor for string to be copied.
(35) = pointer to allocated area.

Operation:
1. Using (6F), the pointer to the string descriptor, move the length of the string from the string descriptor to the accumulator, and the pointer to the string from the descriptor to the X-register and the Y-register.
2. B/D688: If called from B/D4C7, the X-register and Y-register point to the string in page 0 or page 2 that is to be copied to string memory. (22) = X-register and Y-register.
3. Fall through to B/D68C to copy the string from (22) to the area allocated for the string at (35).

Copy String from (22) to (35) for Length of Accumulator B/D68C–B/D6A2

Called by:
JSR at B/D66A Concatenate String; JSR at B/D726 Common Routine for LEFT$, MID$, RIGHT$, CHR$.

Entry Conditions:
(22) = pointer to source string.
(35) = pointer to allocated area.
Accumulator = length of string.

Operation:
1. Y-register = accumulator, the length of the string.
2. If the accumulator = $00, branch to step 9.
3. Push the length of the string onto the stack.
4. DEY.
5. Get the next byte of the source string pointed to by (22) indexed by the Y-register.

6. Store this byte into the area allocated at (35), indexed by the Y-register.
7. If the Y-register is not zero, branch to step 4.
8. After all bytes of the string have been moved, pull the length of the string moved from the stack.
9. Add the number of bytes moved to (35), the pointer to the area allocated for this string. Thus (35) now points one byte past the end of the string. If the number of bytes copied was $00, (35) was not modified.
10. RTS.

Pointer Resets

NEW, CLR, and RESTORE are BASIC statements that can be used to reset certain pointers.

NEW resets pointers to clear both a stored program and any variables from the BASIC storage area. CLR clears variables but does not erase the program. RESTORE resets the pointer to DATA statements to the beginning of the program text area. While pointers are reset by NEW and CLR, no data is actually erased and thus can be reclaimed.

Certain other routines also reset pointers. The text pointer is reset to the start of the program text area minus one by Reset Text Pointer to Start of Program Text minus one. Reset stack pointer and other variables is performed during a BASIC warm start.

NEW
A/C642–A/C658

Called by: Execute Current BASIC statement.

Operation:

1. If any parameters follow NEW, display SYNTAX ERROR and go to the Main BASIC Loop.
2. Store $00 into the first two bytes of the program text area, into the link field of the first line. Since a link field of $0000 signifies the end of a program, this clears any program from memory.
3. Set (2B), the start of variables, to (2B) + 2, thus pointing past the two zero-link bytes.
4. Fall through to C659 to reset (7A), the text pointer, to the start of the BASIC program text area minus one. Then do CLR.

Reset Text Pointer to Start of BASIC Program Minus One and Do CLR
A/C659–A/C65D

Called by:
JSR at A/C4ED Replace/Store BASIC Line; JSR at A/C52A Replace/Store BASIC Line; JSR at A/C87A RUN.

Operation:
1. JSR A/C68E to reset the text pointer (7A) to the start of the BASIC program text area − 1.
2. Accumulator = $00.
3. Fall through to A/C65E to do CLR.

CLR
A/C65E-A/C679

Called by:
Execute Statement; Fall through from Reset Text Pointer or NEW; Alternate C660: JSR at A/C87D RUN; Alternate C663: JMP at E101/E0FE Close RS-232.

Operation:
1. If any parameters follow CLR, then RTS; a SYNTAX ERROR will result.
2. A/C660: JSR FFE7 to the Kernal CLALL routine to close channels and files routine. The routine sets the number of open files to zero, clears serial bus activity, sets the current input device to be the keyboard, and sets the current output device to be the screen.
3. A/C663: Store (37), the pointer to the end of BASIC memory, into (33), the pointer to the bottom of active strings. All active strings are removed by this reset.
4. Store (2D), the pointer to the start of BASIC variables, into (2F), the pointer to the start of arrays. This effectively clears all variables.
5. Also store (2D), the pointer to the start of BASIC variables, into (31), the pointer to the start of the free area. This clears all arrays from the program because the start of the free area now is the end of the program text area + 1.
6. JSR A/C81D to restore the pointer to the DATA statements to be the pointer to the start of the BASIC program.
7. Fall through to A/C67A to reset the stack pointer to $FA, reset the temporary string descriptor stack pointer, disallow CONT, and set 10 to $00.

Reset Stack Pointer and Other Variables
A/C67A-A/C68D

Called by:
Fall through from Reset Text Pointer, NEW, or CLR; JSR at

E382/E46E Warm Start; JSR at A/C462 Error Message Handler.

Operation:

1. Reset 16, the temporary string descriptor stack pointer to $19, thus pointing it to the first location in the temporary string descriptor stack.
2. Pull off the current return address of this routine and save in the accumulator and the Y-register.
3. Set the X-register to $FA and TXS, resetting the stack pointer. This $FA will be the value of the stack pointer after exiting this routine.
4. Push this routine's return address back onto the stack from the accumulator and Y-register.
5. Set 3E to $00, to disallow CONT.
6. Set 10 to $00 to clear the flag that allows (in an expression.
7. RTS.

Reset Text Pointer to Start of Program Text Minus One A/C68E-A/C69B

Called by:

JSR at E1B5 LOAD; JSR at A/C659 Reset Text Pointer and Do CLR.

Operation:

1. Load (2B), the pointer to the start of the BASIC text area.
2. Decrement this value.
3. Store the value into (7A), the text pointer, which thus points to the start of the BASIC program text area minus one.

RESTORE
A/C81D-A/C82B

Called by:

Execute Current BASIC Statement; JSR at A/C677 CLR.

Operation:

Store (2B), the pointer to the start of the tokenized program, minus one into (41), the pointer to the location for reading DATA statements. This points (41) to the initial $00 that precedes the start of the program. When READ finds (41) pointing to $00, it starts its search for DATA statements.

Expression Evaluation

An expression can consist of anything from one variable or constant to a succession of operators, variables, constants, functions, and expressions within parentheses. For example, in the statement X=2, 2 is an expression. In the statement Y=3*5−X/(SIN(10+C)), 3*5−X/(SIN(10+C))is an expression. In this second example, the (10+C) is also an expression for the SIN function. The resulting expression value can be assigned to a variable in a LET statement.

BASIC keywords also use expression values in their execution. Keywords that use arithmetic expressions include ABS, AND, ATN, CHR$, CLOSE, CMD, COS, DEF FN, DIM, EXP, FOR, GET, GET#, IF, INPUT#, INT, LOAD, LOG, MID$, NOT, ON, OPEN, OR, PEEK, POKE, PRINT, PRINT#, RIGHT$, RND, SAVE, SGN, SIN, SPC(, SQR, STR$, SYS, TAB(, USR, AND WAIT. Keywords that use string expressions include ASC, CLOSE, LEFT$, LEN, LOAD, MID$, OPEN, PRINT, PRINT#, RIGHT$, SAVE, and VAL.

When expression evaluation is finished, location 0D = $FF if the expression is a string expression or $00 if it is a numeric expression. If it is a string expression, (64) holds a pointer to the string descriptor for this string, with the string descriptor located either in the program text area or in the temporary string descriptor stack. (64),0 holds the length of the string; (64),1 has the low-order byte of the pointer to the string; and (64),2 has the high-order byte of the pointer to the string.

If this is a numeric expression, then Floating Point Accumulator 1 (FAC1) holds the result in normalized floating point format at exit.

Although one can also consider a separate category for logical expressions, expressions using the relational operators (<,<=,=,<>,>,>=), these logical expressions evaluate to FAC1 with a numeric value. This value is 0 if the expression is false and $FF (or −1) if the expression is true.

It is a common convention in arithmetic operations that certain operations have a higher priority than other operations. For example, in the expression 2+3*7, the result is 23 rather than 35 because multiplication is performed before addition. Also, expressions are scanned by the CHRGET routine in a left-to-right direction. Taken together, this scanning from left-to-right and operator-precedence rules present a problem when an operator (operators are such things as +, −, *, /) that has a higher priority is located to the right of an operator of lower priority.

If operators were applied in CHRGET's strict left-to-right order, the lower priority one would be applied incorrectly. Thus BASIC has rules it follows in the expression scan about evaluation of operators. The following steps describe the expression evaluation for numeric (or logical) expressions. For string expressions the only valid operator is + for concatenation. After concatenation, (64) holds the pointer to the string descriptor in the temporary string descriptor stack for this string. These expression evaluation rules are:

1. Put an initial operator priority of $00 on the stack, thus forcing a priority that is lower than all other operators.
2. Scan the expression from left-to-right. As the expression is scanned, the sequence of operand (a constant or a variable) followed by operator followed by operand ... is expected. However, the monadic operators, negate and NOT, are acceptable when an operand is expected. The monadic plus is acceptable but is just skipped over and not applied. When an operand is being scanned, CHRGET ends by retrieving the operator that follows the operand. When an operator is being scanned, CHRGOT is called to retrieve this operator that was returned by the CHRGET in the operand scan. If the operator retrieved by CHRGOT is not a valid operator, the expression is considered totally scanned, and 4B, the current operator index, is set to $FF to indicate the end of the expression. On this end-of-expression, go to step 5 in these rules.

   If CHRGOT returns a (, then scan the expression should now evaluate the expression within the parentheses. It does this by recursively calling itself, jumping to step 1 to put the initial operator priority of $00 on the stack. When the final) is found and returned by CHRGOT, the parenthesized expression is considered complete and a branch to step 5 with 4B set to $FF is made.

3. If an operand (a constant or a variable) is found, the value of this operand is left in FAC1.
4. If an operator is found, the priority of the current operator is compared to the priority of the last stacked operator.

   If the stacked operator priority is greater than or equal to the current operator priority, remove the stacked left operand from the stack and move this value to FAC2. Then pull the address of this operator from the stack and jump to that routine, leaving the result of the operation in FAC1.

   If the stacked operator priority is less than the current priority, push the following onto the stack: the address of the current operator, FAC1 (the left operand preceding this operator), the code for any check of relational operators, and the priority of this operator. Then continue scanning the expression by recursively calling step 2. By recursively jumping to itself when the end-of-expression is detected, the values that are on the stack are pulled off in correct sequence. Another fact about operator precedence is that the first operator that is scanned will push its values onto the stack as the initial operator value of $00 has to be lower.
5. Once $FF has been set as the index of the current operator, the end of the expression has been reached. Pull off operator priorities the left operand for this operation and move to FAC2, then pull off the address of the operation and leave the result of the operation in FAC1.

   Continue pulling off operator priorities, left operands, and addresses of the operation until the $00 priority operator is pulled off. When this is pulled off, expression evaluation is complete. If evaluating within parentheses, the expression within parentheses has been evaluated. The final result of the expression evaluation is in FAC1.

The operator priorities are from highest to lowest: ↑,$7F; negate (monadic −), $7D; *, $7B; /, $7B; +, $79; −, $79; relational operators (<,<=,=,<>,>,>=), $64; NOT, $5A; AND, $50; OR, $46.

The effect of applying the operator priorities in evaluating expressions is that BASIC actually evaluates expressions as if they were in Reverse Polish Notation. For example, the expression −5*3+2*(7−1) can be represented in Reverse Polish Notation as 5 − 3 * 2 7 1 − * +. In Polish Notation execution, you continually take the leftmost operator and replace it and its operands by the result. Thus the example above goes through these steps: −5 3 * 2 7 1 − * + ; −15 2 7 1 - * +; −15 2 6 * + ; −15 12 +; −3. These steps are exactly the steps that occur in BASIC's expression evaluation as the following example of expression evaluation shows:

Expression: −5*3+2*(7−1)

1. Push the initial operator priority of $00 onto the stack.

   Stack: $00

2. Scan for operand. Monadic operator negate is found which is acceptable. Compare the priority of NEGATE, $7D, to the stacked operator priority of $00. Since the stacked priority is lower, push the NEGATE routine address (actually the address −1 since RTS adds one when it pulls the address off the stack), push FAC1 (which has a dummy value now), push 4D <=> test (the code which indicates if a relational test is to be made), and push the priority of NEGATE.

   Stack: $00, + Address, −15, $00 <=>, $79 priority of +, * Address, 2, $00 <=>, $7B priority of *,$00.

3. Scan for operand. Operand found, so FAC1 = 5. Accumulator holds the * operator from final CHRGET.
4. Scan for operator using CHRGOT, retrieving the *. Compare the priority of *, $79, to the stacked NEGATE priority of $7D. Since NEGATE priority is higher, execute NEGATE. However, also save the index into the operator table for * in 4B so that we can pick up from here when NEGATE finishes. To execute NEGATE, first the priority has already been pulled off the stack for the compare. Next pull off the <=> code and discard it since it is $00. Then pull off the stored left operand FAC1 value of 0 and move to FAC2. This moving of the left operand to FAC2 from the stack is always done when an operator is pulled from the stack and executed. Finally, pull the address of NEGATE from the stack and the next RTS will execute NEGATE, making the current value of FAC1 = −5.

   Stack: $00
   FAC1: −5
   4B: Index for *

5. The RTS from NEGATE returns to retrieve the saved index of * from 4B. Also, the stacked operator, $00, is pulled from the stack. Since $00 is lower in priority, push $00

onto the stack and then push the * routine address, push FAC1 (−5), push 4D ($00) <=> test (the code which indicates if a relational test is to be made), and push the priority of *, $7B.

Stack: $00, * Routine Address, FAC1 (−5), $00 for <=>, $7B * Priority.

6. Scan for operand and return with FAC1 = 3 and (7A) pointing to +.

   FAC1: 3

7. Scan for operator by using CHRGOT to retrieve (7A), the +.
8. Compare priority of +, $79, to priority of last stacked operator, *, $7B. Since * has higher priority, execute it. Save the index into the operator table for + in 4B so that we can pick up from here when * finishes. To execute *, pull off the <=> code and discard it since it is $00. Then pull off the stored left operand FAC1 value of −5 and move to FAC2. Finally, pull the address of * from the stack and the next RTS will execute *, leaving FAC1 = −15.

   Stack: $00
   FAC1: −15
   4B: Index for +

9. The RTS from * returns to retrieve the saved index of + from 4B. Also, the stacked operator, $00, is pulled from the stack. Since $00 is lower in priority, push $00 onto the stack and then push + routine address, push FAC1 (−15), push 4D ($00) <=> test, and push the priority of +, $79.

   Stack: $00, + Address, −15, $00 <=>, priority of +.

10. Scan for operand and return with FAC1 = 2 and (7A) pointing to *.

    FAC1: 2

11. Scan for operator by using CHRGOT to retrieve (7A), the *.
12. Compare the priority of the stacked operator +, $79, to the priority of the current operator *, $7B. Since + is lower in priority, + priority $79 is pushed onto the stack and then push * routine address, push FAC1 (2), push 4D ($00) <=> test, and push the priority of *, $7B.
13. Scan for the operand. However, a (is found, so evaluate the expression with parenthesis. Push the initial priority of $00 for this expression onto the stack.

Stack: $00, + Address, −15, $00 <=>, $79 priority of +, * Address, 2, $00 <=>, $7B priority of *,$00.

14. Scan for the operand. FAC1 loaded with 7. (7A) points to −.

    FAC1: 7.

15. Scan for the operator, returning the − pointed to by (7A).
16. Compare the priority of the stacked operator $00 to the priority of −, $79. Since $00 is lower in priority, $00 is pushed onto the stack and then push − routine address, push FAC1 (7), push 4D ($00) <=>, test, and push the priority of −, $79.

    Stack: $00, + Address, −15, $00 <=>, $7B priority of +, *, Address, 2, $00 <=>, $7B priority of *, $00, − Address, 7, $00 <=>, $79 priority of −.

17. Scan for the operand. Return with FAC1 = 1 and (7A) pointing to).

    FAC1: 1

18. Scan for the operator by retrieving the character pointed to by (7A). The) is retrieved, but) is not a valid operator. Thus, set 4B, the operator index, to $FF to indicate the end of this expression. Pull the priority of the last stacked operator from the stack. As long as we're pulling a nonzero value when 4B is set to $FF, we continue to pull operations off the stack, unwinding the values placed there during recursion. When zero value is pulled, all operators on the stack for this expression (which in this case is within parentheses) are done.
19. Pull off the $00 <=> code, pull the 7 left operand and move to FAC2. Pull the address of the − routine and RTS to execute −, subtracting FAC1 (1) from FAC2 (7) and leaving the result in FAC1.

    Stack: $00, + Address, −15, $00 <=>, $79 priority of +, * Address, 2, $00 <=>, $7B priority of *, $00.
    FAC1: 6

20. Pull the next stacked operator priority. Since this priority is $00, the evaluation of this expression within parentheses is complete. However, now syntax check for the ending right parenthesis by scanning for). This scan ends with CHRGET retrieving the character following the), which in this statement is now the end-of-statement $00 byte.

Stack: $00, + Address, −15, $00 <=>, $79 priority of +, * Address, 2, $00 <=>, $7B priority of *.
FAC1: 6

21. Because the evaluation of the expression within parentheses was done from the JSR in step 7 of the Main Expression Evaluation, now return to step 8. In step 9, call CHRGOT to retrieve the last character scanned by CHRGET, the $00 end-of-statement byte. Since this $00 value is not a value operator, again set 4B to $FF to indicate that the end of the statement has been reached.
22. Pull the priority of the last stacked operator from the stack, which is for *. Since this is a nonzero priority, continue. Pull $00 <=> code and discard, pull left operand 2 and move to FAC2. Pull the address of the * routine and execute *, multiplying FAC2 (2) by FAC1 (6), leaving the result of 12 in FAC1.

    Stack: $00, + Address, −15 $00 <=>, $79 priority of +.
    FAC1: 12
23. Since 4B is still $FF, pull the priority of the last stacked operator from the stack, which is for +. Since this is a nonzero priority, continue. Pull $00 <=> code and discard, pull the left operand −15 and move to FAC2. Pull the address of the + routine and execute +, adding FAC2 (−15) to FAC1 (12), leaving the result of −3 in FAC1.

    Stack: $00.
    FAC1: −3.
24. Since 4B is still $FF, pull the priority of the last stacked operator from the stack, which is the $00 priority that is the initial value. When this $00 priority is pulled off and 4B is $FF, the expression evaluation is complete.

    Stack:
    FAC1: −3.

Main Expression Evaluation A/CD9E–A/CE1F

Called by:
JSR at A/CAB5 PRINT; JSR at A/C928 IF; JSR at A/CEF4 Evaluate Expression Within Parentheses; JSR at A/CFB4 Evaluate First Expression for RIGHT$, LEFT$, MID$; JSR at B/

D1B5 Convert Expression to Signed Integer; JSR at A/CD8A Type Check; JSR at E257/E254 OPEN-CLOSE Evaluate String Expression; JSR at A/C9B7 LET; Alternate A/CDA9: JMP at A/CE2D Recursive Expression Evaluation; Alternate A/CDB8: JMP at B/D677 String Concatenation.

Exit Conditions:
0D = $FF if string expression, = $00 if numeric expression. If it's a numeric expression, FAC1 holds the numeric value in normalized floating point format. If it's a string expression, 64 has a pointer to the string descriptor for this string.

Operation:

1. Decrement (7A), the text pointer, so that when CHRGET is first used it will retrieve the first character of this expression.
2. Set the X-register = $00, and skip step 3 by using a bit instruction to go to step 4.
3. PHA, saving the code from 4D that indicates whether a relational operator is to be performed.
4. Push the X-register onto the stack. The X-register holds the operator priority, either the initial priority of $00 or the priority of an operator that is being pushed onto the stack.
5. Accumulator = $01 to indicate that two more bytes of stack space are needed.
6. JSR A/C3FB to check that space exists on the stack for these two bytes and the reserved 62 bytes. If not, display OUT OF MEMORY.
7. JSR A/CE83 to evaluate the next operand and to return the pointer to the next operator. This routine at A/CE83 does one of the following: converts an ASCII numeric expression to a number in FAC1, converts a variable to a number in FAC1, converts PI to its numeric value in FAC1, allocates space for a string, and copies the expression within quotes to the string with (64) pointing to the descriptor for the string, evaluates the string variable and returns the pointer to the descriptor in (64), handles monadic operators negate and NOT by recursively executing the rest of the expression, executes FN or any of the other BASIC functions and returns the numeric value in FAC1 or the pointer to the string descriptor in (64), and if none of the above, then tries to evaluate the expression within parentheses. If none of these conditions exist, display SYNTAX ERROR.

Also return with (7A) pointing to the first nonspace character past the operand (or past the monadic operator).

8. Set 4D = $00 to clear the flag for the relational operators.
9. A/CDB8: JMP here after string concatenation.

   Execute CHRGOT to retrieve into the accumulator the character following the operand. This character should be an operator.
10. A/CDBB: Subtract $B1, the token value for >, from the accumulator.
11. If this token is < $B1, branch to step 23.
12. Compare the resulting difference to $03. If the token was $B1 for >, the accumulator = $00. If the token was $B2 for =, the accumulator = $01. If the token was $B3 for >, the accumulator = $02.
13. If the difference is >= $03, this can't be the token for = or <, so branch to step 23.
14. Compare the accumulator to $01 to see if =, leaving the carry set if the token is = or <.
15. Rotate the accumulator left, thus the accumulator now = $00 for <, $03 if =, or $05 if >.
16. Exclusive OR with $01, leaving the accumulator = $01 for >, $02 for =, or $04 for <.
17. Exclusive OR with 4D, the previous value for the relational operator, which would turn the bit off in 4D if this is the second time this relational operator has been found in this scan sequence.
18. If after the Exclusive OR in step 17 the accumulator is now < 4D, this is the second time this same relational operator has been used between two operands.
19. If true, display SYNTAX ERROR and go to the Main BASIC Loop.
20. Store the accumulator, containing the new value for the relational operators found this scan, into 4D.
21. Call CHRGET to get the next character.
22. Jump to step 10 to repeat this check. Because the sequence of checking for the relational operators is not important, the following ways of writing the relational operators are all valid: <= or =<, >= or =>, <=>, <> or ><.
23. Branch here after relational operators have been evaluated or if there are no relational operators between these operands. X-register = 4D, the relational operator's value.
24. If the X-register is nonzero, branch to step 47.

25. If carry is set (from the BCS in step 13), this token or ASCII value is >= $B4. Since tokens >= $B4 represent functions, branch to A/CE58 to execute any stacked operator or to RTS if none is stacked. Also set 4B to $FF to force expression evaluation of the function argument.
26. If ASCII value of $B0 or less, then ADC $07. Remember that $B1 has already been subtracted from the token value; for example, accumulator = F9 for + token because $F9 + $B1 = $AA. Similarly, for OR $FF + $B1 = $B0. With the $07 added, the + token = $F9 + $07 = $00 with carry set. Since + is the lowest value of the math and logical operator tokens, any other math and logical operator tokens will also set the carry.
27. If the carry is clear, this ASCII value is not the token for +, −, *, /, ↑, AND, OR. Branch to A/CE58 to set the end-of-expression index of $FF into 4B and start pulling operators off the stack and executing them until the original operator priority of $00 is removed.
28. Carry is now set if it's a math or logical operator. ADC 0D, the flag which indicates whether the last operand evaluated was a string ($FF) or a numeric value ($00).

    For example, if it's a + operator and a string, then $00 from step 26 + carry whose value is $01 + $FF = $00. If it's a numeric operand, then $00 + $01 + $00 = $01. All other numeric and relational operators leave the accumulator results > $00 whether it's a string or a numeric value. Thus only the one valid string operator, + for concatenation, is possible to execute with string operands.
29. If the accumulator is nonzero, branch to step 31.
30. If it's a string and a + operator, concatenate strings by JMP B/D63D. String concatenation returns a pointer to the new string descriptor in (64) and returns to step 9 of this routine by a JMP.
31. The carry is clear now after step 29. ADC $FF to in effect subtract one from the accumulator, restoring the accumulator to the index from 0 to 6 since the ADC in step 28 added one.

    The possible indexes are now 0(+), 1(−), 2(*), 3(/), 4(↑), 5(AND), 6(OR).
32. Store the accumulator in 22.
33. ASL to multiply the index by 2.

34. Add 22 to the accumulator, thus resulting in a value three times the original index. This is the value needed to index into the Math Operation Dispatch Vector Table since each entry in the table for an operator consists of three bytes—the first one is the operator priority and the next two are the address −1 of the math operator routine. The Y-register gets the accumulator.
35. PLA to remove the previously stacked operator priority.
36. Compare this stacked priority to the current operator priority which is obtained by indexing into the Math Operation Dispatch Vector Table at A/C080 using the Y-register.

    Note that step 35 is branched to by the routine that detects relational operators. Thus the relational operators as well as the math and logical operators (AND/OR) are valid comparisons.
37. BCS to A/CE5D to step 4 of Execute Operator if the stacked operator has a priority that is equal to or higher than the current priority.

    If the stacked priority is lower than the current priority, continue with step 38.
38. JSR A/CD8D to display TYPE MISMATCH if the expression evaluated to a string expression.
39. Push the previously stacked priority back onto the stack.
40. A/CDFA: Recursive entry point for monadic operators negate and NOT.

    JSR A/CE20 to push the address of the operator that was just scanned and found higher in value than the previously stacked operator on the stack. Also push FAC1, which holds the left operand preceding this operator, onto the stack. Then LDA 4D, the relational operator test, and JMP A/CDA9 to step 3 of this routine. We have done a JSR here and never came back from the routine with an RTS but rather called again recursively.

    When an RTS in the Execute Operator occurs, that value that was pushed onto the stack for the math operators routine is executed. When the math operator routine does its RTS, control returns to step 41 since the JSR to A/CE20 that we do here puts a return address on the stack to step 41.
41. PLA to restore the priority of the last operator on the stack.
42. Y-register = 4B, to get the index of the math operator routine, which is = $FF if the end of the expression has been reached.

43. BPL to step 53 if it's not the end of the expression.
44. If it is the end of the expression, then X-register = accumulator, the last stacked operator priority.
45. If this priority is $00, we are done with this expression since the initial priority has been removed. RTS with the accumulator = 61, the FAC1 exponent, and with FAC1 containing the value of this numeric expression or with (64) containing a pointer to a string descriptor if it's a string expression.
46. If this priority is nonzero, branch to A/CE66 to step 7 of Execute Operator to apply the stacked operator.
47. A/CE07: Branch here when relational operators are detected: LSR 0D, the string/numeric flag, shifting a 1 into carry if it's a string.
48. Accumulator = X-register, getting the value from 4D.
49. Rotate the accumulator left and decrement (7A), the text pointer. After this rotation the accumulator has one of the following values (low-order nybble shown—high-order is 0000):

	<	=	>
Initially from 4D	**0100**	**0010**	**0001**
If string	**1001**	**0101**	**0011**
If numeric	**1000**	**0100**	**0010**

Later, during the comparison routine, an LSR uses the carry bit shifted out of bit 0 to determine whether the comparison is for strings or numerics.

50. Y-register = $1B, setting the index into the Math Operation Dispatch Vector Table.
51. Save the accumulator in 4B.
52. Branch to step 35.
53. Branch here from step 43 to compare the priority of the stacked operator to the priority of the current operator indexed by the Y-register.
54. If the stacked operator priority is higher, BCS A/CE66 to step 7 of Execute Operator to apply this operator.
55. If the stacked operator priority is lower, branch to step 39 to again recursively place this operator's priority, address, and FAC1 onto the stack.

Math, Logical, and Relational Operators Dispatch and Priority Table
A/C080-A/C09D

Used by: Expression Evaluation.
This table contains the priority of each operator followed by the address minus one of the routine for each operator. The address of the routine minus one is used because it is used as a return address for RTS which adds one to the address. The operator priority is used to determine whether an operator that appears previous to another operator should be executed or pushed onto the stack.

Operator	Location	Priority	Address−1
+	A/C080	$79	B/D869
−	A/C083	$79	B/D852
*	A/C086	$7B	B/DA2A
/	A/C089	$7B	B/DB11
↑	A/C08C	$7F	B/DF7A
AND	A/C08F	$50	A/CFE8
OR	A/C092	$46	A/CFE5
−(Monadic)	A/C095	$7D	B/DFB4
NOT	A/C098	$5A	A/CED3
Relational	A/C09B	$64	B/D015

Invoke Recursive Expression Evaluation
A/CE20-A/CE2F

Called by: JSR at A/CDFA Main Expression Evaluation.

Operation:

1. Using the Y-register to index the operator priority/dispatch table, push the low-order and high-order bytes of the address of the operator onto the stack. The address pushed onto the stack is actually one less than the actual address of the routine because RTS adds one to this address.
2. JSR A/CE33 to push the rounded value of FAC1 onto the stack. This represents the left operand that precedes this operator whose address was pushed onto the stack in step 1. Also set the X-register to the priority of this operator.
3. Accumulator = 4D to retrieve the code for the relational comparison test.
4. JMP A/CDA9 to step 3 of the Main Expression Evaluation. This is where the recursion routine calls itself. Since this

routine was called by a JSR from step 40 of the Main Expression Evaluation but didn't do an RTS, when the operator for this routine eventually executes and does an RTS, control will return to step 41 of the Main Expression Evaluation.

Call SYNTAX ERROR Message A/CE30–A/CE32

Called by:
BCC at A/CDCD Main Expression Evaluation when two of the same relational operators are detected between operands.

Operation:
JMP A/CF08 to display SYNTAX ERROR and go to the Main BASIC Loop.

Save Rounded FAC1 on Stack A/CE33–A/CE57

Called by:
JSR at A/CE28—Invoke Recursive Expression Evaluation; Alternate A/CE38: JSR at A/C7A2 FOR routine to handle STEP value; A/CE43: JMP at A/C788 FOR routine to handle TO value.

Operation:
1. Accumulator = 66, the sign of FAC1.
2. X-register = A/C080 indexed by the Y-register to retrieve the priority of the operator.
3. A/CE38: Y-register = accumulator.
4. Remove this routine's return address from the stack. Add one since that is what RTS would do. Now store this address in (22).
5. Accumulator = Y-register. Push the accumulator onto the stack, saving the sign of FAC1.
6. A/CE43: JSR B/DC1B to round FAC1.
7. Push FAC1—65, 64, 63, 62, 61— onto the stack.
8. JMP(0022) to return to the routine that called this routine without affecting the stack.

Prepare for Stacked Operator Execution and RTS to Stacked Operator Routine
A/CE58—A/CE82

Called by:
BCC at A/CDDF if expected operator is not a valid token; BCS at A/CDDB if expected operator has value >= $B4 (Possible Function); Alternate A/CE5B: BEQ at A/CE03 if 4B end-of-statement value of $FF and $00 priority pulled from stack; Alternate A/CE66: BCS at A/CE1C if Current Operator index in 4B is not $FF and stacked priority operator is higher than current; BNE at A/CE05 if 4B end-of-statement value of $FF and stacked operator priority is not zero.

Operation:

1. Y-register = $FF.
2. Pull the stacked operator priority from the stack.
3. A/CE5B: If the end-of-expression and stacked operator priority is $00, branch to step 11.
4. A/CE5D: Compare the operator priority to $64, the priority for relational operators. If equal, branch to step 6 with carry set, bypassing the check for strings since relational operators can have string operands.
5. JSR A/CD8D to display TYPE MISMATCH if string operands for any other operators. Also clear the carry.
6. Y-register = 4B, the index for this operator or $FF if end-of-expression.
7. PLA to remove the stacked relational comparison value.
8. LSR and put the accumulator into 12, which is used during the relational operator test to determine what comparisons to make. This LSR also puts a 1 into the carry if doing a string comparison or 0 if doing a numeric comparison.
   As a result:

```
          <          =          >
12 = 00000100  00000010  00000001
```

9. Remove the left operand from the stack and store it in FAC2. The right operand will still be in FAC1.
10. Set 6F from the Exclusive OR of the signs of the two floating point accumulators.
11. Accumulator = 61, the exponent for FAC1. This allows immediate tests for adding, multiplying, or dividing by zero when +, −, or / receive control.

12. RTS, which will execute an operator routine since the address of an operator routine is currently the last thing on the stack.

Evaluate Operand or Handle Monadic Operator A/CE83–A/CEA7 and A/CEAD–A/CED3 and A/CEE3–A/CEF0

Called by:
JSR at A/CDB1 Main Expression Evaluation; JSR at B/D643 String Concatenation.

Operation:

1. JMP (030A) with default vector of A/CE86.
2. A/CE86: Set 0D = $00 to indicate a numeric operand until told otherwise.
3. Call CHRGET to get the first character of the operand.
4. If this character is nonnumeric, BCS to step 6.
5. If the character is numeric, JMP B/DCF3 with carry clear to convert an ASCII string into a floating point number in FAC1. The RTS from B/DCFE also exits this routine.
6. If the character is nonnumeric, JSR B/D113 to see if the character is in the range A–Z and set carry if true.
7. If carry is clear, branch to step 9.
8. Since the character is A–Z, assume this is a variable. JMP A/CF28 to convert a numeric variable into a number in FAC1, or if a string variable to return the pointer to the string descriptor in 64. Exit from this routine upon the RTS.
9. Since it is not A–Z, compare the character to $FF, the ASCII code for PI. If it's not equal, branch to step 13.
10. Since it is PI, set the accumulator and the Y-register to point to A/CEA8 to reference the constant for PI.
11. JSR B/DBA2 to copy the value for PI to FAC1.
12. JMP CHRGET to retrieve the next character and then exit.
13. Since it's not PI and not A–Z, see if the first character is a period (.) and if true then branch to A/CE8F to convert the ASCII string into a number in FAC1.
14. If the first character is a hyphen (–), this is a monadic minus (negate). Branch to A/CF0D to Monadic Setup routine.
15. If the first/next character in the expression is plus (+), branch to step 3 to bypass this plus. Thus, the following

can be a valid operand: +++++3. Also, multiple monadic minuses are acceptable in expressions, giving valid expressions such as (3*2−+−−7).

16. If the first character in the expression is a quotation mark ("), continue with step 17. If not, branch to step 20.
17. Since the carry is set as a result of the equal comparison for a quotation mark in step 16, just ADC $00 to (7A) to move the pointer past the quote character to point to the first character of the string.
18. JSR B/D487 to scan for the end of the string, allocate space for the string in the active string region, and copy the string to the allocation space. On return, (64) points to the descriptor for this string on the temporary string descriptor stack.
19. JMP B/D7E2 to copy the pointer to the end of the string (71) to the text pointer (7A), thus leaving (7A) pointing to the final quotation mark.
20. Branch here from step 16 if it's not a string. See if the character is $A8, the token for NOT. If it's not $A8, branch to step 22.
21. Since the character is the token for NOT, set the Y-register to $18, setting the index value into the operator table for NOT and branch to A/CF0F to do Monadic Setup.
22. See if this is the token for FN. If not, branch to step 24.
23. JMP B/D3F4 to execute FN.
24. Compare the accumulator containing this character to $B4, the smallest possible token for the BASIC functions.
25. If the accumulator < $B4, this can't be a function, so branch to step 27.
26. JMP A/CFA7 to invoke a function since tokens >= $B4 represent functions.
27. The only remaining possibility for an operand with correct syntax is to have it appear in parentheses. Fall through to A/CEF1 to evaluate an expression within parentheses or produce a SYNTAX ERROR if not within parentheses.

Evaluate Expression Within Parentheses A/CEF1–A/CEF6

Called by:
BCC at A/CEEC Evaluate Operand or Handle Monadic Operator; JSR at A/CFD1 Invoke Function; JSR at B/D3FD FN.

Operation:

1. JSR A/CEFA to syntax check for a (at start and then retrieve the next character by calling CHRGET.
2. JSR A/CD9E to the Main Expression Evaluation to evaluate the expression within parentheses, leaving a numeric result in FAC1, or if it's a string result, leaving a pointer to the string descriptor in 64.
3. Fall through to A/CEF7 to syntax check for a) and then use CHRGET to retrieve the next character.

Monadic Operator Setup A/CF0D-A/CF13

Called by:

BEQ at A/CEB3 Evaluate Operand or Handle Monadic Operator (negate); Alternate A/CF0F: BNE at A/CED2 Evaluate Operand or Handle Monadic Operator (NOT).

Operation:

1. For negate (Monadic −) set the Y-register to $15 to index to the table entry in the Math Operation Dispatch Vector Table.
2. A/CF0F: if entered for NOT, then Y-register = $18 to index to the NOT entry in the Math Operation Dispatch Vector Table.

   Pull two bytes off the stack, removing the current return address.
3. JMP A/CDFA, step 40 of the Main Expression Evaluation, to push this operator's address and priority on the stack, also pushing FAC1, the left operand. Then recursively call the scan expression routine to scan for the next operand.

Locate Variable for Expression Evaluation A/CF28-A/CF83 and A/CF92-A/CFA6

Called by:

JMP at A/CE97 Evaluate Operand or Handle Monadic Operator.

If a variable does not already exist when it is used in an expression, the value of $00 is returned for this variable and a new variable descriptor is not created.

Operation:

1. JSR B/D08B to either locate or create a variable whose first character is currently pointed to by (7A). If the vari-

able name does not already exist or if the name is ST or TI, return with the value of the accumulator and the Y-register pointing to data value of $00 for this variable. Otherwise, the accumulator and the Y-register point to valid data in the variable descriptor for this already existing variable.

2. Set (64) from the accumulator and the Y-register, thus pointing (64) to the data for this variable.
3. X-register = 45 and Y-register = 46, loading the name of this variable and its type flags.
4. If the variable type, 0D, indicates this is a numeric variable, branch to step 15. If a string variable, continue with step 5.
5. Set 70 = $00.
6. JSR A/CF14 to see if (64), the address of this string descriptor, is in the range of A/C000–E342/E387. If the address is not in this range, just BCC to A/CF5C to RTS with (64) pointing to the descriptor for this string.
7. If the pointer is in this range, see if the variable name is TI$; if not, BNE to A/CF5C to RTS.
8. For a variable named TI$, JSR A/CF84 to do the Kernal routine to read the jiffy clock and store into FAC1.
9. Set 5E to $00.
10. Set 71 to $FF.
11. Set 5D to $06, initializing the number of digits before the binary point.
12. Set the Y-register to $24 to point to the table values for doing TI$ conversion.
13. JSR B/DE68 to convert FAC1 to a six-character ASCII string starting at 00FF.
14. JMP B/D46F to set a pointer to the string at 00FF and compute the length of the string. Since this string originates in page 0, allocate space for the string in the active string area and copy the string to the allocated space. Then leave the descriptor to the string on the temporary string descriptor stack and pointer string descriptor in (64). Exit this routine.
15. For numeric variables, if 0E has the high-order bit off, this is a floating point variable. If it's a floating point variable, branch to step 18.
16. For an integer variable, LDX from (64),0, retrieving the high-order value, and LDY from (64),1, retrieving the

low-order value. TXA so that the accumulator now has the high-order value.

17. JMP B/D391 to convert the two-byte integer in the accumulator and the Y-register to a normalized floating point number in FAC1 and then exit this routine.
18. For a floating point variable, see if (64) points to data for this floating point variable in the range A/C000–E342/E387 by doing a JSR A/CF14, returning with carry clear if not in this range. BCC to step 26.
19. If the first character of the variable name is T and the second is not I, branch to step 26. If the first character of the variable is not T, branch to step 24.
20. If the variable is TI, then JSR A/CF84 to copy the jiffy clock into FAC1.
21. Set the accumulator = $00 and the X-register = $A0 to indicate the exponent of +32 (decimal) for the TI value in FAC1.
22. JMP B/DC4F to set 61 to exponent of +32 ($80 + $A0), set the sign to positive, and then normalize FAC1. Exit from this routine.
23. Test for variable name of ST. If not found, branch to step 26.
24. If ST found, JSR FFB7 to execute the Kernal READST routine, returning the I/O status word, 90, in the accumulator, and then exit from this routine.
25. For all other floating point variables, other than TI or ST, LDA 64 and LDY 65, loading the contents of the pointer to the variable data in the descriptor.
26. JMP B/DBA2 to copy the floating point number pointed to by the accumulator and the Y-register to FAC1, then exit this routine.

Determine Value for TI or TI$ A/CF84–A/CF91

Called by:

JSR at A/CF7B Locate Variable for Expression Evaluation; JSR at A/CF48 Locate Variable for Expression Evaluation.

Operation:

1. JSR FFDE to execute the Kernal routine to read the jiffy clock, returning the high-order byte in the accumulator, the middle-order byte in the X-register, and the low-order byte in the Y-register.

2. STX 64, STY 63, STA 65, and store $00 into 62. Thus FAC1 now contains the value of the jiffy clock with 65 containing the most significant byte of the clock, 64 the next most significant, and 63 the least significant.
3. RTS.

See If Location of Variable Descriptor Is in Range of A/C000-E342/E387 and A/CF14-A/CF27

Called by:
JSR at A/CF6E Locate Variable for Expression Evaluation–Floating Point; JSR at A/CF3B Locate Variable for Expression Evaluation–String.

Operation:
1. (64) contains a pointer (the address) of the data in the variable descriptor of this variable.
2. If the address in (64) is in the range A/C000–E342/E387, return with the carry clear. B/DF13 contains the dummy variable of 0 that is used for TI or TI$.

The purpose for this routine is unclear. It may be used to make sure that the current variable descriptor is located in this range before allowing assignment to TI, TI$, or ST.

Perhaps this routine is used to determine whether the location pointed to by 64 is located in BASIC and perhaps prevent moving BASIC to RAM. This explanation seems unlikely since the top of the range does not extend all the way to the top of BASIC, it stops before getting to the warm and cold start routines.

Variables and Arrays

BASIC allows you to create or to reference variables either in direct mode or in program mode.

Let's use some examples to point out how variables are created and how they can be deleted both in program and direct mode. Follow these steps in exact order.

Enter the direct mode statement:

```
A = 2
```

Enter the direct mode statement:

```
PRINT A + 3
```

The computer responds with:

```
5
```

Conclusion 1: A variable descriptor is created and retained when entered in direct mode.

Enter the BASIC program:

```
10 PRINT A
20 C = 5
30 D = C * 5
```

Enter the direct mode statement:

```
PRINT A
```

The computer responds with:

```
0
```

Enter the direct mode statement:

```
PRINT D
```

The computer responds with:

```
0
```

Conclusion 2: Entering a program (actually entering or changing any line of a program) deletes all variable descriptors.

Enter the direct mode statement:

```
A = 2
```

Enter the direct mode statement:

```
LIST
```

The computer responds with:

```
10 PRINT A
20 C = 5
30 D = C * 5
```

Enter the direct mode statement:

```
PRINT A
```

The computer responds with:

```
2
```

Enter the direct mode statement:

```
PRINT D
```

The computer responds with:

```
0
```

Conclusion 3: Listing a program or entering any direct mode statements other than RUN or CLR or NEW does not delete previously created variable descriptors.

Enter the direct mode statement:

```
RUN
```

The computer responds with (this is the PRINT A response):

```
0
```

Enter the direct mode statement:

```
PRINT D
```

The computer responds with:

```
25
```

Conclusion 4: RUN deletes all previously created variable descriptors, and then creates new variable descriptors for the variable contained in the program.

Enter the direct mode statement:

```
CLR
```

Enter the direct mode statement:

```
PRINT D
```

The computer responds with:

```
0
```

Conclusion 5: CLR deletes all previously created variable descriptors. (NEW will also delete the variable descriptors.)

Enter the direct mode statement:

```
NEW
```

Enter the BASIC program:

```
20 C = 5
30 D = C * 5
40 STOP
50 PRINT X
```

Enter the direct mode statement:

```
RUN
```

The computer responds with:

```
BREAK IN 40
READY.
```

Enter the direct mode statement:

```
CONT
```

The computer responds with:

```
0
```

Conclusion 6: A variable returns an initial value of 0 the first time it is referenced in an expression.

Enter the direct mode statement:

```
RUN
```

The computer responds with:

```
BREAK IN 40
READY.
```

Enter the direct mode statement:

```
X = C * 2
```

Enter the direct mode statement:

```
CONT
```

The computer responds with:

```
10
```

Enter the direct mode statement:

```
RUN
```

The computer responds with:

```
BREAK IN 40
READY.
```

Enter the direct mode statement:

```
X = C * 2
```

Enter the direct mode statement:

```
RUN 50
```

The computer responds with:

```
0
```

Enter the direct mode statement:

```
RUN
```

The computer responds with:

```
BREAK IN 40
READY.
```

Enter the direct mode statement:

```
X = C * 2
```

Enter the direct mode statement:

```
GOTO 50
```

The computer responds with:

```
10
```

Conclusion 7: A variable can be created in direct mode and be referenced during execution of a program, and the direct mode variable is not deleted. Thus it is the RUN that deletes all variables, not statement execution of a program. Also notice how you can GOTO a line of the program.

The above examples illustrate some aspects of variable creation, deletion, and reference. The discussions in this section on variables provide the detailed explanation of the above examples plus some other information about variables.

To understand why RUN deletes all variables, you also need to be aware of the various pointers maintained by BASIC that point to areas of storage:

- (2B)—Pointer to the start of the stored tokenized program.
- (2D)—Pointer to the start of variable descriptors for strings, floating point, and integer variables.

- (2F)—Pointer to the start of arrays.
- (31)—Pointer to the start of the free area from which storage can be allocated for new active strings, arrays, or variable descriptors.
- (33)—Pointer to the bottom of active strings (active strings refer to strings created by concatenation or strings explicitly referenced in an expression that are not found as part of the stored text of a BASIC program line).
- (37)—Pointer to the top of the contiguous BASIC memory area.

CLR does the following to pointers: (37) is stored in (33), deleting all active strings; (2D) is stored in (2F) and into (31), thus deleting all variables and arrays.

The strings, variables, and arrays are not actually deleted since they have not been cleared from memory. Only the pointers have been changed and you can recover from an inadvertent CLR by resetting the pointers.

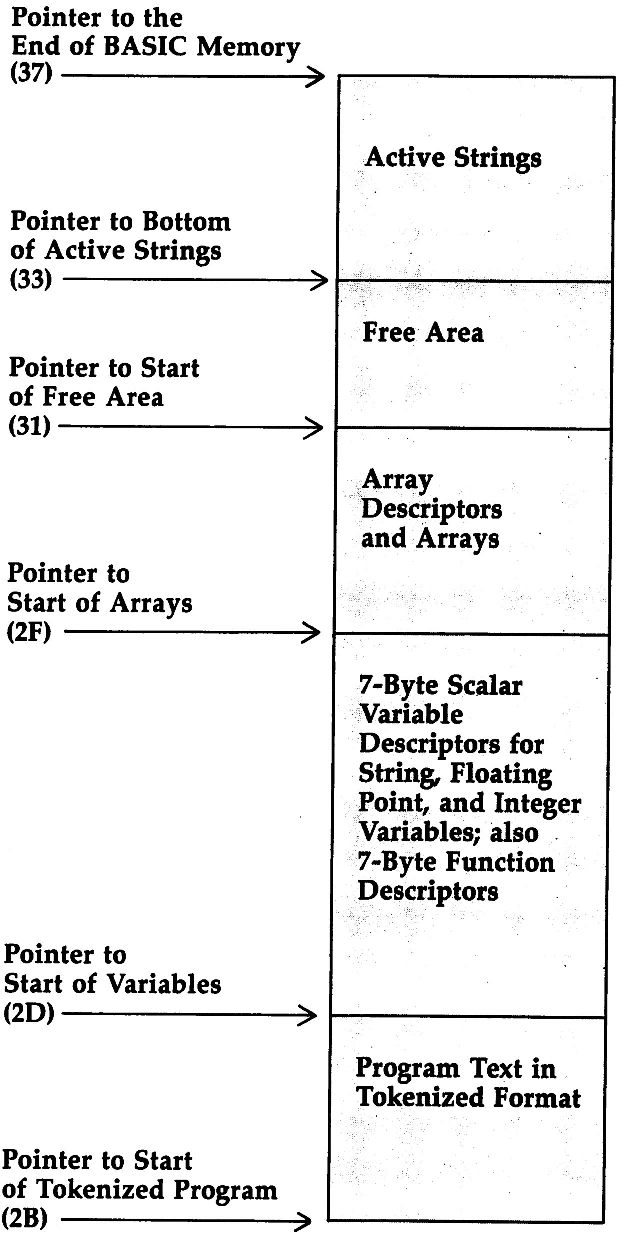
If you have an expansion board with a reset key and a machine language monitor cartridge (such as HESMON), it is possible to reset the pointers (short of doing a RUN to recreate the variables, arrays, and strings) by hitting the reset switch to switch you to the HESMON environment. Then look at the BASIC storage area to locate the start of arrays, the start of the free area, and the start of the active string area.

After locating the start of these three areas, store these hex locations into (2F) for the start of arrays, (31) for the start of the free area, and (33) for the bottom of active strings. Store these in the low byte–high byte format. After resetting these pointers, enter the HESMON command G E37B (Commodore 64) or G E467 (VIC) to directly execute the BASIC warm start which does not reset the pointers.

Of course, you could again run your program, and the same variables should again be created with the same values if all other conditions have remained unchanged. This is true because as part of the RUN routine, CLR is executed before the stored program text statements are actually executed.

The pointers can be illustrated by a diagram (Figure 1) of memory (using the default Commodore 64 memory configuration in this example). The relative size of each section can change; this diagram is not meant to be a scale drawing of the amount of memory allocated to each area.

Figure 1. BASIC Pointers



Each scalar variable (scalar means having only one value) is represented by a seven-byte descriptor. The first two bytes contain the name of the variable and high-order bit settings to signify the type. The final five bytes differ for string, integer, and floating point variables.

The format of the scalar variable descriptors can be pictured as:

Integer Variable Descriptor:

Name		**Value**		**Unused**		
High Bit On	High Bit On	High Byte	Low Byte	0	0	0

As an example, consider the integer variable AB% = 6.

$C1	**$C2**	**$00**	**$06**	**$00**	**$00**	**$00**

String Variable Descriptor:

Name			**Pointer**		**Unused**	
High Bit Off	High Bit On	**Length**	Low Byte	High Byte	0	0

As an example, consider the string variable AB$ = "hi" and assume it is located at $BFC2.

$41	**$C2**	**$02**	**$C2**	**$BF**	**$00**	**$00**

Floating Point Variable Descriptor:

Name		Exponent	Normalized Mantissa			
High Bit Off	High Bit Off	+$80	High Bit =Sign			

As an example, consider the floating point variable AB = PI, which is 3.14159265.

$41	**$42**	**$82**	**$49**	**$0F**	**$DA**	**$A1**

A few things should be noted about the floating point variable representation. The representation differs from that in the floating point accumulators because the floating point accumulators have a separate byte that represents the sign. The mantissa of the floating point variable is stored in normalized fashion as if the leftmost bit of the mantissa is one. I say "as if" because before this normalized mantissa is stored, the leftmost bit of the mantissa is converted to an indication of the sign of the variable, with 1 being a negative number and 0 a positive number.

Since normalizing a number always leaves the leftmost bit equal to 1, we have not lost any information. When the floating point variable is later loaded to a floating point accumulator, the leftmost bit is first checked to determine how to set the sign byte of the floating point accumulator and then the leftmost bit is always set to 1. Since floating point variables are always loaded first to a floating point accumulator before being used, the correction of the leftmost bit back to 1 is taken care of.

The exponent for the floating point number is represented in excess-128 notation, where 128 (decimal) or $80 (hex) is always added to the actual exponent. This excess-128 notation prevents having to use negative representations of exponents since negative exponents are just stored as values less than $80.

Array Descriptor

Name (same as for scalars)		Total Size Including Description		Number of Dimensions	Number of Elements in Last Dimension			Number of Elements in First Dimension	
		Low Byte	High Byte		Low Byte	High Byte		Low Byte	High Byte

As an example, consider the array defined for a DIM AB(5,3).

$41	**$42**	**$00**	**$81**	**$02**	**$00**	**$04**		**$00**	**$06**

The array is stored directly following the array descriptor. For integer arrays, each array element takes two bytes; for string arrays, each array element takes three bytes; and for floating point variables, each array element takes five bytes. The elements of the array are stored in *column major* form. An illustration of this storage format is given in the detailed description of the creation of arrays.

A few oddities about DIM exist. DIM A(5.7,3,2) does not give a syntax error and in fact creates an array as if (5,3,2) has been specified. DIM A,B,C,D,E,F does not create array descriptors or arrays, but rather creates the seven-byte variable descriptors for these variables and initializes them to a value of zero. This quirk could be useful if you wanted to declare all of your variables in one statement at the start of your program. One reason for doing this is to prevent having to move arrays up in memory if a new variable is declared after an array has been created.

Another quirk of arrays is found during expression evaluation. Consider the statement A = 5 * X + T(1,3,2). If variable X does not already exist, the value $00 is returned and no variable descriptor is created. However, if the array T does not already exist, the array descriptor for T and the area of storage needed for the array are allocated, and the pointer to the start of the free area is reset. The value returned is still $00. The reason for arrays but not variables being created is that during creation of scalar variables, a check is made to see whether the routine that called for creation of the scalar variable is the expression evaluation routine, and if so, it does not create the variable descriptor. The array creation routine has no such check to see if it is being called by the expression evaluation.

Locate or Create Variable B/D08B–B/D112

Called by:
JSR at B/D3C0 DEF Dependent Variable; JSR at A/CD24 NEXT; JSR at A/CF28 Search for Variable during Operand Evaluation; JSR at A/CC15 READ/INPUT; JSR at A/C9A5 LET; Alternate B/D090 JSR at B/D082 DIM; Alternate B/D092 JSR at B/D3EA FN.

Exit Conditions:
0D = $FF if string or $00 if numeric.
If 0D = $00 then 0E = $80 if integer or $00 if floating point.
45, 46 = variable name with appropriate flag settings to indicate type of variable.
(47) and accumulator, Y-register = pointer to the data for this variable.
SYNTAX ERROR if TI or ST.
TI$ or T shift-I return value of $00.

Operation:

1. X-register = $00 to prepare for the setting 0C.
2. JSR to CHRGOT to retrieve the last character scanned by CHRGET.
3. B/D090: JSR to here if you are doing a DIM with the X-register = $86. 0C = X-register. 0C is later used to determine if you are trying to DIM an array that already exists. If so, display REDIM'D ARRAY error message.
4. B/D092: 45 = accumulator, getting the first character of this variable name.
5. JSR to CHRGOT to load the accumulator with the first character of the variable name.
6. JSR B/D113 to see if this first character is in the range A–Z inclusive. If in this range, return with carry set.
7. If carry is set, branch to step 9.

8. For carry clear condition, JMP A/CF08 to display SYNTAX ERROR as this variable does not start with A–Z as required.
9. Set 0D and 0E to $00 to indicate a numeric variable and a floating point variable respectively until told differently.
10. JSR to CHRGET to retrieve next character (skipping spaces), returning with character in the accumulator and carry clear if it's a numeric character. Since spaces are skipped, variable names with space(s) between the first and second character of the variable name are valid.
11. If carry is clear branch to step 14.
12. Since the character is nonnumeric, JSR B/D113 to see if it is in the range A–Z.
13. If carry is now clear, it's not in the range A–Z, so branch to step 15 without saving the character in the X-register which temporarily holds the second character of the variable name.
14. Transfer this A–Z or 0–9 value to the X-register.
15. We now have a variable name with the first character in the range A–Z, stored in 45, and the second character if it exists is stored in the X-register and has a value of A–Z or 0–9.

    Now skip over characters until a nonalphanumeric character is found. This is done by calling CHRGET to get the next character. If CHRGET finds a numeric character, just call CHRGET again. If a nonnumeric character is found, JSR B/D113 to see if it's in the range A–Z, and if it is, just call CHRGET again. When finally a nonalphanumeric character is found, this step ends with the character in the accumulator.

    This skipping over characters is what allows variable names to be longer than two characters, but it also prevents the extra characters from forming part of the actual variable name value.
16. If the character following the variable name is $, this is a string variable. Branch to step 21 if true.
17. If the character is %, this is an integer variable. If not %, then branch to step 22.
18. Now that we have determined this is an integer variable, check 10, the flag for disallowing integer variables. If 10 is nonzero, integer variables are not allowed, so branch to step 8 to display SYNTAX ERROR.

Two situations disallow integer variables: The loop variable in a FOR statement and the function variable FN cannot be an integer variable. For example, FOR A% = 3 TO X, FNA%(3*2) and DEF FNA%(X) = 3*X all would produce syntax errors.

19. If 10 is zero, set 0E to $80 to indicate this is an integer variable.
20. Since this variable is an integer, turn on the high-order bit of 45, the first character of the variable name.
21. Branch here from step 16 if it's a string, or fall through from step 20 if it's an integer. For a string or integer variable, take the second character of the variable name, which is stored in the X-register, turn on its high-order bit, and leave the result back in the X-register.
22. JSR CHRGET to retrieve the next character following the variable name which possibly contains $ or % following the name.
23. 46 = X-register, thus storing the second character of the variable name.
24. ORA the accumulator with 10, which is set to a nonzero value if array variables are not allowed. The same conditions which disallow integer variables also disallow array variables, namely FOR and FN.
25. Subtract $28, the ASCII code for the left parenthesis.
26. If the result is zero, this character is a (. JMP B/D1D1 to find an array item or to create an array.
27. If the result is not zero, either this character is not a (or array variables have been disallowed. Set 10 to $00 if (is not found. If array variables have been disallowed, a syntax error results. SYNTAX ERROR is displayed even though no specific branch to display the syntax error is done here. This happens because FOR calls LET to evaluate the variable. LET expects to find an equal sign (=) following the variable and when it finds the (instead, the syntax error is displayed.
28. It's time to search for the integer, string, or floating point variable now. Begin the search from the start of variables (2D) by setting the X-register and accumulator to (2D).
29. B/D0EF: 60 = X-register.
30. B/D0F1: 5F = accumulator. (5F) points to the next variable descriptor to be checked.
31. Compare the accumulator and X-register to (2F), the

pointer to the end+1 of BASIC variables. If equal, the variable does not already exist in the variable area since the entire variable area has been scanned without finding this variable name. Branch to B/D11D to create a new variable.

32. See if 45 and 46, the name of this variable, match the name of the variable pointed to by (5F). This name of the variable pointed to by (5F) is in (5F),0 and (5F),1. If this comparison matches, branch to B/D185 to set (47) to the address of the actual data in the variable descriptor of this variable. This actual start of the data begins with the byte in the descriptor following the two-byte name area.
33. If the comparison in step 32 did not produce a match, add 7 to (5F) without changing the value in (5F). If carry is clear after this addition, branch to step 30 to store the accumulator as the new value in 5F. If carry is set, then INX, and branch to step 29 to store the X-register as the new value in 60. Each descriptor in the variable area occupies seven bytes.

Create New Variable Descriptor B/D11D-B/D184

Called by: BEQ at B/D0F9 Locate or Create Variable.

Operation:

1. PLA then PHA, thus obtaining the low-order byte of the routine that called Locate or Create Variable.
2. Compare to $2A. The return address of A/CF2B is set when Locate or Create Variable is called by doing a JSR B/D08B at A/CF28. Because only the low byte of the return address is being compared, this comparison works on both the Commodore 64 and the VIC.
3. If comparison is not equal, branch to step 5.
4. If comparison is equal, point the accumulator to $13 and the Y-register to $BF (Commodore 64) or $DF (VIC). These are the low and high bytes respectively of location B/DF13 which contains $00. RTS to the routine in A/CF28, Locate Variable for Expression Evaluation.

Thus you cannot create a new variable descriptor during expression evaluation. If you reference a variable that does not exist during an expression, the value of $00 is returned.

5. Set the accumulator to 45 and the Y-register to 46 to reload the name of this variable along with its associated string, floating point, or integer flags.
6. If this name is TI or ST, display SYNTAX ERROR and return to the Main BASIC Loop.
7. If this name is TI$ or T shift-I, branch to step 4 to RTS with the pointer to $00 byte. TI$ if used results in a syntax error, but T shift-I is a valid variable name.
8. Store (2F), the pointer to start of BASIC arrays, into (5F).
9. Store (31), the pointer to the end+1 of BASIC arrays (the start of the free area), into (5A).
10. Add 7 to this pointer to the start of BASIC free area. Leave the results in the accumulator and Y-register and also store them in (58).
11. JSR A/C3B8 to check that seven bytes are available in the free area. Then move the area from (5A) back through (5F), which is the entire area for arrays, to the destination starting at (58) and working downward in memory. Thus the BASIC arrays are moved seven bytes upward in memory to allow for the new seven-byte variable descriptor. At exit from the move, (58) points to the start of the array area minus one. This complete move of arrays when a new variable is created can be avoided by declaring all of your variables before you declare your arrays.
12. Increment (58) so that it points to the start of arrays, and store it in (2F), the pointer to the start of arrays.
13. (5F) now points to the start of the area reserved for the seven-byte descriptor. Store the name of the variable into the first two bytes. Then store $00 in the remaining five bytes to initialize the value of the variable to zero. If it's a string variable, the length of the string is 0 and the pointer points to $0000.
14. Fall through to B/D185 to RTS with (47) pointing to the byte following the two-byte name of this variable, thus actually pointing to the data for this variable.

Return Address of Data Area in Descriptor of Variable B/D185–B/D193

Called by:
Fall through from Create Variable; BEQ at B/D106 Locate or Create Variable (if located).

Operation:

1. Add 2 to (5F), which contains the address of this variable descriptor.
2. Store the result in (47). Thus (47) now has the address of the data area in this descriptor that is located just past the two-byte variable name. Also exit with the accumulator containing the low-order byte of the address and the Y-register containing the high-order.
3. RTS.

Locate or Create Array
B/D1D1–B/D244

Called by: JMP at B/D0E4 Locate or Create Variable.

Operation:

1. Do logical OR of 0C and 0E and push the result onto the stack. 0C is $86 if a DIM is being done. 0E is $80 if it's an integer variable or $00 if it's a floating point variable.
2. Push 0D, which represents a string variable if $FF or a numeric variable if $00, onto the stack.
3. Y-register = $00 to set the initial count of the number of dimensions of the array to 0.
4. Transfer the Y-register to the accumulator and push onto the stack.
5. Push 45 and 46, the variable name, onto the stack.
6. JSR B/D182 to convert the expression within parentheses that represents the element of the array to an integer in FAC1, with the two-byte integer left in 64 and 65. The maximum value is 32,768 (decimal); if greater than this, display ILLEGAL QUANTITY. If the array element is specified as a negative number, also display ILLEGAL QUANTITY. If the array element is specified as a string or a string variable, display TYPE MISMATCH. After any of these errors, return to the Main BASIC Loop.
7. Restore 45 and 46, the name of this array, from the stack.
8. Pull the count of the number of dimensions from the stack and restore to the Y-register.
9. Set the X-register to the stack pointer value to allow retrieval from the stack without using PLA.
10. Retrieve the result of 0C ORA 0E from the stack by doing LDA 0102,X, and push this value back onto the stack.

Retrieve the value of 0D from the stack by doing LDA 0101,X and push this value back onto the stack.

11. Accumulator = 64, the high-order byte of the element value for this dimension. Store the accumulator at 0102,X, thus now moving this value to the value that was previously occupied in the stack by the ORA of 0C and 0E. Accumulator = 65, the low-order byte of the element value for this dimension. Store the accumulator at 0101,X, thus now moving this value to the value that was previously occupied in the stack by 0D.
12. INY, incrementing the count of the number of dimensions.
13. Call CHRGOT to retrieve the character already scanned that follows this element expression.
14. If the current character is a comma, another dimension of the array follows, so branch to step 4 to determine the element index for this dimension and to push this index onto the stack.

Steps 4–14 are executed for each dimension of the array so that when all dimensions are done, the 0C ORA 0E and 0D are on the top of the stack and the array indexes are below these values in the stack.

For example, if the array reference is A(1,3,5), after all the dimensions have been scanned the stack can be pictured as follows, with each element of a dimension taking up two bytes on the stack. When the routine that returns the address of the data of an array element uses the element values to determine the offset into the array, it pulls off the dimension elements working from the last dimension to the first.

	Location	**Value**	
Stack Pointer→	X		
	X+1	00	0D (Numeric Variable)
	X+2	00	0C ORA 0E
	X+3	05	
	X+4	00	
	X+5	03	
	X+6	00	
	X+7	01	
	X+8	00	

15. If the current character is not a comma, we're done with array dimension calculations. Save the Y-register, the number of dimensions in this array, in 0B.

16. JSR A/CEF7 to check for a) following the last element of the array; if not found, display SYNTAX ERROR and go to the Main BASIC Loop.
17. Pull the string/floating point variable type from the stack and save it in 0D.
18. Pull the ORA of 0C and 0E from the stack and save it in 0E.
19. AND this value with $7F to turn off the indicator that this is possibly an integer array and store the result into 0C. 0C is thus not affected by any value in 0E while 0E is $86 if this array is being referenced by DIM.
20. X-register = 2F and accumulator = 30. (2F) points to the start of arrays.
21. Store the X-register in 5F and accumulator in 60, setting the value for the pointer that moves through the array area to compare for a matching array name.
22. Compare (5F) to (31), the pointer to the end+1 of the arrays.
23. If equal, branch to B/D261 to create an array since an array by this name does not exist.
24. Otherwise, see if the name in the current array descriptor pointed to by (5F) matches the name of the array being searched for that is stored in 45 and 46. If the array names match, branch to B/D24D to check for redimensioning of an array. If the array has not been redimensioned, return a pointer to the data of the array element specified in (47). Then exit from this routine.
25. If the names didn't match, add the length of this array, which is found in the third and fourth bytes of the array descriptor, to (5F), leaving the results in the X-register and accumulator.
26. Branch to step 21 to check the next array descriptor.

Array Already Exists—Check for REDIM'D ARRAY, Locate Element in Array B/D24D—B/D260

Called by: BEQ at B/D235 Locate or Create Array.

Operation:

1. If 0C is not zero, a DIM statement is being executed. Since an array by this same name already exists, display REDIM'D ARRAY and return to the Main BASIC Loop since

once an array has been created, it cannot be created again by a DIM.

2. JSR B/D194 to calculate the length of this array descriptor by multiplying the number of dimensions by two since each dimension needs two bytes for element indexes. Then add five to account for the standard two-byte name, two-byte length of array, and one-byte number of dimensions. The array descriptor starts at (5F) and ends at (58) minus one. The actual data for the array starts at (58).
3. Compare 0B, the computed number of dimensions for this array from this most recent reference, to the number of dimensions specified in the array descriptor. If not equal, display BAD SUBSCRIPT and return to the Main BASIC Loop.
4. JMP B/D2EA to locate a particular element of this array, returning the address of the data for this element in (47). Eventually a value will be stored into this data area if this variable reference is on the left side of the parentheses (a LET statement) or the data will be loaded from the data area if this variable reference is on the right side of the parentheses (part of an expression).

Determine Size of Array Descriptor

1. Accumulator = 0B, the number of dimensions of this array.
2. Multiply the accumulator by two by doing an ASL, obtaining the number of bytes in the descriptor needed for the two-byte dimension indexes.
3. Add five to account for the fixed five bytes of the array descriptor—the two-byte name, the two-byte length, and the one-byte number of dimensions.
4. Add (5F), which points to the end+1 of the arrays if this array was not found, or to the current array descriptor if this array was found.
5. Store the result in (58), which thus points to one past the array descriptor, the start of the actual values for this array. Also, the accumulator and Y-register point to this location.
6. RTS.

Return Address of Data of an Element of the Array B/D2EA–B/D34B

Called by:

Fall through from Create an Array; JMP at B/D25E Array Already Exists.

There are two things to keep in mind while reading the description of the calculation to return the address of the data area for an element of any array.

First, the array element indexes are pulled from the stack in reverse order to what they are specified in the reference. For example, in a reference of A(2,3,7), the order of pulling the indexes from the stack is 7, 3, 2.

Second, the descriptor for the array has a two-byte field for each dimension of the array that specifies the maximum number of elements in that dimension. These values are stored in the descriptor starting with the last dimension and ending with the first. For example, if an array is created by DIM A%(3,5,8), the format of the array descriptor is two-byte name, two-byte total length, one-byte number of dimensions, two-byte maximum number of elements in the third dimension of 9, two-byte maximum number of elements in the second dimension of 6, and two-byte maximum number of elements in the first dimension of 4.

Operation:

1. (5F) points to the descriptor for this array, and the Y-register is 4 at entry. Load the accumulator with the number of dimensions in this array from the fifth byte of the descriptor.
2. Save the number of dimensions in 0B.
3. Set the accumulator to $00 and store in 71.
4. Store the accumulator into 72, storing the high-byte relative offset.
5. Retrieve the two-byte value from the stack that gives the element number for this dimension and save it in 64, 65. Remember that the elements are being pulled off the stack from the last dimension down to the first. Also, the accumulator has the 65 value and the X-register has the 64 value. If a dimension element has the two-byte value 00 03, then 64 and the X-register have the $03 value and the accumulator and 65 have the 00 value.

6. Compare to the next two-byte field in the array descriptor that contains the maximum element index plus one for this dimension. For example, if a dimension has a maximum number of 11 elements, the possible index values range from 0 through 10. If the index value pulled from the stack is less than the maximum number of elements, this is a valid dimension reference. If the index value is equal to or greater than the maximum number of elements, display BAD SUBSCRIPT and return to the Main BASIC Loop.
7. This element is valid so continue the computation to determine its relative offset.

   Load the accumulator from 72 and then ORA 71. If the result is zero, this is the first dimension computed, so branch to step 12 to skip the following multiplication. Could also branch if all indexes specified so far have been $00.
8. JSR B/D34C to multiply the index for this element by the total number relative offset so far, returning the result in the X-register and the Y-register.
9. Add 64, the low-byte of the array element index, to the value returned in the X-register and put the sum back into the X-register.
10. Store the Y-register in the accumulator.
11. Load the Y-register from 22, which was holding the pointer to the low-order byte of this dimension subscript.
12. Add 65, the high-order byte of the array element index, to the accumulator which contains the results of the multiplication in step 8.
13. STX 71, saving the new low-order relative offset to this element.
14. Decrement 0B, the number of dimensions.
15. If 0B is not yet zero, branch to step 4 to add the length of the next dimension to the offset calculated so far.
16. If all the dimensions are done, store accumulator in 72, saving the high byte of the relative offset to this array element.
17. Compute the value in the X-register to reflect the number of bytes per element for this kind of array. If it is floating point, X=5; if string, X=3; or if integer, X=2.
18. Save the X-register in 28, saving the number of bytes per array element.
19. Set the accumulator to $00 to prepare for next step.

20. JSR B/D355 to multiply (71), the number of elements in the array to this element requested, by (28), the number of bytes per element. Leave the result in the X-register and the Y-register.
21. Add the X-register and the Y-register, which now contain the offset to the element requested in the array, to (58), which points to the start of this array.
22. Store the sum in (47), which now points to the element requested in the array, and also exit with the accumulator and the Y-register containing this pointer value.

Examples:
If X%(1,2) element has been requested and the X array has been created with a DIM X%(2,2) statement, the descriptor will have maximum values of 3 and 3 for the dimensions of this array.

A 3 X 3 array contains nine elements that can be represented as follows with the boxes labeled with their index values. Also, if this is an integer array, each element occupies two bytes, and the boxes also show the relative offsets in decimal using zero as the start of the first element. The following order is sometimes called the *column major-order* since columns are stored before rows.

0 (0,0)	6 (0,1)	12 (0,2)
2 (1,0)	8 (1,1)	14 (1,2)
4 (2,0)	10 (2,1)	16 (2,2)

For X%(1,2) the first time step 7 above is reached, (71) contains $00, so the multiplication is not done. Instead, the accumulator value of $00 is added to 65 and stored in 72 while the X-register value of $02 is stored in 71, thus adding the number of elements specified for the second dimension and giving the result of 2. The next time step 7 is reached, (71) contains $0002 so the multiplication routine is called. This routine multiplies 2 (the previous result set in (71)) times 3 (the maximum number of elements in the first dimension) to return a result of 6. Now the first element index of 1 is added to this 6 to give 7. Now that all dimensions have been processed, multiply the size of each element in the array—which is 2 for integer arrays—by 7 to give 14 as the offset to the element requested.

With another array created by DIM X%(2,2) consider the reference X%(2,0). For the last dimension, the multiplication routine is not called because (71) is zero. Then 0 is added to (71), leaving (71) = 0. For the first dimension, the value in (71) is still zero, so just add 2 to the 0 in (71) to give the result of 2. Since both dimensions have now been processed, multiply the number of bytes per element—2 for integers—by 2 to give an offset of 4 for this element.

As a final example, consider a reference to an array that has been declared with a DIM X%(1,3). Again, picture the array as shown below.

0 (0,0)	4 (0,1)	8 (0,2)	12 (0,3)
2 (1,0)	6 (1,1)	10 (1,2)	14 (1,3)

For the reference X%(1,2) the calculation of the offset is: Add the second dimension reference, 2, to (71), which contains $00, leaving the result in (71) of 2. Multiply 2 by the maximum number of elements in the first dimension, 2, to give 4. Now add the 1 for the first dimension element, giving 5. Multiply by 2 bytes per element to give an offset of 10.

As you can see from these examples, this routine does return with the correct offset from the start of the array. The address of the start of the array is then added to this offset to obtain the address of this element of the array, which is returned as a pointer in (47).

Compute Offset to Element in Array or Compute Size of Array to Be Allocated B/D34C–B/D37C

Called by: JSR at B/D29D; JSR at B/D316.

Operation:

1. Save the Y-register in 22, holding the current pointer to the array descriptor, pointer to the second byte of a two-byte dimension field.
2. Load the accumulator from the low-order element index pointed to by (5F),Y and save in 28.
3. DEY and LDA (5F),Y to put the high-order element index into the accumulator.
4. B/D355: JSR here from B/D33B when retrieving an element of the array and all dimensions have already been computed to give the relative offset to this element. If called from B/D33B, the accumulator = $00 and 28 holds the byte size of this type of array element.

   Store the accumulator into 29. Thus (28) now holds either the byte size of elements in the array or the maximum number of elements in this dimension of the array.
5. Store $10 into 5D, preparing a counter of the 16 bits that must be multiplied.
6. Store zero into the X-register and Y-register which will hold the final result.
7. Do an ASL of the X-register and an ROL of the Y-register, thus computing YX = YX * 2.
8. If the carry is set as a result of the rotation of Y-register, branch to B/D30B to display OUT OF MEMORY and return to the Main BASIC Loop.
9. With the carry now clear, do an ASL 71 and an ROL 72, thus shifting a 0 into 72, 71 and shifting the high-order bit out of 72, 71 into the carry. At exit from this routine, 72 and 71 will both be $00 since 16 bits are shifted.
10. If carry is clear as a result of step 9, branch to step 13.
11. If carry is set as a result of ROL 72, add the number of elements in this dimension (or the number of bytes per element). X-register = X-register + 28; Y-register = Y-register + 29.
12. If carry is set as a result of the addition in step 11, branch to B/D30B to display OUT OF MEMORY.

13. Decrement 5D, which holds the number of bits remaining to be multiplied. If not zero, branch to step 7.
14. When 5D reaches zero, RTS with the X-register and the Y-register holding the final result of this multiplication.

Create an Array
B/D261–B/D2E9

Called by: BEQ at B/D226 Locate or Create Array.

Operation:

1. JSR B/D194 to determine the length of the array descriptor needed for this array and with (58) pointing to the end+1 of the area needed for the descriptor. Since new arrays are created in free memory above the value in (58), (58) represents the pointer to the start of free memory plus the length of the descriptor. (5F) points to the start of the descriptor.
2. JSR A/C408 to see if the address pointed to by the accumulator and the Y-register is less than the top of available free space. If so, enough space exists to add this array descriptor. If enough space does not exist, try garbage collection of strings one time, and then again see if enough memory exists for the descriptor. If there is still not enough, display OUT OF MEMORY and go to the Main BASIC Loop.
3. Store $00 into 72, the high-byte of the length of the array to be allocated.
4. X-register = $05 to set the default array element size to that of the floating point elements.
5. Store 45, the first byte of the array name, into the first byte of the array descriptor at (5F),0.
6. If this first byte has its high-order bit on, this is an array of integers. If true, then DEX so that the X-register is now $04.
7. Store 46, the second byte of the array name, into the second byte of the array descriptor at (5F),1.
8. If this second byte has its high-order bit on, this is either an array of strings or an array of integers. If true, then DEX twice so that the X-register = $02 if it's an array of integers, or the X-register = $03 if it's an array of strings. If it's a floating point array, the X-register is still $05.

9. 71, the number of bytes per element of this array, = X-register.
10. Store the number of dimensions of the array, 0B, into the fifth byte of the array descriptor.
11. X-register = $0B to set a default maximum number of elements per dimension size of 11 (decimal) for use in implicit array creation.
12. Accumulator = $00.
13. If bit 6 of 0C is clear, an implicit array creation is being made by referencing an array, so branch to step 15.
14. If bit 6 of 0C is set, a DIM is being performed to do an explicit array creation. If a DIM, pull two bytes from the stack. These bytes were placed there by DIM and are the maximum index for this dimension. Add one to get the maximum number of elements in this dimension. Leave the results in the accumulator and X-register.

    Since bytes of the dimensions are pulled from the stack starting with the last dimension and ending with the first, the maximum number of elements for each dimension is stored in the reverse-order. The last dimension maximum number of elements is the first value stored in the descriptor.
15. Store the accumulator and X-register in the next two bytes of the descriptor for this array.
16. JSR B/D34C to continue to compute the size of storage needed to hold all array elements. The first time the routine is called, (71) contains the number of bytes per element of the array—5 for floating point, 3 for string, or 2 for integer. This number of bytes per element is multiplied by the maximum number of elements in the first dimension to get the maximum number of bytes needed for this first dimension. Following dimensions will multiply their maximum number of elements by the value that has been accumulated so far in (71).

    For example, if a DIM(3,4) statement is executed to create an array and this is a string array, the number of bytes per element, 3, is multiplied by the maximum number of elements in the first dimension, 4, to give 12. Then 12 is multiplied by the maximum number of elements in the second dimension, 5, to give a total of 60 bytes of storage needed for this array. You could picture the array area as follows with the decimal number representing the address of that element relative to 0. With 20 total elements in the array (4 rows by 5 columns), you can see that 60 total bytes are needed for this string array.

0 0,0	12 0,1	24 0,2	36 0,3	48 0,4
3 1,0	15 1,1	27 1,2	39 1,3	51 1,4
6 2,0	18 2,1	30 2,2	42 2,3	54 2,4
9 3,0	21 3,1	33 3,2	45 3,3	57 3,4

17. STX 71 and STA 72 so that (71) holds the accumulated length of the array through this dimension.
18. Reload the pointer to the array descriptor that was saved in 22.
19. Decrement 0B, the number of dimensions of the array.
20. If 0B is not zero, branch to step 11 to process the next dimension.
21. (58) contained the pointer to the start of the data area for this array. Add the X-register and accumulator, which hold the number of bytes needed for this array to (58), leaving the result in the accumulator and Y-register which now point one byte past the end of the area needed for this array data. Also, if this addition caused a carry during addition of the high-order byte, then display OUT OF MEMORY and return to the Main BASIC Loop.
22. JSR A/C408 to see if there is enough memory left in the free area to add this array data. If not, try garbage collection once. If there's still not enough, display OUT OF MEMORY and go to the Main BASIC Loop.
23. Store the accumulator and Y-register in (31), setting a new pointer to the start of the free area to be one byte past the end of this newly allocated array.

24. Now store zeros into all bytes of the array data, working back from the end of the array to the start.
25. Subtract the pointer to the start of this array descriptor (5F), from the pointer to the start of the free area (31) to determine the length of this array including the array descriptor. Store this total array length into the third and fourth bytes of the array descriptor.
26. If 0C is nonzero, indicating this array is being created by a DIM, exit this routine.
27. If the array is being created because of a reference to the array, INY to point the Y-register to the location in the array descriptor of the number of dimensions for this array, and fall through to B/D2EA to return a pointer to the element of the array specified in (47). Thus, if you reference an array that has not already been created, not only do you create the array, but you can also retrieve a pointer to this array element and store a value into the array element if you're doing a LET statement.

DIM
B/D07E-B/D08A

Called by: Statement Execution.

DIM has a few oddities. DIM A(5.7,2,1) is valid and is the same as DIM A(5,2,1). DIM A,B,C creates three seven-byte variable descriptors for A, B, and C in the variable area. A subsequent DIM A,B,C does not produce a REDIM'D ARRAY error message. Consider the statement X = 3 + A + B(2,2) when neither A nor the array B existed before this expression evaluation. A descriptor for A is not created in variable memory, while an array descriptor and the associated data area for the array is created.

Operation:

1. B/D081: This is the main entry point for DIM. TAX to transfer the token for DIM, $86.
2. Store the accumulator in 0C to indicate that a DIM is being performed. If an array with this same name already exists, then REDIM'D ARRAY will be displayed later.
3. JSR B/D090 to locate or create a variable. If the variable is located, REDIM'D ARRAY error is displayed. If the variable does not already exist, create the variable.

4. Call CHRGOT to retrieve the last character scanned.
5. If this last character is the $00 end-of-line character, then RTS.
6. If this last character is not $00, branch to B/D07E.
7. B/D07E: JSR A/CEFD to syntax check for a comma, thus allowing statements of the form DIM A(3,2),B(5,3). If a comma is found, fall through to step 1 at B/D081.

LET
A/C9A5–A/CA1C and A/CA2C–A/CA7F

Called by:
JMP at A/C804 Execute the Current BASIC Statement; JSR at A/C746 FOR; Alternate A/C9C2 READ/INPUT/GET; Alternate A/C9DA READ/INPUT/GET.

LET can be called in two ways. The LET keyword starting a BASIC line is the explicit manner while starting a line with a nonkeyword value is the implicit way to execute LET. If the implicit way is done, the first item encountered in the line has to be a variable or a syntax error results.

Operation:
1. JSR B/D08B to either locate or create a variable. If the variable is found, then upon return, 45 and 46 hold the name of the variable with the appropriate flag settings for the variable type, 0D = $FF if it's a string variable or $00 if it's numeric; 0E = $80 if it's an integer variable or $00 if it's floating point (0E valid only if 0D = $00). The accumulator and Y-register as well as (47) contain a pointer to the actual data area for this variable in the variable descriptor or in the array element. If the variable name is TI$ or T shift-I, return with 47 pointing to a value of $00 in the data area. If the variable name is TI or ST, display SYNTAX ERROR.
2. Store the accumulator in 49 and the Y-register in 4A, setting (49) to be the pointer to the data area for this variable.
3. Load the accumulator with $B2, the token for =.
4. JSR A/CEFF to see if the first nonspace character following the variable name is = and display SYNTAX ERROR if not =. Call CHRGET to return with the pointer to the first nonspace character of the expression following =.
5. Push 0E and 0D onto the stack.

6. JSR A/CD9E to evaluate the expression to the right of the = sign. Convert numeric expressions to a normalized floating point number in FAC1. If it's a string expression, then (64) is a pointer to the string descriptor, which has a pointer to the string and the string's length.

    If the LET statement appears in direct mode, the string has been stored in the active string area and a descriptor to the string is in the temporary string descriptor stack. If the LET statement appears in program mode, the string is stored intact in the program text area in the LET statement and a string descriptor in the temporary string has been concatenated. The newly concatenated string is stored in the active string area.
7. Pull the value of 0D from the stack and rotate left into the carry without storing this value back into 0D. If the LET variable is a string variable, carry is set.
8. JSR A/CD90 to determine whether the expression evaluated is the same type as the variable to which it is to be assigned. If not, display TYPE MISMATCH and go to the Main BASIC Loop. At A/CD90 a BIT of 0D, which now contains the type of the expression on the right side of the equal sign, is done. If 0D is $FF and carry is set from step 7, this is an assignment of a numeric expression to a numeric value.
9. If 0D is nonzero, indicating a string variable, branch to step 17.
10. Pull the value of 0E from the stack.
11. A/C9C2: If the high-order bit is zero, indicating a floating point variable, then BPL to step 16.
12. For an integer variable, JSR B/DC1B to round FAC1.
13. JSR B/D1BF to convert FAC1 from a normalized floating point number to a signed integer in FAC1, with 64 and 65 holding the two-byte integer. If FAC1 holds a value greater than the largest permissible integer (32768), then display ILLEGAL QUANTITY ERROR and go to the Main BASIC Loop.
14. Store 64 at (49),0 and store 65 at (49),1. (49) points to the data area for this integer variable.
15. RTS after handling this integer variable.
16. JMP B/DBD0 to copy the number in FAC1 to the data area for this floating point number pointed to by (49), and then exit from this routine.

A number in a floating point accumulator differs slightly from the form in a floating point variable. When a floating point accumulator is stored to a floating point variable, the high-order bit of the first byte of the variable is set to zero if it's a positive number or to one if it's a negative number. When the floating point variable is loaded into the floating point accumulator, the high-order bit of the first byte is examined to see if it's a negative or a positive number and is then set to one.

17. Branch here for string variables. PLA to pull off the 0E value to restore the stack to its proper state, but there's no need to use this value for strings.
18. A/C9DA: Y-register = 4A, which is the high-order byte of the string data area as computed when the temporary string descriptor was created.
19. Compare the Y-register to $BF (Commodore 64) or to $DF (VIC), which is the high byte of $B/DF13, the dummy variable of 0 that is returned for TI$ or T shift-I.
20. If not equal, branch to step 40 to process all other strings.
21. If equal, handle TI$. JSR B/D6A6 to delete the temporary string descriptor for TI$ from the stack.
22. If the string specified for TI$ is not six characters long, display ILLEGAL QUANTITY ERROR and go to the Main BASIC Loop. For example, TI$="23" gives an ILLEGAL QUANTITY. However, TI$="ABCDEF" is perfectly acceptable although it will assign an unusual value to TI$. Try these examples:

```
10 TI$="ABCDEF"
20 PRINT "TI$="TI$
```

Run the program.

Now add these lines:

```
30 FOR A=1 TO 10
40 FOR I=1 TO 1000:NEXT
50 PRINT "TI$="TI$
60 NEXT
```

Run the program.

Finally, add this line and run the program again:

```
70 TI$="12CDEF":PRINT "TI$="TI$
```

23. Store $00 into 61, the exponent of FAC, and into 66, the sign of FAC1.
24. Store the Y-register (which is initially $00) into 71.
25. JSR A/CA1D to add the next digit (or nonnumeric value) in the string to FAC1.
26. JSR B/DAE2 to multiply FAC1 by 10 to prepare for the next digit.
27. Increment 71 and the Y-register.
28. JSR A/CA1D to add the next character to FAC1.
29. JSR B/DC0C to copy FAC1 to FAC2 with rounding, returning with accumulator = 61, the exponent of FAC1.
30. Transfer the exponent to the X-register.
31. If the exponent is zero, branch to step 34 to skip the next multiplication.
32. Increment the X-register, thus incrementing the exponent.
33. JSR B/DAED to add FAC2 to FAC1, and also increment the exponent, 61, of FAC1, thus multiplying the total by 2.
34. Load the Y-register from 71.
35. Increment the Y-register, and if not equal to six, branch to step 25 to do the next character of the TI$ string.
36. After all six characters of the string have been processed, JSR B/DAE2 to multiply FAC1 by 10.
37. JSR B/DC9B to convert the floating point number to a signed integer in FAC1.
38. Load the X-register from 64, the Y-register from 63, and the accumulator from 65.
39. JMP FFDB to set the jiffy clock A2, A1, A0, from the accumulator, X-register, and Y-register respectively, and then exit from this routine.
40. Continue with other strings. If (64),1 and (64),2—which contain the pointer to the string—are less than (33), the pointer to the bottom of active strings, branch to step 42. If the branch is taken, the string already exists in active string memory.
41. If (64), the current string descriptor for the string variable, is less than (2D), the pointer to the start of variables, then continue at step 42. If this branch is taken, the string already exists in stored format in the program text area. If 64 >= (2D), branch to step 44.
42. Load the accumulator from 64 and the Y-register from 65, thus loading the pointer to the string descriptor.

43. Jump to step 49 to store the pointer to this string descriptor in (50) and then copy the string descriptor from the temporary string descriptor stack to the string descriptor for this string variable and exit.
44. Y-register = $00 and accumulator = (64),0, thus loading the length of this string.
45. JSR B/D475 to allocate an area for this string in the active string area and set (50) to the start of the string area allocated.
46. Store (50) into (6F).
47. JSR B/D67A to copy the string to the area allocated for it in the active string area, leaving (61) pointing to the string descriptor for the new string.
48. Accumulator = $61 and Y-register = $00 to point to this new string descriptor.
49. Store the accumulator and Y-register into (50), which thus points to this string descriptor.
50. JSR B/D6DB to remove the entry for this string descriptor from the temporary string descriptor stack if it is found there.
51. Copy the three-byte string descriptor, pointed to by (50), to the data area for this string variable, pointed to by (49).

Add Character for TI$
A/CA1D–A/CA2B

Called by:
JSR at A/C9EF LET—TI$; JSR at A/C9F9 LET—TI$.

Operation:
1. Load the accumulator with the next character of the ASCII string pointed to by (22),Y.
2. JSR 0080 (part of CHRGET). Although I guess this JSR was intended to be a test for a numeric value in the TI$ string, it actually isn't. Instead the carry is only set for ASCII values <$30. Thus letters and other ASCII characters >= $30 are valid for the TI$ string.
3. If carry is set, display ILLEGAL QUANTITY.
4. Since carry is now clear if this step is reached, SBC $2F to actually subtract $30 from this ASCII character, thus supposedly converting it to a digit from $00–$09. However, because of the bug mentioned in step 2, values greater than $09 can result.
5. JMP B/DD7E to add the accumulator to FAC1, then RTS.

Type Check Variables or Expressions A/CD8A-A/CD9D

Called by:
JSR at A/C775; JSR at A/C79C FOR; JSR at B/D438 FN; JSR at B/D79E Evaluate Expression and Convert to Fixed Point Integer; JSR at B/D7EB Get Parameters for POKE and WAIT; JSR at E12A/E127 SYS; Alternate A/CD8A: JSR at A/C772 FOR; JSR at A/CDF6 Main Expression Evaluation; JSR at A/CE61 Prepare for Stacked Operator Execution; JMP at A/CFE3 Function Invocation; JSR at B/D1B8 Float to Fix Conversion Only If Numeric Variable Type; JSR at B/D3C3 DEF; JSR at B/D3F1 Check Syntax for FN Descriptor and Return Pointer to Its Data Area; JSR at B/D400 FN; JSR at B/D465 STR$; Alternate A/CD8F: JSR at A/CFBA Function Invocation; JSR at B/D646 String Concatenation; JSR at B/D6A3 Type Check for String Variable and If String Then Return String Length and Pointer to String; Alternate A/CD90: JSR at A/C9BC LET; JSR at B/D016 Comparison of Operands.

Operation:
1. A/CD8A: JSR A/CD9E to evaluate an expression and to set 0D to $FF if it's a string expression or to $00 if it's a numeric expression.
2. A/CD8D: CLC to indicate checking for a numeric variable or expression and then use a BIT dummy instruction to fall through to step 4.
3. A/CD8F: SEC to indicate checking for a string variable or expression.
4. A/CD90: BIT 0D, the flag which indicates the type of the most recent variable or expression evaluated.
5. BMI to step 8 if it's a string variable.
6. BCS to step 9. The carry is set if you are looking for a string type. Step 6 is entered if this variable or expression is actually numeric. Thus a type mismatch has occurred.
7. RTS if there is a successful match of the expected type value to the actual type value.
8. BCS to step 7 to RTS since the variable actually was a string and the carry is set indicating that is what was expected. Fall through to step 9 if the variable actually was a string, but the carry was clear indicating that a numeric type was expected.
9. Display TYPE MISMATCH and go to the Main BASIC Loop.

Floating Point Operations

Floating point operations play a crucial role in BASIC. Virtually every BASIC statement that is executed makes use of some aspect of the floating point routines.

On larger computers the floating point mathematical routines are often built into the hardware of the computer. On the VIC and Commodore 64, though, all the floating point operations are carried out through software routines.

The floating point routines can be classified into functional groups that include:

- Transferring floating point variables between memory and the floating point accumulators.
- Transferring the value in one floating point accumulator to the other.
- The main floating point math operations: +,−,*, and /.
- Other floating point math and logical operations such as AND, NOT, SQR.
- The conversion routines: ASCII to floating point, floating point to ASCII, floating point to fixed point integer, and fixed point integer to floating point.
- Miscellaneous routines such as ABS, SGN, INT that did not fit into any of the above groups.

The trig routines and the routines such as LOG and EXP are discussed in their own separate sections even though they make extensive use of the floating point routines.

The section on variables discussed the format of floating point variables. The major work areas in doing floating point operations are Floating Point Accumulator 1 (FAC1) and Floating Point Accumulator 2 (FAC2). FAC1 has its exponent in 61; its four-byte mantissa in 62, 63, 64, 65; its sign in 66; and its rounding byte in 70. FAC2 has its exponent in 69; its four-byte mantissa in 6A, 6B, 6C, 6D; and its sign in 6E. FAC1 contains the result of the various floating point operations.

A couple of comments are needed about the manner in which the floating point value is stored. Excess-128 notation is used in the exponent where $80 (128 decimal) is always added to the actual exponent so that we don't have to deal with negative numbers in the exponent. The mantissa is in normalized format with the leftmost or high-order bit of the mantissa always one. The binary point is assumed to be immediately to the left of the leftmost bit.

Ways to do unnormalized floating point arithmetic exist, but are not used in BASIC. A negative sign is indicated by a high-order bit of 1 in the sign byte, while a high-order bit of 0 in the sign byte indicates a positive number. Negative numbers are stored and used in their two's complement representation. Another section in this book discusses how to directly use the floating point routines from a machine language program.

Those who believe that computers always give the correct answer are in for a surprise when they see some of the examples below of strange things that can happen in floating point routines. I am not trying to scare you away from using floating point routines. Indeed, most of the time floating point routines give you the correct answer. Just be aware that certain situations can cause unexpected errors. For an excellent discussion of the vagaries of floating point arithmetic and some of the pitfalls, see Donald Knuth's *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms.*

Following are some examples of things that Knuth pointed out that might not work properly and indeed they didn't. The final example shows one additional bug in BASIC that is due to an error in coding rather than to any inherent problem with floating point operations.

Example 1: Associative Law of Addition. This law can fail in certain situations:

Enter this BASIC Program:

```
1Ø A= (11111113 + -11111111)+7.5111111
2Ø PRINT "A = " A
3Ø B=  11111113 +(-11111111 +7.5111111)
4Ø PRINT "B = " B
```

When you run the program the computer responds with:

A = 9.5111111
B = 9.5111084

Example 2: Associative Law of Multiplication. This can also fail.

Enter this BASIC Program:

```
10 U = 20000
20 V = -6.0
30 W = 6.0000003
40 A = (U * V) + (U * W)
50 PRINT "A = " A
60 B = U * (V + W)
70 PRINT "B = " B
```

When you run this program, the computer responds with:

```
A = 3.01742554E-03
B = 6.03497028E-03
```

Example 3: Because BASIC provides no indication that underflow occurs, you can sometimes get very erroneous results, as the following example shows:

Enter this BASIC program:

```
10 U = 2.99E-39
20 V = 2.99E-05
30 W = 1.5E+35
40 PRINT U
50 PRINT V
60 PRINT W
70 A = (U * V) * W
80 PRINT "A = " A
90 B = U * (V * W)
100 PRINT "B = " B
```

When you run this program, the computer responds with:

```
2.99000001E-39
2.99E-05
1.5E+35
A = 0
B = 1.341015E-08
```

Example 4: A minus zero is not permitted.

Enter this BASIC program:

```
10 U = -0
20 V = 88.2E+05
30 W = 73.5E-27
40 PRINT "U = " U
50 X = (U * V) + W
60 PRINT "X = " X
```

When you run this program, the computer responds with:

```
U = 0
X = 7.35000001E-26
```

Example 5: This final example is due to a flaw in BASIC. During floating point addition or subtraction, if the high-order byte of the mantissa, 62, is zero, FAC1 is given a total value of zero. In this example, the number .0000000002 resulting from the subtraction is easily within the range of numbers that can be represented in floating point notation.

Enter this BASIC program:

```
10 A = .5000000005
20 B =-.5000000003
30 C = A + B
40 PRINT "C = " C
```

When you run this program, the computer responds with:

```
C = 0
```

Floating Point Moves

The following routines are used to move a floating point variable to a floating point accumulator, either FAC1 or FAC2, or to move FAC1 to memory. No direct FAC2-to-memory move exists.

Copy Floating Point Variable to Floating Point Accumulator One
B/DBA2-B/DBC6

Called by:
JSR at A/C78F FOR—move default step size of 1; JSR at A/CD42 NEXT—move step size; JSR at A/CEA2 Expression Evaluation of Next Operand for PI; JSR at A/CFA4 Locate Variable for Expression Evaluation; JSR at B/DB09 Divide FAC1 by 10; JSR at B/DF78 SQR; JSR at E0C2/E0BF RND; JSR at E2C9/E2C6 TAN.

Operation:

1. Store the accumulator and Y-register in 22 to point to the floating point variable. 22 = accumulator, the low-order byte of the pointer, and 23 = Y-register, the high-order byte of the pointer.

2. Do the following copies: (22),4 to 65; (22),3 to 64; (22),2 to 63; (22),1 to 66, thus setting the sign byte, 66, to have its high-order bit = 0 if this is a positive number or 1 if it's a negative number.
3. ORA $80 to turn on the high-order bit of this last byte loaded, which is the leftmost byte of the fraction. Remember that the high-order bit of this fraction has been used as the sign of the floating point variable and converted to 0 if it's a positive variable. All floating point variables are normalized, though, where the leftmost bit of the fraction is = 1. This ORA correctly puts this leftmost bit of the mantissa back to 1.

   Store the resulting value into 62.
4. Copy (22),0 to 61, setting the exponent for the fraction.
5. Store zero into 70, the byte used for rounding FAC1 during rounding operations.
6. RTS with accumulator = exponent and Y-register = $00.

Copy Floating Point Variable to Floating Point Accumulator Two B/DA8C–B/DAB6

Called by:

JSR at B/D850 Subtraction; JSR at B/D867 Addition; JSR at B/DA28 Multiply; JSR at B/DB0F Divide.

Besides the obvious difference of copying memory to FAC2 instead of FAC1, this routine differs from the previous one by computing a sign comparison between FAC1 and FAC2 and by not clearing the rounding byte.

Operation:

1. Store the accumulator and Y-register in (22) to point to the floating point variable. 22 = accumulator, the low-order byte of the pointer and 23 = Y-register, the high-order byte of the pointer.
2. Do the following copies: (22),4 to 6D; (22),3 to 6C; (22),2 to 6B; (22),1 to 6E; thus setting the sign byte, 6E, to have its high-order bit = 0 if this is a positive number or 1 if it's a negative number.
3. Exclusive OR the sign of FAC2 with the sign of FAC1, and store the result in 6F. If 6F is nonzero, the signs are different; if 6F is zero, the signs are the same.

4. ORA $80 to turn on the high-order bit of the leftmost byte of the mantissa. Store the resulting value into 6A. Thus FAC2 is in correct normalized format.
5. Copy (22),0 to 69, setting the exponent for the fraction.
6. RTS with accumulator = exponent and Y-register = $00.

Prepare X-Register and Y-Register for Copying FAC1 to Memory B/DBD0-B/DBD3

Called by:
JMP at A/C9D6 LET—Store FAC1 in floating point variable descriptor; JSR at A/CD52 NEXT—Update value of FOR loop variable.

Operation:
1. X-register = 49.
2. Y-register = 4A.
3. Fall through to B/DBD4 to perform copy of FAC1 to (49).

Copy Floating Point Accumulator One to Memory Pointed to by X-Register and Y-Register B/DBD4-B/DBFB

Called by:
BEQ at B/DBCE Copy FAC1 to 57–5B or 5C–60; JSR at B/D420 FN—Copy value of expression within parentheses to data area for dependent variable; JSR at B/DF88 POWER—Copy power to 4E–52; JMP at E0F6/E0F3 TAN/RND—for TAN copy to 4E–52, for RND copy to 8B–8F.

Operation:
1. JSR B/DC1B to see if 70, the rounding byte, is >= $80 (128 decimal). If so, then round up FAC1 by adding one to the low-order byte, 65, of FAC1.
2. 22 = X-register and 23 = Y-register, so that (22) points to the destination.
3. Copy FAC1 to memory area pointed to by (22). During this copy the sign byte of FAC1, 66, which has its high-order bit on if this is a negative value, is ORAed with $7F and then ANDed with 62 and the result stored at (22),1, thus setting the leftmost byte of the mantissa. If the sign is negative,

this ORA/AND/STA sequence stores a high-order bit of 1, while if positive it stores a high-order bit of 0.
4. Store $00 into 70, the rounding byte for FAC1.
5. RTS with Y-register = $00 and accumulator = the exponent.

Prepare X-Register and Y-Register for Copy of FAC1 to Memory at 0057 or 005C
B/DBC7-B/DBCF

Called by:
JSR at E05D/E05A polynomial computation; Alternate B/DBCA JSR at E044/E047 LOG/SIN/ATN; JSR at E2B4/E2B1 TAN.

Operation:
1. B/DBC7: Set X-register = $5C and fall through to step 3 by using a BIT dummy instruction.
2. B/DBCA: Set X-register = $57.
3. Set Y-register = $00.
4. Branch to B/DBD4 to copy FAC1 to either 57–5B or to 5C–60.

Transfers

The next two routines are used to copy FAC2 to FAC1 and to copy FAC1 to FAC2, thus providing a way to transfer values between the floating point accumulators. When FAC1 is copied to FAC2, the value is rounded, but no rounding occurs when copying from FAC2 to FAC1.

Copy Floating Point Accumulator Two to Floating Point Accumulator One
B/DBFC-B/DC0B

Called by:
JSR at A/CFFC AND; JMP at B/D86C ADD; Alternate B/DBFE POWER; Alternate B/DC00 EXP.

Operation:
1. Load accumulator from 6E, the sign for FAC2.
2. B/DBFE: Store accumulator into 66, the sign for FAC1.
3. B/DC00: Copy the exponent and the four mantissa bytes of FAC2, 69–6D, to FAC1, 61–65.
4. Store zero into 70, the rounding byte for FAC1.
5. RTS.

Copy Floating Point Accumulator One to Floating Point Accumulator Two with Rounding B/DC0C-B/DC1A

Called by:
JSR at A/C9FC LET—Compute TI$; JSR at B/DAFE Divide FAC1 by 10; JSR at B/DAE2 Multiply FAC1 by 10; JSR at B/DD7F Add Accumulator to FAC1; JSR at B/DF71 SQR; JSR at E26B/E268 COS; JSR at E277/E274 SIN.

Operation:
1. JSR B/DC1B to round FAC1 by adding 1 to 65, the low-order byte of FAC1, if 70, the rounding byte, is >=$80.
2. Copy FAC1, 61–66, to FAC2, 69–6E.
3. Store $00 into 70, the byte used for rounding.
4. RTS.

Floating Point Addition and Subtraction

Before considering how floating point addition or subtraction is done, first look at this example of doing floating point addition in the conventional method by hand (using base 10).

Align Exponents:

```
     4 E+01   4   E+01    or     40
+    2 E+00    .2 E+01          + 2
              --------         ----
              4.2 E+01           42
```

To perform floating point addition the exponents must be made to have the same value as that of the higher exponent. Do this by shifting the mantissa for the floating point number with the lower exponent right one position at a time and adding one to its exponent until the exponents are the same.

After the exponents are the same, then just add the mantissas to give the mantissa of the sum. The exponent of the sum is the exponent of the originally higher exponent value (to which the smaller value has now been raised). Then convert the resulting sum back into its normalized format.

BASIC uses essentially the routine just described to perform floating point addition. It has a couple of shortcuts where if it finds FAC1 equal to zero it just copies FAC2 to FAC1 and if it finds FAC2 equal to zero it just exits.

BASIC also determines how many places the low value floating point number must shift its mantissa by subtracting the exponent of FAC1 from FAC2. If the difference between exponents is equal to or greater than eight, a routine is called to shift bytes and any remaining partial byte. If the difference between exponents is less than eight, then only the number of bits less than eight is shifted in the partial byte-shift routine.

If the signs of FAC1 and FAC2 are the same, just add the mantissas of FAC1 and FAC2.

However, if the signs differ, subtract the mantissa of the floating point number that is lower in value from the higher value one. This is done by converting the lower mantissa to two's complement form and adding to the higher mantissa. If the mantissa of the lower value number is greater than the mantissa of the higher value number, convert the result in FAC1 to two's complement form. This handles floating point subtraction and addition of numbers with differing signs.

Consider the following examples of floating point addition of numbers with unlike signs, which can also be considered floating point subtraction. Assume that both exponents are equal and thus no alignment of mantissa has been made. In this case, if the exponents are equal, then FAC1 is defined to have had the originally higher value exponent.

Example 1:
Consider the case with the mantissa value of FAC1 greater than mantissa value of FAC2 for numbers in binary format:

```
  1010 FAC1
 +1011 FAC2
 -----
  0101 = 5 decimal
```

Example 2:
Consider a case with the mantissa value of FAC1 less than the mantissa value of FAC2 for numbers in binary format:

```
  1010 FAC1         10 decimal
 -1100 FAC2       - 12 decimal
 -----            -------
                  -  2 decimal
```

To do subtraction, you take the two's complement form of 1100, or 0011 + 1 = 0100, and add. Thus:

```
  1010 FAC1
 + 0100 FAC2
 -----
  1110
```

Now convert 1110 back to two's complement: 1110 = 0001 + 1 = 0010 or 2 (decimal). Converting the result to two's complement form negates the number, so the result is actually −2 (decimal).

Floating Point Addition B/D86A-B/D8D1

Called by:
Execute + during Expression Evaluation (with left operand already copied to FAC2); Fall through from Add Floating Point Variable to FAC1; JMP at B/D85F Floating Point Subtraction: JMP at B/DD8E Add accumulator to FAC1.

Entry Conditions:
Accumulator contains 61, the exponent of FAC1 (except for the JMP at B/DD8E).

Operation:

1. If the exponent of FAC1 is zero, FAC1 represents zero. Since this in effect adds 0 to FAC2, the result can be obtained by just copying FAC2 to FAC1, and this is what is done by the JMP to B/DBFC.
2. If FAC1 contains a nonzero value, then save 70, the byte used for rounding, in 56.
3. Set the X-register = $69, pointing to the exponent for FAC2.
4. Load the accumulator from 69, the exponent of FAC2, and TAY.
5. If the accumulator is zero, indicating FAC2 is zero, then exit since this means adding a value of 0 to FAC1.
6. Subtract 61, the exponent for FAC1, from the accumulator, which has the exponent for FAC2.
7. If the result is zero, the exponents are the same and no alignment needs to be done. Branch to step 21.
8. If the FAC2 exponent is less than the FAC1 exponent, BCC to step 15 with the difference in two's complement form.
9. This step is reached if the FAC2 exponent is greater than the FAC1 exponent.
10. 61 = Y-register, thus getting the original value that was in the FAC2 exponent.
11. Copy the sign of FAC2, 6E, to the sign of FAC1, 66.
12. Compute the two's complement of the difference between

exponents by doing EOR $FF, ADC $00 with the carry already equal to 1.

13. Store $00 in 56, the overflow byte.
14. X-register = 61, pointing to the floating point accumulator that was originally lower in value, and branch to step 16.
15. For FAC1 exponent > FAC2 exponent, store $00 into 70, the rounding byte.
16. Compare the two's complement difference between exponents to $F9. This checks if the difference between exponents is >= 8 (decimal), the length of one byte.
17. If the difference between the exponents is >= 8, branch to B/D862 where a JSR B/D999 is done. At B/D999 align the two floating point accumulators by rotating the mantissa of the one that has the lower exponent right, one byte at a time, and adding 8 to the originally lower exponent value, until the exponents are equal or until the difference between them is less than 8. If the difference between them is less than 8, rotate the mantissa right for this final byte, adding 1 to the exponent with each rotation, until the exponents are equal.

    Branch to step 21.
18. If the difference between the exponents is < 8, less than a full byte needs to be shifted right to make the exponents equal in value.
19. Shift $00 into the high-order bit of the first byte of the mantissa.
20. JSR B/D9B0 to rotate this mantissa right for this byte, adding one to the exponent with each rotation, until the exponents are equal.
21. If the floating point accumulators have been aligned, the accumulator = 70, which has the last byte that was shifted right from the floating point accumulator during the alignment.

    BIT 6F, which has its high-order bit on if the signs of the two floating point accumulators are unlike.
22. If the signs are the same, branch to B/D8FE to add the mantissa of FAC2 to FAC1. If adding the mantissas caused an overflow and thus left the carry set, increment the exponent, and shift the mantissa right one position while still maintaining a one in the leftmost bit. Then exit from this routine.

23. If the signs are different, determine which of the two floating point accumulators was originally lower in value. Subtract this lower one in value from the higher one, leaving the result in FAC1.
24. Since both values have been aligned to have equal exponents, it is possible that the original lower value had a higher mantissa. Fall through to B/D8D2 to determine if this is the case, and if it is, convert the result in FAC1 to two's complement form.

    Fall through to B/D8D7 to normalize FAC1.

Here are a couple of examples of the way that steps 23 and 24 work. Although the examples use base 10 notation, the principle is the same.

Example 1:
For the unlike signs computation:

```
 (  .1537) FAC1
+(-.1325) FAC2
----------
    .0212
```

Since the exponent of FAC1 is the same as that of FAC2, the sign of FAC1 is not changed before the actual subtraction in step 23. In step 24 the mantissa value of FAC2 is less than the mantissa value of FAC1, so the result is not converted to two's complement form. Rather it is just normalized.

Example 2:
For the unlike signs computation:

```
 (  .1537) FAC1
+(-.1789) FAC2
----------
   -.0252
```

Since the exponent of FAC1 is the same as that of FAC2, the sign of FAC1 is not changed before the actual subtraction in step 23. In step 24 the mantissa value of FAC2 is greater than the mantissa value of FAC1, so the result is converted to two's complement form and then normalized.

Add Floating Point Accumulators When They Have Like Signs B/D8FE-B/D91C

Called by: BPL at B/D8A5 Floating Point Addition.

Operation:

1. Add 56 to the accumulator, which contains the value of 70 at entry, and store the result in 70, the rounding byte.
2. Add the mantissa of FAC2 to FAC1, leaving the result in the mantissa of FAC1—62, 63, 64, 65.
3. JMP B/D936 to see if overflow occurred during the addition in step 2. If so, then increment the exponent of FAC1. After this increment, shift the mantissa of FAC1 right one bit, with a one being shifted back into the leftmost bit of the mantissa.

Call Routine to Align Floating Point Accumulators for Addition or Subtraction B/D862-B/D866

Called by: BMI at B/D899.

1. This routine is called if the floating point addition routine finds that it needs to align the floating point accumulators by one byte or more.
   JSR B/D999 to perform the alignment, and return with carry clear.
2. Branch to B/D8A3, step 21 of the the floating point addition routine.

Align Floating Point Accumulators for Floating Point Addition or Float-to-Fix Conversion B/D999-B/D9BB

Called by:

JSR at B/D862 Floating Point Addition; JSR at B/DCB5 Float-to-Fix Conversion; Alternate B/D9B0 JSR at B/D8A0 Floating Point Addition; JSR at B/DCC6 Float-to-Fix Conversion.

Entry Conditions:

X-register points to the exponent of the floating point accumulator to be aligned; accumulator = difference between exponents of the floating point accumulators.

Operation:

1. ADC $08 to add 8 to the difference between exponents.
2. BMI B/D985 if result is < 0 then at least another byte needs to be aligned. The difference between exponents is passed to this routine in two's complement form.

3. If adding $08 gave a result of 0, one byte remains to be shifted, so in this case also branch to B/D985.
4. Subtract $08 from the accumulator as there was no need to align a full byte, thus restoring the difference between exponents to its correct value for the following alignment of less than a byte.
5. TAY, to save this difference between exponents.
6. Accumulator = 70, loading the byte that was last shifted right out of this floating point accumulator for the alignment of bytes.
7. If subtracting $08 left the carry set, all bytes have been processed since the difference between exponents was an exact multiple of 8. Branch to step 12.
8. Otherwise, from 1 to 7 bits remain to be aligned. Continue with step 9.
9. Rotate the mantissa pointed to by the X-register right one bit.
10. Increment Y-register and if the result is not zero, branch to step 9 to do the next bit.
11. If Y-register is zero, all the bits have been shifted and the exponents of the two floating point accumulators are now equal. The accumulators are considered to be aligned when the exponents are equal.
12. CLC and RTS.

Shift Bytes for Aligning Floating Point Accumulators for Floating Point Addition or Float-to-Fix Conversion B/D985–B/D998

Called by:
BMI at B/D99B Alignment; BEQ at B/D99D Alignment; Alternate B/D983: JSR at B/DA5B Partial Product Multiplication.

Entry Conditions:
X-register points to the exponent of the floating point accumulator to be aligned; 68 = work byte of $00

Operation:
1. B/D983: X-register = $25.
2. B/D985: Shift all bytes of the mantissa of this floating point accumulator right one byte, shifting 68, an overflow work byte (that is zero in this case), into the leftmost byte of the

mantissa and shifting the rightmost byte of the mantissa into 70, the overflow byte.

3. Fall through to B/D999 to see if done with alignment.

Complement FAC1 If Carry Clear and Call Normalize B/D8D2–B/D8D6

Called by:
Fall through from floating point addition when unlike signs; JSR at B/DC55 Integer-to-FAC1 conversion; JSR at B/DCE6 INT.

Operation:

1. If carry is clear at entry, then JSR B/D947 to convert FAC1 to its two's complement representation by reversing all bits and adding one.
2. Fall through to B/D8D7 to normalize FAC1.

Make FAC1 Zero If High-order Byte of Mantissa, 62, Is Zero; Otherwise, Call Normalize B/D8D7–B/D8FD

Called by:
Fall through from Complement FAC1 if Carry Set; JSR at C/E0EC RND; JMP at B/DB9F Copy Product to FAC1; Alternate B/D8F7: BCS at B/D92E Underflow during Normalize; JMP at B/D7B2 VAL; JMP at B/DADC Overflow/Underflow Error; Alternate B/D8F9 JMP at B/DF81 POWER; Alternate B/D8FB JMP at B/DACC Add exponents of FAC1 and FAC2.

Operation:

1. If 62, the leftmost (high-order) byte of the mantissa for FAC1 is nonzero, branch to B/D929 to normalize FAC1.
2. If 62 is zero, though, FAC1 is converted to zero by the following steps. **Caution: Bug Here.** Because this check of whether to zero FAC1 only examines the high-order byte of the mantissa, very erroneous floating point results can occur. Although the situations in which they occur are infrequent, this is a design flaw that should not exist in floating point operations. Consider the following examples:

   Enter this BASIC program:

```
10 A = .500000005
20 B = −.500000003
30 C = A + B
40 PRINT C
```

When you run this program, the computer responds with:

```
2.32830644E-09
```

This example above works fine. However, move the final 5 and 3 right one more place and disaster occurs.

Enter this BASIC program:

```
10 A =   .5000000005
20 B = -.5000000003
30 C = A + B
40 PRINT C
```

When you run this program, the computer responds with:

```
0
```

The number is not too small to be represented in floating point notation. Rather, 62 has been found to be zero and thus the entire floating point accumulator is set to zero.

3. Shift the mantissa left one byte at a time until this has been done four times. First 70 is shifted into the low order mantissa byte, and then 70 is set to zero. Thus, the final value in the mantissa consists of a high byte with the original value of 70 and three low-order bytes of zero.
4. B/D8F7 Set accumulator = $00.
5. B/D8F9 Store accumulator into 61, the exponent of FAC1.
6. B/D8FB Store accumulator into 66, the sign of FAC1.
7. RTS.

Normalize Floating Point Accumulator One B/D91D-B/D946

Called by:
Actual Entry Point B/D929—BNE at B/D8DD; Alternate Entry B/D936 Floating Point Add for Like Signs; Alternate Entry B/D938 JSR at B/DC28 ROUND.

Entry Conditions: Accumulator = $00 if entry at step 2.

Operation:

1. B/D91D: Add one to the accumulator, which accumulates the number of corrections made, and then shift all bits in the mantissa of FAC1 left one bit, shifting into the low-order bit from the high-order bit of 70.

2. Actual Entry Point B/D929: BPL B/D91D—branch if 62, the leftmost byte of the mantissa, does not yet have its high-order bit set to 1.
3. Now that the mantissa has been normalized, subtract 61, the starting exponent, from the accumulator which contains the number of shifts made to normalize the mantissa.
4. If carry is set as a result, underflow has occurred, so branch to B/D8F7 to handle the underflow error.
5. Using the result from step 3 in the accumulator, EOR $FF and ADC $01, thus converting back to the correct form for the exponent, and store in 61, the exponent for FAC1.

   As an example for steps 3–5, consider that a value that had an original exponent of $83 and 6 shift lefts were required to produce a normalized mantissa:

```
 0000 0110      $06
+0111 1101      −$83(in 2's comp)
──────────
 1000 0011
 0111 1100      EOR $FF
 000  0001      (Carry 0) ADC $01
──────────
 0111 1101 =$7D
```

6. B/D936: If carry is clear, then RTS since no overflow occurred in step 5 or in the floating point addition of mantissas for two like signs.
7. If carry is set, then overflow occurred when adding the mantissas for two like signed numbers.
8. B/D938: Increment 61, the exponent for FAC1.
9. If 61 is now zero, branch to B/D97E to handle the OVERFLOW ERROR of the exponent.
10. If the exponent did not overflow, shift the mantissa right one position, shifting a carry into the high-order bit.
11. RTS.

Add Floating Point Variable in Memory to Floating Point Accumulator B/D867-B/D86C

Called by:
JSR at A/CD4F NEXT; JSR at B/DA01 LOG; JSR at B/DA1D LOG; JSR at B/D84D Add .5 to FAC1; JSR at E081/E07E Polynomial Computation; JSR at E0D0/E0CD RND; JSR at E2A4/E2A1 SIN; JSR at E268/E265 COS.

Entry Conditions:
Accumulator has low-order byte of address of floating point variable, Y-register has high-order byte of address of floating point variable.

Operation:

1. JSR B/DA8C to copy the floating point variable pointed to by the accumulator and Y-register to FAC2. Set the sign of FAC2 according to the high-order bit of the leftmost byte of the mantissa, and turn this high-order bit on before storing in the mantissa of FAC2. Also set 6F to be 1 if the signs of FAC1 and FAC2 are different.
2. Fall through to B/D86A to perform floating point addition, adding FAC2 to FAC1 and leaving the result in FAC1.

Floating Point Subtraction—Subtract Floating Point Accumulator One from Floating Point Accumulator Two B/D853–B/D861

Called by:
Execute — During Expression Evaluation (with left operand already copied to FAC2); JSR at E02D/E02A EXP; JSR at E281/E27E SIN.

1. Reverse 66, the sign of FAC1 to convert the right operand of the — to a negative number.
2. Exclusive OR this new value of 66 with 6E, the sign for FAC2, and store the result in 6F. 6F now has its high-order bit on if the signs are different.
3. Accumulator = 61, the exponent for FAC1.
4. JMP B/D86A to perform floating point addition.

Subtract Floating Point Accumulator One from Floating Point Variable in Memory B/D850–B/D852

Called by:
JSR at B/DA0F LOG; JSR at E288/E285 SIN; JSR at E334/E331 SIN.

Entry Conditions:
Accumulator has low-order byte of address of floating point variable, Y-register has high-order byte of address of floating point variable.

Operation:

1. JSR B/DA8C to copy the floating point variable pointed to by the accumulator and Y-register to FAC2. Set the sign of FAC2 according to the high-order bit of the leftmost byte of the mantissa, and turn this high-order bit on before storing in the mantissa of FAC2. Also set 6F to be 1 if the signs of FAC1 and FAC2 are different.
2. Fall through to B/D853 to subtract FAC1 from FAC2, leaving the result in FAC1.

Floating Point Multiplication

In an overall view floating point multiplication seems quite simple. To multiply two floating point numbers, you just add the exponents and multiply the mantissas. For example, if A= 5E+07 and B = 3E+19, then C= A * B gives a product of 15E+26 (although the result is printed as 1.5E+27). Negative numbers must be in two's complement form before multiplication.

When adding the exponents, you need to check for possible overflow or underflow:

Consider the exponents:

```
  $83   =          1000 0011
 +$86   =          1000 0110
 ---------------------------
  $09   Carry=1    0000 1001
```

If the carry is set and the high-order bit is zero, neither overflow nor underflow has occurred. Add $80 to convert the number back to excess-128 notation (same as subtracting $80 since two's complement of $80 is $80).

$09 + $80 = $89

Consider the exponents:

```
  $FE   =          1111 1110
 +$86   =          1000 0110
 ---------------------------
  $84   Carry=1    1000 0100
```

If the carry is set and the high-order bit is one, overflow has occurred.

Consider the exponents:

```
  $3F   =          0011 1111
 +$40   =          0100 0000
 ----------------------------
  $7F   Carry=0    0111 1111
```

If the carry is clear and the high-order bit is zero, underflow has occurred. Underflow means that this value is less than the smallest value that can be represented on the computer, which for the VIC and Commodore 64 is 2.93873588E−39.

−39 is represented in excess-128 notation as 89 decimal or hex 59 = 0101 1001.

Before explaining how BASIC handles multiplication of the mantissas, examine this conventional example of doing binary multiplication:

```
  220(decimal) = $DC=                         1 1 0 1 1 1 0 0
X 178(decimal) = $B2=                         1 0 1 1 0 0 1 0
---------------                             -----------------
39,160(decimal)                               0 0 0 0 0 0 0 0
 = $98F8                                    1 1 0 1 1 1 0 0
                                          0 0 0 0 0 0 0 0
                                        0 0 0 0 0 0 0 0
                                      1 1 0 1 1 1 0 0
                                    1 1 0 1 1 1 0 0
                                  0 0 0 0 0 0 0 0
                                1 1 0 1 1 1 0 0

                              -------------------------------
=                             1 0 0 1 1 0 0 0 1 1 1 1 1 0 0 0
= $                               9       8       F       8
```

Notice how for each bit in the multiplier ($B2), the partial product written down is shifted one bit to the left. Also notice that when a bit in the multiplier is 1, the multiplicand value is copied down, while if it is 0, just zeros are copied down. After all partial products are finished, the products are added to give the final product. This summation of partial products could occur after each bit is processed in the multiplier without any change in the answer.

Okay, so now comes the problem of just how you actually do multiplication of the mantissas of the floating point accumulators, each mantissa being four bytes long. The multiplier also used 70, the rounding byte, as its low-order byte for a total of five bytes. The first problem you might notice is that the final product takes up to twice as many bits to represent the answer as the bits needed to represent the multiplicand or multiplier. Since the partial product is stored in locations 26, 27, 28, 29, and an overflow byte of 70, some bits must be dropped. If you drop bits off the left side (high-order) of the answer, you will get a very inaccurate answer. The only alternative in this limited situation is to drop bits off the right (low-order), losing some accuracy, but not nearly as much as if you had dropped bits from the left.

The second problem is how do you keep shifting the partial product left when only four bytes are allocated for it? You don't. Instead, since you are going to drop the rightmost bits anyway, just drop off a bit when preparing the new partial product by shifting the partial product right one bit. And instead of waiting until all partial products have been added to give a final product, you add partial products as you go along, and after the final bit of the multiplier is processed, the partial product contains the final product.

One last problem remains. The multiplier is five bytes, not just one. All that needs to be done to handle this is to load one byte at a time from the multiplier, working from low-order (right side) to high-order (left side), and do this multiplication for each byte in turn. One shortcut occurs at this point. If the byte is zero, multiplication by this byte does not occur. Rather, the partial product is just shifted to the right by one byte to save the effort of having to do eight individual shifts without adding the multiplicand. One slight difference occurs for the final high-order byte. Since the FAC1 is normalized, this final byte has to contain at least a high-order 1. Thus the multiplication must be done, and when this byte is finished the partial product contains the final product.

After the final product has been obtained, FAC1 is normalized. Not much normalization is needed though, since at most the mantissa needs to be shifted one bit to the left to produce a normalized number. Why? Consider: .1 * .1 = .01 and both multiplicand and multiplier must be normalized for floating point multiplication.

Floating Point Multiplication—Multiply FAC1 by FAC2 Leaving Product in FAC1 B/DA2B-B/DA58

Called by:

Execute * During Expression Evaluation; Fall through from

Multiply Floating Point Variable in Memory by FAC1.

Operation:

1. At entry the accumulator contains the exponent of FAC1. If this exponent is zero, just exit since multiplying by zero produces a zero result no matter what value FAC2 has.
2. JSR B/DAB7 to add the exponents of FAC1 and FAC2, leaving the result in FAC1 and also setting the sign of FAC1 from the Exclusive OR of the signs of FAC1 and FAC2. Test for possible underflow or overflow. If overflow results, display an error message. If underflow results, just set FAC1 to zero and exit this routine.
3. Store $00 into 26, 27, 28, and 29, which are used as the partial product area during multiplication.
4. For 70, the rounding byte of FAC1, and for the mantissa bytes 65, 64, and 63: Load the accumulator with each of these bytes in succession, and for each byte JSR B/DA59 to multiply this byte by FAC2, shifting the partial product right one byte and adding this byte's product to the partial product.
5. For all of the previous bytes a $00 was acceptable, and if found, the partial product was just shifted right one byte and the $00 was not multiplied by FAC2. However, for 62, the first byte of the mantissa which is in normalized format, the high-order bit must be 1. Thus multiplication must occur, so for this 62 byte JSR B/DA5E to force multiplication for this last byte, leaving the result in the final partial product of 26, 27, 28, 29.
6. JMP B/DB8F to copy the partial product at 26–29 to 62–65, the mantissa for FAC1. For a description, see the floating point division routines.

Multiply Each Byte of FAC1 by FAC2 Leaving Result in the Partial Product 26–29
B/DA59–B/DA8B

Called by:

JSR at B/DA3F, B/DA44, B/DA49, B/DA4E Floating Point Multiplication; Alternate B/DA5E: JSR at B/DA53 Floating Point Multiplication.

1. If the accumulator is zero, JMP B/D983 to shift the partial product right one byte, shifting 68 into the leftmost byte of the partial product, 26. 68 is typically zero.

2. If the accumulator is not zero, we actually have a value from this byte of FAC1 to multiply by FAC2.
3. B/DA5E: LSR to shift the low-order bit into the carry.
4. ORA $80 to turn on the high-order bit to force all eight bits to be tested in this byte.
5. TAY to hold the byte from FAC1 that is being multiplied.
6. If carry is clear from last LSR, branch to step 8 to skip the addition of FAC2 to the partial product for this bit.
7. Add FAC2 to the partial product 26–29, storing the result back into 26–29. Note that the final add of 26 and 6A could leave the carry set, and then the following rotate sends a 1 into the leftmost bit of the partial product.
8. Rotate the partial product right one bit, including 70 as the low-order byte.
9. Restore this byte to the accumulator from the Y-register.
10. LSR.
11. If after the LSR the accumulator is now 0, all eight bits have been tested, so RTS. If not yet 0, branch to step 5.

Handle Overflow/Underflow/Zero B/DAD4–B/DAE1

Called by:
JSR at E00B/E008 EXP; Alternate B/DAD8: BPL at B/DAC4 Underflow During Add Exponents for Floating Point * or /; Alternate B/DADA: BEQ at B/DAB9 If FAC2 Exponent Is 0 During Add Exponents for F/P. * or /; Alternate B/DADF: BMI at B/DAC0 Adding Exponents; BEQ at B/DB23 FAC2/ FAC1; BCS at B/DAEB FAC1 * 10; BEQ at B/DAF6 FAC1 * 10.

Operation:
1. Load the accumulator from 66, the sign for FAC1.
2. Exclusive OR with $FF.
3. B/DAD8: BMI B/DADF to display OVERFLOW ERROR. Fall through if branch here for underflow condition.
4. B/DADA: Pull the return address of this routine from the stack, then JMP B/D8F7 to store $00 into 61, FAC1 exponent, and 66, FAC1 sign and RTS to return two levels back in the calling sequence.
5. B/DADF: JMP B/D97E to display OVERFLOW ERROR and go to the Main BASIC Loop.

Multiply FAC1 by 10 (Decimal) B/DAE2-B/DAF8

Called by:
JSR at A/C9F2, A/CA0E LET; JSR at B/DD5B, B/DD71 Convert ASCII String to Floating Point; JSR at B/DE21 Convert Floating Point to ASCII String; Alternate Entry B/DAED JSR at A/CA04 LET.

Operation:

1. JSR B/DC0C to copy FAC1 to FAC2.
2. Transfer the accumulator, which contains the exponent, to the X-register, to provide the setting to indicate whether the exponent is zero.
3. If the exponent is zero, then RTS.
4. Add 2 to the exponent, having the effect of two left shifts or multiplying by 4.
5. If carry is set after add, display OVERFLOW ERROR and exit to the Main BASIC Loop.
6. B/DAED: Store $00 into 6F to indicate that the signs of the two floating point accumulators are the same.
7. JSR B/D877 to add FAC1 to FAC2, storing the result back into FAC1. Thus FAC1 is now equal to five times its original value.
8. Increment 61, having the effect of another multiply of FAC1 by 2, thus giving FAC1 a value of the original value of FAC1 at entry to this routine times ten.
9. If 61 is now zero, display OVERFLOW ERROR and exit to the Main BASIC Loop.
10. RTS.

Multiply Floating Point Variable by FAC1 B/DA28-B/DA2A

Called by:
JSR at B/DE04 Convert Floating Point to ASCII; JSR at B/DFAA POWER; JSR at B/DFF1 EXP; JSR at E04C/E049, E056/E053—Function Series Evaluation for LOG, SIN, ATN; JSR at E0C9/E0C6 RND; JSR at E070/E06D Polynomial Computation.

Entry Conditions:
Accumulator has the low-order address of address of floating point variable and Y-register has the high-order address.

Operation:

1. JSR B/DA8C to copy the floating point variable from the location addressed by the accumulator and Y-register to FAC2. Also set 6F to have its high-order bit on if the signs differ.
2. Fall through to B/DA2B to perform floating point multiplication of FAC1 * FAC2.

Add Exponents of Floating Point Accumulators for Multiplication or Division B/DAB7-B/DAD3

Called by:
JSR at B/DA30 Floating Point Multiplication; JSR at B/DB1E Floating Point Division; Alternate Entry B/DAB9; JSR at E03F/E03C EXP.

Operation:

1. Load the accumulator from 69, the exponent of FAC2.
2. B/DAB9: If the accumulator = $00, branch to B/DADA in Handle Overflow/Underflow/Zero to also store $00 into the exponent and sign of FAC1 since dividing into zero or multiplying by zero gives a result of zero.
3. Add 61, the exponent of FAC1, to the accumulator which contains the exponent of FAC2.
4. If carry is clear, definitely no overflow occurred, so branch to step 7.

   As an example, consider the exponents:

```
  $83   =          1000 0011
+ $76   =          0111 0110
----------------------------
  $F9  Carry=0     1111 1001
```

5. BMI B/DADF to display OVERFLOW ERROR if the carry is set and the high-order bit is also set. Consider the example:

```
  $FE   =          1111 1110
+ $89   =          1000 1001
----------------------------
  $87  Carry=1     1000 0111
```

6. If the high-order bit = 0 and carry is set, then CLC and use a BIT instruction to fall through to step 8.

```
   $83   =          1000 0011
 + $86   =          1000 0110
 ----------------------------
   $09   Carry=1    0000 1001
```

7. BPL to B/DADA if underflow occurs, which is indicated by carry clear and high-order bit = 0. Consider the example:

```
   $40   =          0100 0000
 + $3F   =          0011 1111
 ----------------------------
   $7F   Carry=0    0111 1111
```

8. If overflow or underflow did not occur, correct the exponent by adding $80 to convert the exponent back to excess-128 notation. Consider the valid exponents from step 6—$09, and from step 4—$F9. After adding $80 to each of these, the result is $89 in the first case and $79 in the latter.
9. Store the accumulator in 61 to set the new value of the exponent for FAC1.
10. If the result is zero, then JMP B/D8FB to also store zero into 66, the sign byte for FAC1, and exit.
11. If the exponent is not zero, then store 6F, which has its high-order bit on if the signs of the floating point accumulators are different, into 66, the sign for FAC1.
12. RTS.

Floating Point Division

Floating point division uses the conventional manner of doing division by hand with a few adjustments. First examine the following example of conventional division.

```
(In Decimal)          (In Binary)
     5                Quotient→          101
    ___                                 ______
4 | 21                Divisor→ 100 |  10101    ←Dividend
    20                                100
    --                                -----
     1                                   10    ←New Dividend
                                          0
                                      -----
                                        101    ←New Dividend
                                        100
                                      -----
                                          1    ←Remainder
```

As for the exponents in the binary example, the 100 has an exponent of +3 and 10101 has an exponent of +5. (Ignore the excess-128 notation for this example.) To compute the exponent of the quotient: 5−3+1=3 so 101 or 5 (decimal) is the quotient and the remainder is 1.

Binary division is simpler than decimal division in one aspect—you don't have to estimate how many times the divisor goes into the dividend. If the dividend is greater than or equal to the divisor, then 1 is placed in the quotient and the divisor is subtracted from the dividend. If the dividend is less than the divisor, a 0 is placed in the quotient and the dividend is left unchanged. After each bit is divided, the new dividend that results is shifted right one bit. Division can continue for however many bits you want in your quotient. The quotient exponent will be used to take this raw quotient and convert it to normalized form.

In BASIC the dividend is the mantissa of FAC2 (6A 6B 6C 6D), and the divisor is the mantissa of FAC1 (62, 63, 64, 65). The quotient is stored in 26, 27, 28, 29 and in the two high-order bits of 70. After division ends, which it does after 34 comparisons have been made (4*8+2), the quotient in 26–29 is copied to the mantissa of FAC1 and the mantissa of FAC2 contains the remainder. The quotient is then normalized using the exponent of the quotient that has been computed by FAC2 exponent − FAC1 exponent + 1.

After each comparison of the unchanging divisor to the dividend, a 1 or 0 is placed in the carry which is rotated left into a quotient work byte. After each eight bits, this quotient work byte is stored into one of 26–29. If a 1 was placed into the carry, a new dividend is formed by subtracting the FAC1 mantissa from the FAC2 mantissa. After each compare, the dividend (FAC2 mantissa) is rotated left one bit. After four quotient work bytes have been computed, two more comparisons of divisor to dividend and setting of the carry are done to provide two high-order bits for 70.

Whew! If you got lost in the above discussion, maybe this example of how BASIC handles the mantissa part of floating division with a purposely shortened dividend and divisor will help.

Example 1:

Dividend = 10101 Normalized Format
Divisor = 10000 Normalized Format
Quotient Work Byte = 0000 0001

Dividend is >= divisor. Shift a 1 into the quotient work byte and subtract the divisor from the dividend.

Dividend = 00101
Quotient Work Byte = 0000 0011
Rotate the dividend left one bit.
Dividend = 01010
Carry = 0

Example 2:

Dividend = 01010 Carry = 0
Divisor = 10000
Quotient Work Byte = 0000 0011

Since the carry is clear and the high-order bit of the dividend is 0, no comparison of divisor to dividend is necessary since the normalized divisor is clearly greater than the dividend. Shift a 0 into the quotient work byte and leave the dividend unchanged.

Quotient Work Byte = 0000 0110
Rotate the dividend left one bit.
Dividend = 10100
Carry = 0

Example 3:

Dividend = 10100 Carry = 0
Divisor = 10000
Quotient Work Byte = 0000 0110

Since the carry is clear and the high-order bit of the dividend is 1, a comparison is needed to determine if the dividend is >= divisor. The comparison finds dividend >= divisor, so shift a 1 into the quotient work byte and subtract the divisor from the dividend.

Dividend = 00100
Quotient Work Byte = 0000 1101
Rotate the dividend left one bit.
Dividend = 01000
Carry = 0

Example 4:
Dividend = 01000 Carry = 0
Divisor = 10000
Quotient Work Byte = 0000 1101

Since the carry is clear and the high-order bit of the dividend is 0, no comparison of the divisor to the dividend is done by BASIC since the normalized divisor has to be greater than the dividend. Shift a 0 into the quotient work byte and leave the dividend unchanged.

Quotient Work Byte = 0001 1010
Rotate the dividend left one bit.
Dividend = 10000
Carry = 0

Example 5:
Dividend = 10000 Carry = 0
Divisor = 10000
Quotient Work Byte = 0001 1010

Since the carry is clear and the high-order bit of the dividend is 1, a comparison is needed to determine if the dividend is >= the divisor. The comparison finds the dividend >= the divisor, so shift a 1 into the quotient work byte and subtract the divisor from the dividend.

Dividend = 00000
Quotient Work Byte = 0011 0101
Rotate the dividend left one bit.
Dividend = 00000
Carry = 0

Example 6:
Dividend = 00000 Carry = 0
Divisor = 10000
Quotient Work Byte = 0011 01010

We have now reached a stage in the division where the dividend is always going to be less than the divisor. Each of these steps would rotate the dividend left and keep rotating zeros into the quotient work byte since the dividend is always less than the divisor. The rotates end when the rotate of the quotient work byte rotates a one into the carry. Thus the final status of the quotient work byte is Quotient Work Byte = 1010 1000.

Since the exponent as computed previously is 3, this is actually 101.01 or 5.25 (decimal), which is exactly the result you expect when dividing 21 (decimal) by 4 (decimal).

Floating Point Division—Divide Floating Point Accumulator Two by Floating Accumulator One Leaving Quotient in FAC1 B/DB12—B/DB8E

Called by:
Execute / During Expression Evaluation (with FAC2 Containing the Dividend); JSR at B/DB0C Divide FAC1 by 10 (with Original FAC1 Already Copied to FAC2).

In the following discussion, references to FAC1 and FAC2 refer to just their mantissa parts.

Operation:

1. The accumulator contains the exponent of FAC1 at entry. If this exponent is zero, branch to step 31 to display DIVISION BY ZERO error message.
2. JSR B/DC1B to see if 70, the rounding byte, is >= $80. If so, then round up FAC1.
3. Compute the two's complement value of exponent of FAC1 and store back into 61.
4. JSR B/DAB7 to add the exponents of FAC2 and FAC1. Since FAC1 is in two's complement form, this is the same as FAC2 exponent minus FAC1 exponent.
5. Increment 61, the FAC1 exponent. This is part of the algorithm that produces the correct exponent in the result. The examples in the previous section gave illustrations of how this worked.
6. If 61 is zero, display OVERFLOW ERROR and return to the Main BASIC Loop.
7. Set the X-register = $FC. The X-register is used as an index when storing each byte of the quotient into 26, 27, 28, and 29.
8. Set the accumulator = $01. This will force all eight bits to be rotated left into the accumulator as the quotient. The accumulator will be rotated left until the carry is 1.
9. Compare FAC1 to FAC2. FAC1 is the divisor and FAC2 the dividend. Comparison is of the Y-register holding successive values from FAC2 to the successive bytes in FAC1.
10. Save the status of the compare on the stack. If the dividend is greater than or equal to the divisor, the carry is set.
11. Rotate the accumulator left, which is the counter to force eight bits to be examined at a time. The accumulator also

serves as the one-byte quotient and the rotate left rotates the carry into the low-order bit. If the dividend is >= the divisor, a 1 is rotated in. If the dividend is < the divisor, a 0 is rotated in.

12. If the carry is clear, eight bits have not yet been shifted, so branch to step 18.
13. After eight bits have been handled, we have a new byte to store into the quotient. Increment the X-register to point to the next byte of the quotient.
14. Store the accumulator at 29,X. Since the X-register is initially $FC, this stores at 29+$FD=26, 29+$FE=27, 29+$FF=28, 29+$00=29 with each successive byte.
15. If the X-register is now zero, branch to step 26 to handle the division for the rounding byte, 70.
16. BPL to step 27 after division by the rounding byte is complete.
17. The X-register is still negative if branches from step 15 or step 16 are not taken. Set the accumulator = $01 to again force eight bits for the quotient.
18. Restore the status of the compare, with the carry set if the dividend >= the divisor.
19. If the carry is set, branch to step 24 to subtract the divisor (FAC1) from the dividend (FAC2) to form a new dividend from the remainder.

    If the carry is clear, there's no need to form a new dividend from the remainder. Continue with step 20.
20. Shift the dividend (FAC2) left one bit, with the high-order bit of the mantissa rotated into the carry. Shift a zero into the low-order bit of the mantissa.
21. If the carry is set, there is no need to compare the divisor to the dividend since the dividend is clearly greater than or equal to the divisor if the carry is set after this rotate. Branch with carry set to step 10.

    For example, if the previous compare left state 1 below, after rotating left, the carry is set in state 2 and the dividend >= the divisor.

State 1:

Dividend		11001011
Divisor		11110000

State 2:

Dividend	C=1	10010110
Divisor		11110000

22. If the carry is clear and the high-order bit of the dividend is equal to 1, an actual comparison of the divisor to the dividend needs to be made. Branch to step 9.
23. If the carry is clear and the high-order bit of the dividend is zero, no compare is needed in this case because the normalized divisor (with a 1 in its high-order bit) is clearly greater than the dividend. Branch with carry clear to step 10.
24. The dividend is >= the divisor when this step is branched to from step 19. A new dividend is formed from the remainder of subtracting the divisor from the dividend. This remainder becomes the new dividend. This is accomplished by subtracting FAC1 from FAC2 and storing the result back into FAC2.
25. JMP to step 20 to rotate the dividend left one bit and continue computing the quotient.
26. Branch here from step 15 after the four-byte quotient in 26–29 has been produced. Set the accumulator to $40 to force two more quotient bits to be produced in the accumulator, because after two rotate lefts of the accumulator, the carry will be set. Branch to step 18.
27. Branch here from step 16 after these last two bits of the quotient are done. Shift the accumulator left six bits since only bits 1 and 0 were set in the division for the rounding byte.
28. Store the accumulator in 70, the rounding byte.
29. Pull the last status from the stack to keep the stack in order.
30. JMP B/DB8F to copy 26–29, the quotient, to 62–65, the mantissa for FAC1, and then normalize FAC1. At exit FAC2 contains the remainder.
31. Branch here from step 1 if FAC1 is zero to branch to A/C437 to display DIVISION BY ZERO error and return to the Main BASIC Loop.

Copy Quotient or Product to Mantissa of Floating Point Accumulator One B/DB8F-B/DBA1

Called by:
JMP at B/DA56 Floating Point Multiplication; JMP at B/DB87 Floating Point Division.

Operation:
1. 26–29 contains the product of the multiplication routine or the quotient for the division routine. Copy 26–29 to 62–65, the mantissa for FAC1.
2. JMP B/D8D7 to normalize FAC1.

Divide Floating Point Variable by FAC1 B/DB0F-B/DB11

Called by:
JSR at B/DA08 LOG; JMP at E2D9/E2D6 TAN; JSR at E321/E31E ATN.

Entry Conditions:
Accumulator contains low-order byte of address of floating point variable and Y-register contains high-order byte of address.

Operation:
1. JSR B/DA8C to copy the floating point variable pointed to by the accumulator and the Y-register to FAC2.
2. Fall through to B/DB12 to divide FAC2 by FAC1, leaving the quotient in FAC1 and the remainder in FAC2.

Divide FAC1 by 10 B/DAFE-B/DB0E

Called by:
JSR at B/DD52 Convert ASCII to Floating Point; JSR at B/DE28 Convert Floating Point to ASCII; Alternate B/DB07: JSR at E274/E271 SIN.

Operation:
1. JSR B/DC0C to copy FAC1 to FAC2.
2. Set the accumulator = $F9 and the Y-register = $BA (Commodore 64) or $DA (VIC). At B/DAF9 is the value of decimal 10 in floating point format.
3. Set 6F to zero to indicate the signs are the same.
4. JSR B/DBA2 to copy the floating point number pointed to by the accumulator and the Y-register (10 decimal) to FAC1.
5. JMP B/DB12 to divide FAC2 by FAC1, leaving the quotient in FAC1 and the remainder in FAC2.

Miscellaneous Floating Point Math and Logic Routines

Included below are the detailed descriptors for various math and logical operations that affect the floating point accumulators: Add .5 to FAC1, Round FAC1 ↑(Exponentiation), SQR, NOT, Monadic Minus (Negate), AND, and OR.

Add .5 to FAC1 to Round FAC1 to Next Integer B/D849-B/D84F

Called by:
JSR at B/DE2F Convert Floating Point to ASCII or TI$; JSR at E290/E28D SIN.

Operation:
1. Load the accumulator with $11 and the Y-register with $BF (Commodore 64) or $DF (VIC) to point the floating point representation of .5 at B/DF11.
2. JMP B/D867 to copy .5 to FAC2 and then add FAC2 to FAC1.

Round FAC1 Based on Rounding Byte B/DC1B-B/DC2A

Called by:
JSR at A/C9C4 LET; JSR at A/CE43 Expression Evaluation—Stack Left Operand on Stack; JSR at B/DB14 Divide FAC2 by FAC1; JSR at B/DBD4 Copy FAC1 to Memory; JSR at B/DC0C Copy FAC1 to FAC2; Alternate B/DC23: JSR at B/DFFA EXP.

Operation:
1. If the exponent, 61, of FAC1 is zero, then exit.
2. ASL 70 to shift high-order bit into the carry.
3. If the carry is clear, then 70 was < $80 so no round up, just RTS.
4. If the carry is set, then 70 was >= $80 so round up FAC1. JSR B/D96F to increment the low-order byte, 65, of FAC1.
5. If FAC1 was incremented without overflow, then exit.
6. If overflow occurred during increment, shift the mantissa right one bit and increment the exponent by one. If overflow occurs when incrementing the exponent, then display OVERFLOW error message and return to the Main BASIC Loop.

↑ (Exponentiation) Operation B/DF7B–B/DFB3

Called by:
Execute Expression for ↑ with FAC2 containing left operand; fall through from SQR.

Operation:

1. At entry the accumulator contains 61, the exponent for FAC1. If this is zero, branch B/DFED to compute EXP(0). For example, 3↑0=1. EXP(0)=1. Any number raised to the zero power is = 1.
2. If 69, the exponent of FAC2 for the left operand is zero, then JMP B/D8F9 to set 61 and 66 to zero, thus setting FAC1 to zero. For example, 0↑5=0.
3. JSR B/DBD4 to copy FAC1 (the power) to 4E 4F 50 51 52.
4. If the left operand is positive, branch to step 11.
5. If the left operand is negative, then JSR B/DCCC to INT to convert FAC1 to an integer. 07 = low-order byte of the integer value.
6. Set the accumulator to $4E and Y-register to $00 to point to the power that is stored at 4E–52.
7. JSR B/DC5B to compare the floating point variable at 4E–52, the power, to FAC1, which is the integer part of the power. If the integer part = original value, then the accumulator = 0; if the integer part < power, then the accumulator = $01; if the integer part > power, then the accumulator = $FF.
8. If the accumulator is not zero, then power is not an integer, so branch to step 11.
9. Set the accumulator = $00.
10. Load the Y-register from 07, the integer of the power.
11. JSR B/DBFE to set the sign of FAC1 to positive and to copy FAC2 to FAC1.
12. TYA and PHA to save the sign-change flag.
13. JSR B/D9EA to perform LOG on FAC1, taking the LOG of the power.
14. Load the accumulator with $4E and the Y-register with $00 to point to the power that is stored at 4E-52.
15. JSR B/DA28 to multiply the power by FAC1, which contains LOG(X) if we consider the operation as X↑N.
16. JSR B/DFED to perform EXP on FAC1, thus doing EXP(N*LOG(X)).

17. Pull the sign status from the stack and LSR. If the carry is clear, no sign change is needed, so just exit.
18. If the carry is set, fall through to B/DFB4 to do NOT.

Consider the following short BASIC program that handles the above algorithm for positive numbers. Lines 10–20 calculate 5↑3 using the ↑ operator, and lines 30–100 use the above algorithm.

```
10 A=5↑3
20 PRINT "A="A
30 B=LOG(5)
40 PRINT "B="B
50 C=3
60 PRINT "C="C
70 D=B*C
80 PRINT "D="D
90 E=EXP(D)
100 PRINT"E="E
```

When the program is run, the computer responds with:

```
A= 125
B= 1.60943791
C= 3
D= 4.82831374
E= 125
```

SQR
B/DF71–B/DFA

Called by: Invoke Function.

Operation:

1. JSR B/DC0C to copy FAC1 to FAC2 with rounding.
2. Set the accumulator = $11 and the Y-register = $BF (Commodore 64) or $DF (VIC) to point to B/DF11, which contains the constant .5 in floating point format.
3. JSR B/DBA2 to copy .5 to FAC1.
4. Compute (FAC2)↑.5 by falling through to the evaluate ↑ operation at B/DF78.

Monadic Minus (Negate)
B/DFB4–B/DFBE

Called by:
Execute Monadic Minus During Expression Evaluation; JMP at

E33A/E337; JSR at E313/E310 ATN; JSR at E29D/E29A SIN; JSR at E2AA/E2A7 SIN; JSR at E030/E02D EXP; JSR at B/DD67 Convert String to Floating Point.

Operation:
1. If 61, the exponent for FAC1 is zero, then just RTS.
2. Exclusive OR 66, the sign for FAC1, with $FF, and store in 66, thus reversing the sign.
3. RTS.

NOT
A/CED4–A/CEE2

Called by: Execute NOT During Expression Evaluation.

Operation:
1. JSR B/D1BF to convert the operand of the NOT to an integer in FAC1. If > 32767, display ILLEGAL QUANTITY. The two-byte integer is left in 64 and 65.
2. Exclusive OR 64 with $FF and store in the Y-register, then Exclusive OR 65 with $FF, thus reversing the values of these two bytes.
3. JMP B/D391 to convert the integer to a floating point number in normalized format in FAC1.

AND
A/CFE9–B/D015

Called by: Execute AND During Expression Evaluation.

Operation:
1. Set the Y-register to $00 to set the AND indicator.
2. Store the Y-register into 0B, which now is $00 if doing AND or $FF if doing OR.
3. JSR B/D1BF to convert the right operand to an integer, leaving the results in 64 and 65. If the value from the right operand is > 32767, then display ILLEGAL QUANTITY.
4. Exclusive OR 64 with 0B, store the result in 07; Exclusive OR 65 with 0B, store the result in 08. Leaves the value unchanged if AND, reverses value if OR.
5. JSR B/DBFC to copy FAC2, which holds the left operand, to FAC1.
6. JSR B/D1BF to convert the left operand to an integer, leaving the result in 64 and 65. If the value from the right operand is > 32767, display ILLEGAL QUANTITY.

7. Do the following sequence: LDA 65, EOR 0B, AND 08, EOR 0B.
8. Do the following sequence: LDA 64, EOR 0B, AND 07, EOR 0B, TAY.
9. JMP B/D391 to convert the integer in the accumulator and Y-register to a normalized floating point number in FAC1.

Consider the examples for AND and OR:

3 AND 5	3 OR 5	
0011	0011	
0101	0101	
0001	0111	<=CONVENTIONAL

Step 4:

	FOR AND	FOR OR
65 =	0101	0101
EOR 0B	0000	1111
08	0101	1010

Step 7

LDA 65	0011	0011
EOR 0B	0000	1111
	0011	1100
AND08	0101	1010
	0001	1000
EOR 0B	0000	1111
	0001	0111

OR
A/CFE6–A/CFE8

Called by: Execute OR During Expression Evaluation.

Operation:
1. Set the Y-register = $FF to set the value for doing OR.
2. Fall through to do AND at A/CFEB.

ASCII Decimal Number to Binary Floating Point Conversion

Decimal numbers can be converted to their representation in floating point format in a very straightforward manner. The decimal number is scanned from the left. Before each digit is evaluated, the current accumulated value in the floating point accumulator is multiplied by ten. The digit $00–$09 obtained by subtracting $30 from the ASCII format is then added to the floating point accumulator.

Basically that is the crux of the conversion. When you include the few minor details such as checking for decimals (.), plus signs (+), and minus signs (−), and checking for the exponent *E* and its value, you have a genuine ASCII decimal to floating point conversion. When you do the multiplication and addition in binary, you get the conversion to binary floating point.

Let's give a simple example using decimal notation in the result:

```
532 (decimal) = $35 $33 $32 (ASCII decimal)
1. $35 $33 $32
    ▲ Scanner
   Multiply FAC1 * 10; FAC1 = 0.
   Add $35−$30 to FAC1; FAC1 = 5.
2. $35 $33 $32
        ▲ Scanner
   Multiply FAC1 * 10; FAC1 = 50.
   Add $33−$30 to FAC1; FAC1 = 53.
3. $35 $33 $32
   Scanner ▲
   Multiply FAC1 * 10; FAC1 = 530.
   Add $32−$30 to FAC1; FAC1 = 532, giving the final result of
   the conversion.
```

Now for a more complex example using an exponent and a decimal point in the number to be converted. In this example we also introduce some new variables used by the BASIC conversion routine. 5F has its high-order bit turned on once a period is found. 5D counts the number located to the right of the period. 5E contains the exponent value when the scan is complete. 60 has the sign of the exponent.

```
532.25E−03 (decimal) =
$35 $33 $32 $2E $32 $35 $45 $AB $30 $33 (ASCII decimal)
 1. $35 $33 $32 $2E $32 $35 $45 $AB $30 $33
     ▲ Scanner
    Multiply FAC1 * 10; FAC1 = 0.
    Add $35 − $30 to FAC1; FAC1 = 5.
```

2. **$35 $33 $32 $2E $32 $35 $45 $AB $30 $33**
   ▲ Scanner
   Multiply FAC1 * 10; FAC1 = 50.
   Add $33 − 30 to FAC1; FAC1 = 53.

3. **$35 $33 $32 $2E $32 $35 $45 $AB $30 $33**
   Scanner ▲
   Multiply FAC1 * 10; FAC1 = 530.
   Add $32 − $30 to FAC1; FAC1 = 532.

4. **$35 $33 $32 $2E $32 $35 $45 $AB $30 $33**
   Scanner ▲
   Now that the period is found, set the high-order bit of 5F to indicate this and continue the scan with the next character.

5. **$35 $33 $32 $2E $32 $35 $45 $AB $30 $33**
   Scanner ▲
   Multiply FAC1 * 10; FAC1 = 5230.
   Add $32 − $30 to FAC1; FAC1 = 5322.
   Increment 5D; 5D = 1.

6. **$35 $33 $32 $2E $32 $35 $45 $AB $30 $33**
   Scanner ▲
   Multiply FAC1 * 10; FAC1 = 53220.
   Add $35 − $30 to FAC1; FAC1 = 53225.
   Increment 5D; 5D = 2.

7. **$35 $33 $32 $2E $32 $35 $45 $AB $30 $33**
   Scanner ▲
   Now that the E is found, the part of the number to the left of the E is complete.

8. **$35 $33 $32 $2E $32 $35 $45 $AB $30 $33**
   Scanner ▲
   Now that the token for − is found, set the high-order on in 60 to indicate negative exponent.

9. **$35 $33 $32 $2E $32 $35 $45 $AB $30 $33**
   Scanner ▲
   Add $30 − $30 = 0 to 5E, the exponent; 5E = 0

10. **$35 $33 $32 $2E $32 $35 $45 $AB $30 $33**
    Scanner ▲
    Multiply 5E * 10 by doing ASL, ASL, ADC 5E, ASL; 5E = 0
    Add $33 − 30 = 3 to 5E, the exponent; 5E = 3

11. Scan complete. Test 60, the exponent sign. Sign is negative, so convert 3 to its two's complement form. For convenience we'll just represent this as −3.

12. Subtract 5D, the number of digits past the period, from 5E.
    −3 − 2 = −5; 5E = −5 as final exponent.
13. If 5E is not zero, which it isn't in this case, we need to adjust FAC1 to correspond to an exponent of zero.
    FAC1 = 53225.
    5E = −5.
14. Divide FAC1 by 10 and increment 5E until 5E = 0
    FAC1 = .53225.
    5E = 0.

Notice how this final result compares with the original ASCII decimal value of 532.25E-03, which is the same as .53225E+00.

BASIC uses binary arithmetic for the multiply by 10, divide by 10, and add a digit to FAC1 or to 5E routines. The result in FAC1 is thus in binary format.

BASIC does check for a few more things when converting an ASCII decimal number. If any character is scanned that is not a digit, ., −, +, or E, the scan is stopped and the conversion is considered complete. If a . is found, then a second . also stops the scan. If two digits are scanned for the exponent, then a third digit will cause an OVERFLOW ERROR. However, there is one quirk with this check of the number of digits in the exponent. The check is done only if the sign of the exponent is positive. Thus negative-signed exponents can have more than two digits. For example, neither 23.2E−355 nor 23.2E−000005 produces syntax errors. The latter value when printed shows up as 2.32E.−04. However, the first value is silently set to zero when underflow occurs without giving you any error indication.

ASCII Decimal Number to Binary Floating Point Conversion
B/DCF3-B/DD7D and B/DD91-B/DDB2

Called by:

JSR at A/CC89 INPUT/GET/READ; JSR at A/CE8F Evaluate Expression; JSR at B/D7DA VAL

Operation:

1. Store zero into bytes 5D–67, to correctly initialize before being used in this routine.

2. BCC to step 7 if the first character in the ASCII number is numeric. The carry has been fixed from the last CHRGET that retrieved the first character in this ASCII number.
3. If it's a nonnumeric first character, see if it is −. If not, branch to step 5.
4. For −, store $FF into 67, which will be used in step 37 to see if the floating point number must be converted to its negative format. Branch to step 6.
5. If the character is +, fall through to step 6. If not +, branch to step 8.
6. Scan the second and successive characters in the ASCII string by JSR CHRGET to retrieve the character in the accumulator and leave the carry clear if it's numeric.
7. If it's a numeric character, branch to step 39.
8. For a nonnumeric character, see if the character is . and if true, branch to step 24.
9. See if the character is E. If not, branch to step 26. If it is, then JSR CHRGET to retrieve the next character.
10. If the next character following E is numeric, branch to step 18.
11. See if the next character following E is the token for −, $AB. If so, branch with carry set to step 16.
12. If it is the actual ASCII −, also branch with carry set to step 16.
13. If the next character following E is the token for +, $AA, branch to step 17.
14. Also branch to step 17 if actual ASCII +.
15. If it's a nonnumeric character following E, but neither + nor −, then branch to step 30. For example, this branch would be taken for 23.52E+*2.
16. Branch here with the carry set if it's a − following E. Rotate the carry into the high-order bit of 60, the sign of the exponent.
17. JSR CHRGET to retrieve the next character of the exponent.
18. If this next character is numeric, branch to step 48.
19. For a nonnumeric character, consider the exponent complete.
20. See if 60 indicates a negative sign for the exponent.
21. If no, branch to step 26.
22. For a negative-signed exponent, convert the exponent to two's complement format.

23. JMP to step 27.
24. Branch here if . is found. ROR 5F. Since the carry is set when this branch to step 24 is made, this rotates a 1 into the high-order bit.
25. If bit 6 of 5F is also 1 after the rotate, this is the second . to be found. If true, this is the end of the number, so fall through to step 26. If just the first ., branch to step 6.
26. Come here after all characters in the number have been scanned. Load the accumulator from 5E, the exponent. Of course if no exponent was specified, this value is zero.
27. Subtract 5D, the number of digits following . (up to E if E is present), from 5E, the exponent.
28. If the answer is zero, the number specified was already in correct format. Branch to step 37. For example, 23.5E+1 and 2.35E+2 would not need to be corrected at this point.
29. For a nonzero result, BPL step 34 if result is positive. For example, 23.5E+2.
30. For a negative result (for example 5.32E−3), JSR B/DAFE to divide FAC1 by 10.
31. Increment 5E.
32. If 5E is not zero, branch to step 30.
33. When 5E reaches zero, branch to step 37. With the above example, the final result is .00532E+0.
34. For a positive result, JSR B/DAE2 to multiply FAC1 by 10.
35. Decrement 5E.
36. If 5E is not yet zero, branch to step 34. If 5E is zero, continue with step 37.
37. If 67, the sign of the number, has its high-order bit off, then RTS since conversion is complete.
38. If the high-order bit is on in 67, JMP B/DFB4 to reverse the sign of FAC1 and exit this routine.
39. Branch here from step 7 if a numeric character is retrieved. Push the character onto the stack.
40. Test 5F to see if a . has been found.
41. If no . then branch to step 43.
42. Since . has been found, increment 5D, which counts the number to the right of the period (.).
43. JSR B/DAE2 to multiply FAC1 by 10.
44. Pull this ASCII number from the stack.
45. Subtract $30 to leave the number in the form of $00–$09.

46. Add number to FAC1 by JSR B/DD7E to add the accumulator to FAC1.
47. JMP to step 6 to handle the next character of the number.
48. Branch here to handle exponent numbers. Load the accumulator from 5E, which contains the exponent so far, or zero if there's no exponent so far.
49. Compare the accumulator to 10 (decimal). If it's less than 10, branch to step 53.
50. Test the sign of the exponent, 60.
51. If it's a negative sign, branch to step 58. This allows you to specify a number such as 23.2E−355 or 23.2E−000005. In the first case, you silently get assigned zero when underflow occurs. Print the value of the latter and you get 2.32E−04.
52. If it's a positive sign and 5E is already >=10, this is the third digit being scanned for the exponent, which is not permitted. JMP B/D97E to display OVERFLOW ERROR and return to the Main BASIC Loop.
53. ASL twice. Accumulator = 5E * 4.
54. ADC 5E. Accumulator = 5E * 5.
55. ASL. Accumulator = 5E * 10.
56. Add (7A),0 adding the ASCII value of this character pointed to by the last CHRGET.
57. Subtract $30 to make the character just added in the range of $00–$09.
58. Store the result in 5E, the exponent.
59. Jump to step 17 to handle the next character of the exponent.

Floating Point to ASCII Conversion

The floating point to ASCII conversion routine can be summarized as doing the following:

1. Test for the trivial case of a floating point number of zero.
2. See if the floating point number needs to be displayed in scientific notation (with an E).
3. Keep track of where to store the decimal point and store it into the ASCII string at the appropriate location.
4. If a number such as .01 is to be converted to an ASCII string, make sure the 0 gets placed following the period (.).
5. Convert the floating point number to a series of ASCII characters stored at 0100 and store the . if needed.

6. If scientific notation is being used, store E followed by + or − in the ASCII string.
7. Convert the floating point exponent to a two-character ASCII string and store following the sign.
8. Exit with the accumulator and Y-register pointing to the start of the string at 0100.

This routine can also be used to compute a value for TI$. TI$ just uses a different table for conversion of the floating point number to a series of ASCII characters.

If the number in FAC1 is < .01 or > 999,999,999, the ASCII string is stored in scientific format.

The end of the string at 0100 is signified by a byte = $00.

This routine has one aspect that could sometimes cause you problems. It uses the bottom of the stack from 100 to store the ASCII string created. The longest string created can be nine ASCII decimal characters, two signs, E, and two exponent numbers for a total of 14 characters, thus potentially using the stack space from 0100–010E. If your program just happens to be using this stack space, which is unlikely but could happen, your program might jump to some unexpected places.

If you treat this routine as a black box, you can feed it a floating point number in FAC1, and it returns an ASCII string at 0100 with the accumulator and Y-register pointing to this string.

The above is a very brief summary of the routine. The following descriptor of the routine is also quite summarized since it would be easy to get lost in a very detailed description of the routine. A detailed example of the conversion of a floating point number to a series of ASCII characters is included, and this example uses base 10 notation since that might be easier to follow.

Convert Floating Point Accumulator One to ASCII String
B/DDDD-B/DF10

Called by:
JSR at A/CABC PRINT; Alternate $B/DDDF: JSR at B/DDD7 Decimal Output; JSR at B/D46A STR$; Alternate B/DE68 TI$.

Operation:

1. If the floating point accumulator has a negative sign, store

$2D, ASCII − as the first character of the ASCII string.
2. If FAC1 is zero, store a $30, ASCII zero, as the first byte of the string, followed by the $00 end of string.
3. If the exponent of FAC1 is > $80 (decimal 0 in excess-128 form), set the base 10 exponent, 5D, to $00, setting for base ten numbers that are >= 1.0, and continue with step 4.
4. For exponents <= $80, set the base 10 exponent to −9 and multiply FAC1 by 1,000,000,000.
5. Now see if FAC1 is in the range of 99,999,999.9 <= FAC1 <= 999,999,999.

   If FAC1 is >999,999,999, divide FAC1 by 10 and increment the base ten exponent, 5D, and repeat this until finally FAC1 is <999,999,999.

   If FAC1 <99,999,999.9, multiply FAC1 by 10 and decrement the base 10 exponent, 5D, and repeat this until finally FAC1 is >99,999,999.9.

   The purpose of this step is to get FAC1 into the range that can be displayed using normal notation. If by getting it into this range the base 10 exponents are too large or too small, then the number is displayed in scientific format.
6. JSR B/DC9B to convert FAC1 to a four-byte signed integer in FAC1.
7. Now add 10 to the base 10 exponent, which was initially set to −9 if <1.0 or to 0 if >= 1.0. If the resulting exponent is less than zero or greater than ten, the number is to be displayed in scientific format.

   Here is an example calling the routines to do the conversion in step 4 and step 5.

.0078125 = 7A 80 00 00 in FAC1.
Original Base 10 is −9, or $F6.
Multiply by 1,000,000,000.
Result = 97 EE 6B 28 00 in FAC1.

Since this FAC1 value is less than 99,999,999.9, which is 9B 3E BC 1F FD, FAC1 is multiplied by 10 and the base 10 exponent is decremented so now 5E = $F5.

Result = 9E BA 43 B7 40 in FAC1.

Since this result in FAC1 is now within the range of 99,999,999.9 to 999,999,999, we are done with its adjustment.

Now add $0A (10) to the base 10 exponent, $F5. $F5 + $0A = $FF. Since this value is less than 0, the number is to be displayed in scientific format. If you try to PRINT .0078125, the display is 7.8125E−03.

8. Now set the decimal point location and the base 10 exponent based on step 7. If scientific notation is to be used, the decimal point location is 1, since all scientific numbers have one digit preceding the decimal point.

   If normal notation is to be used, take the base ten exponent minus 1.
9. If the decimal point position is 0, then store a . as the first character of the ASCII string. If standard notation is being used and the exponent computed in step 6 was 0, the decimal point location = −1, and in this case store $30, ASCII zero, following the decimal point. Thus, .0625 would result in a . followed by a 0. Since values less than .01 are displayed in scientific notation, this one zero is all that we need be concerned with. Other zeros that may appear following the decimal point—for example, 5.7203—are stored there during the conversion of FAC1 to ASCII characters.
10. B/DE68: Alternate entry point for TI$ with the Y-register = $24 to point to the table conversion values for 10 hrs, 1 hr, 10 min, 1 min, 10 sec, 1 sec.

    Once the decimal point and the base 10 exponent are taken care of, convert FAC1 to a series of ASCII characters. Although the code for this is quite complex, what effectively happens is that FAC1 is divided by a series of numbers, with each following division using as a dividend the remainder from the previous division. These numbers that are divided into FAC1 are −100,000,000; +10,000,000; −1,000,000; +100,000; −10,000; +1,000; −100; +10; −1. Although performed in binary in the computer, the principle can be illustrated using base 10 as the following example shows. Ignore the decimal point in this example.

Convert FAC1 of 7305 to ASCII string.

```
                          7 = $37 = "7"
              ____________________________
100,000,000 | 7305.00000
              7000 00000
              ----------
```

```
                      3 = $33 = "3"
             ___________________________
 10,000,000 | 30500000
              30000000
              --------

                      0 = $30 = "0"
             ________________________
  1,000,000 | 500000
                   0
              ------

                      5 = $35 = "5"
             ________________________
    100,000 | 500000
              500000
              ------
```

Thus the ASCII string produced in this example is "7305".

After each digit is stored in the buffer at 0100, the decimal pointer counter is decremented, and when it reaches zero a decimal point is stored in the ASCII string.

The reason for the alternating positive and negative numbers in the tables is due to the way this conversion algorithm works. A long example showing how this works (using base 10 for illustration purposes) follows at the end of this detailed routine.

Also note that since nine divisions take place, the maximum number of ASCII digits that can appear in the resulting string is nine.

11. After all the numbers in FAC1 have been converted to an ASCII string and stored in the buffer at 0100, the last number stored is retrieved.

    If this last character is $30, ASCII zero, the pointer is backed up until a nonzero is located. If this nonzero is a period, the pointer is moved back past it also. Thus all zeros past the last significant digit of the number are dropped. If there are no significant digits to the right of the decimal point, the decimal is also dropped. Thus 5.0500 is stored as 5.05 and 5.00 is stored as 5, with these numbers followed either by the $00 end-of-string byte if in normal notation or by E if in scientific notation.
12. Now handle the exponent. If the base 10 exponent is 0, no exponent is needed, so just end the string by storing $00. If the exponent is needed, first store an E. Then determine the sign of the exponent. If the exponent is less than zero, store a − sign. If the exponent is greater than zero, store a + sign. Convert the exponent to ASCII format by seeing

how many times you can divide the exponent by 10 through repeated subtraction. This number is the first digit of the exponent, and the remainder is the second digit of the exponent. Store a $00 end-of-string byte following the second digit of the exponent.

13. Set the accumulator = $00 and the Y-register = $01 to point to this ASCII string at 0100 and exit.

Example of FAC1 Conversion to ASCII Characters by Repeated Additions of Table Values

This example uses base 10 notation since it might be easier to follow and the principle is the same as if using binary. The only thing you need to be aware of is that, just like two's complement representation of negative numbers, there can also be nine's complement representation of negative base 10 numbers. The nine's complement is computed by subtracting the number from 9's and adding 1.

Let's go ahead and convert the negative table entries to nine's complement:

```
-100,000,000    =     999,999,999
                     -100,000,000
                     ------------
                      899,999,999
                               +1
                     ------------
                      900,000,000

-1,000,000      =     999,999,999
                     -  1,000,000
                     ------------
                      998,999,999
                               +1
                     ------------
                      999,000,000

-10,000         =     999,999,999
                     -     10,000
                     ------------
                      999,989,999
                               +1
                     ------------
                      999,990,000

-100            =     999,999,999
                     -        100
                     ------------
                      999,999,899
                               +1
                     ------------
                      999,999,900
```

```
−1                 =     999,999,999
                                  −1
                        ────────────
                         999,999,998
                                  +1
                        ────────────
                         999,999,999
```

Each successive number from the table is added to the original value in FAC1. For table entries that are negative, the addition continues until the carry is clear. With each addition, increment a count, so that when the carry is clear, this count contains the number of times this value fit into the number + 1. Subtract 1 from this value, convert it to ASCII format by preceding it with a $3 and store in the output buffer as this ASCII character.

For table entries that are positive, add the value until the carry is set. Again keep a count of the number of times addition is made before the carry is set. However, when you're finished, Exclusive OR this value with $FF and add $0A.

Actually, this just converts the number to two's complement, so in the calculation below, just subtract this value from 10. Again convert this digit to its ASCII format and store it in the output buffer.

Now for the actual example:

Given the number 572,570,000.

1. Add −100,000,000 until the carry is clear.

```
                    Counter
    572,570,000
_  +900,000,000
   ────────────
1   472,570,000        1

_  +900,000,000
   ────────────
1   372,570,000        2

_  +900,000,000
   ────────────
1   272,570,000        3

_  +900,000,000
   ────────────
1   172,570,000        4

_  +900,000,000
   ────────────
1   072,570,000        5
```

```
_   +900,000,000
0    972,570,000      6
```

Carry is clear, so 6−1 =5 so store $35 into the buffer.

2. Add 10,000,000 until carry is set.

```
     972,570,000
_   + 10,000,000
0    982,570,000      1

_   + 10,000,000
0    992,570,000      2

_   + 10,000,000
1    002,570,000      3
```

Carry is set, so 10 − 3 = 7, so store $37 into the buffer.

3. Add −1,000,000 until carry is clear.

```
     002,570,000
_   +999,000,000
1    001,570,000      1

_   +999,000,000
1    000,570,000      2

_   +999,000,000
0    999,570,000      3
```

Carry is clear, so 3−1 = 2 and store $32 into the ASCII string.

This should be enough to give you a feel for what is happening. If you wish, you can complete this conversion as an exercise. Remember that while the examples are in base 10, the carry is set or cleared in exactly the same fashion when adding the binary values of these table entries.

Floating Point to Fixed Point Integer Conversion

The floating point to fixed point integer conversion routine converts floating point numbers whose exponents in FAC1 are in the range $81–$9F and whose mantissas are in normalized format. After $81-$9F is converted to decimal and the excess-128 notation is removed, the valid exponents are 1–31. These apply to normalized numbers such as .1E+01 or .9E+31. If these numbers are represented with one digit to the left of the decimal point, the range of floating point numbers that can be converted to fixed point integer becomes 1.0E+00–9.0E+30. If you have a mantissa part such as 1.235E+00 or 9.999E+30, the mantissa parts of the number are dropped in the final fixed point integer representation in FAC1 since these mantissas have been rotated out of the low-order bit of the FAC1 mantissa.

When the values with exponents are greater than +30 (considering one digit to the left of the decimal point) or less than +00 (considering one digit to the left of the decimal point), then these floating point values are too great to be represented or are a mantissa that should not be represented as an integer. In either case, the FAC1 mantissa is left as all zeros.

Another feature of this routine is the way it treats values that have a negative sign in 66. The values in FAC1 are complemented or reversed (not two's complement), and the sign bit of 1 is shifted into the number from high-order to low-order. See the example below for the exponent of $99 with a negative sign. One of the features of propagating this negative sign from high-order to low-order during the shift is that you can specify an exponent such as $A0 or $FF with a negative sign in 66. When the routine finishes, you will have propagated ones throughout the FAC1 mantissa, which now appears in hex as FF FF FF FF. If you used $FF as your exponent when doing this and converted the sign in 66 back to $00 when done, you would have produced the highest possible floating point value.

This routine works by determining how many bits need to be shifted right to make the floating point number into a fixed point integer.

Here are some examples of before and after conversion:

Before:					**After:**			
Exponent	**62**	**63**	**64**	**65**	**62**	**63**	**64**	**65**
9F	FE	DC	BA	98	7F	6E	5D	4C
99	FE	DC	BA	98	01	FD	B9	75
(with negative sign):								
99	FE	DC	BA	98	FE	02	46	8A
98	FE	DC	BA	98	00	FE	DC	BA

90	FE	DC	BA	98	00	00	FE	DC
8F	FE	DC	BA	98	00	00	00	FE
82	FE	DC	BA	98	00	00	00	03
81	FE	DC	BA	98	00	00	00	01
80	FE	DC	BA	98	00	00	00	00
7F	FE	DC	BA	98	00	00	00	00
A0	FE	DC	BA	98	00	00	00	00
FF	FE	DC	BA	98	00	00	00	00

Float-to-Fix Conversion B/DC9B–B/DCCB

Called by:
JSR at A/CA11 LET; JSR at B/D1CE Convert Integer Float-to-Fix; JSR at B/D801 Convert Address Float-to-Fix; JSR at B/DCD2 INT; JSR at B/DE32 Convert Float to ASCII.

Operation:

1. Load the accumulator from 61, the exponent of FAC1.
2. If 61 is zero, branch to B/DCE9 to also store $00 into 62, 63, 64, and 65 and then RTS.
3. Subtract $A0 from the exponent, but since the exponent is in excess-128 notation, actually subtracting 32. By subtracting 32, 32 fewer right shifts are done which prevents the four-byte FAC1 mantissa from being rotated out so that the binary point will be at the right of 65 rather than at the left of 62.
4. If FAC1 is positive, branch to step 6.
5. For a negative FAC1, JSR B/D94B to convert the mantissa of FAC1 to two's complement form.
6. If the exponent − 32 leaves less than eight bits to be shifted, branch to step 8.
7. For eight or more bits to be shifted, JSR B/D999 to shift the mantissa of FAC1 right one byte at a time and any final partial byte.
8. For eight or fewer bits to be shifted, JSR B/D9B0 to shift less than eight bits right.

Specialized Floating Point to Fixed Point Integer Conversions

The following routines are all specialized routines for conversion of floating point values to fixed point integer values.

The routines check for values being in certain ranges.

Some also check for the type of the variable or expression being converted. The routines include:

- Convert floating point to fixed point and display ILLEGAL QUANTITY if not in the range of −32768 to 32767.
- Convert floating point to integer in the range of −32768 to 32767 only if the last variable or expression was numeric.
- Convert expression to floating point format and then convert floating point to fixed point in the range of −32768 to 32767 if numeric expression.
- Convert floating point to fixed point in the range of −32768 to 32767 and return the resulting two-byte integer with the accumulator containing the high-order and the Y-register containing the low-order.
- Convert floating point to fixed point integer in the range of 0–255.
- Convert expression to floating point number and convert floating point number to fixed point integer in the range of 0–255.

Convert Floating Point Numbers in Range −32768 to 32767 to Fixed Point Integer B/D1BF-B/D1D0

Called by:
Fall Through from Convert Float-to-Fix Only If Numeric Variable; JSR at A/C9C7 LET for Integer Variable; JSR at A/CED4 NOT: JSR at A/CFED, A/CFFF AND/OR; JSR at B/D1AA Float-to-Fix Convert Returning Integer in accumulator and Y-register.

Operation:

1. See if the exponent for FAC1, 61, is < $90. If so, branch to step 3 as this FAC1 floating point value is <=32767.
2. If the exponent is $90, see if this floating point value is −32768. Note that +32768 is out of the range of integers. Perform the check by JSR B/DC9B to compare FAC1 to the maximum integer value at B/D1A5. If the exponent is greater than $90, FAC1 holds a value greater than the largest possible integer. Display ILLEGAL QUANTITY and return to the Main BASIC Loop.
3. JMP B/D8D2 to convert FAC1 to a four-byte signed integer in the mantissa of FAC1. Only bytes 64 and 65 will contain

significant digits since the maximum size was limited in the steps above to 32768. If the integer is negative, the mantissa in FAC1 is in two's complement.

Float-to-Fix Conversion Only If Numeric Variable Type
B/D1B8-B/D1BE

Called by:
JSR at B/D7A1 Float-to-Fix for Integer 0–255; Fall Through from Convert Expression to Integer.

Operation:
1. JSR A/CD8D to see if the current variable is numeric. If not, display TYPE MISMATCH error and return to the Main BASIC Loop.
2. If FAC1 is negative, display ILLEGAL QUANTITY error message and return to the Main BASIC Loop.
3. Fall through to B/D1BF to convert FAC1 to a fixed point integer.

Convert Numeric Expression from Floating Point to Fixed Point Integer
B/D1B2-B/D1B7

Called by:
JSR at B/D1E3 Convert Array Dimension Element for Create or Locate Array.

Operation:
1. JSR CHRGET to retrieve the first character in the expression.
2. JSR A/CD9E to evaluate this expression to a value in FAC1.
3. Fall through to B/D1B8 to check that this expression was numeric, then convert FAC1 to a fixed point integer.

Convert Floating Point Numbers to Fixed Point Integer—Return Low-order Byte in Y-Register and High-order Byte in Accumulator
B/D1AA-B/D1B1

Called by:
None. This routine does not appear to be used by anything else in BASIC or the Kernal.

Operation:

1. JSR B/D1BF to convert FAC1 to a fixed point integer. Since B/D1BF converts only values in the integer range, the final two-byte integer is left in 64 and 65.
2. Load the accumulator from 64, the high-order byte of the integer.
3. Load the Y-register from 65, the low-order byte of the integer.

Convert Floating Point to Integer in the Range 0–255 B/D7A1-B/D7AC

Called by:

Fall Through from Convert Expression to Fixed Point 0–255; JSR at B/D6EC CHR$.

Operation:

1. JSR B/D1B8 to see if the current variable or expression is numeric. If not, display TYPE MISMATCH. If it's a negative number or if it's out of range of the integers, display ILLEGAL QUANTITY. Convert FAC1 to a fixed point signed integer in 64 and 65.
2. If 64 is nonzero, this value is greater than 255. Branch to B/D798 to display ILLEGAL QUANTITY.
3. Load the X-register from 65, getting the value of 0–255.
4. JMP CHRGOT to retrieve the last character scanned by CHRGET and exit with this character in the accumulator.

Evaluate Expression and Convert to Fixed Point Integer in Range 0–255 B/D79E-B/D7A0

Called by:

JSR at A/C94B ON; JSR at A/CA86 CMD; JSR at A/CB85 GET#; JSR at A/CBA5 INPUT#; JSR at A/CFC7 FN; JSR at B/D7F4 POKE/WAIT; JSR at E200/E203 Evaluate Expression Following Comma; JSR at E221/E21E OPEN/CLOSE.

Operation:

1. JSR A/CD8A to evaluate expression. The text pointer (7A) is pointing to the first character of the expression when step 1 receives control.

2. Fall through to B/D7A1 to type check for a numeric expression and then to convert FAC1 to an integer in the range of 0–255.

Integer to Floating Point Conversion

The following routines convert one- or two-byte integers to floating point format.

To convert a two-byte integer to floating point, the two-byte integer is stored into the two high-order bytes of FAC1 mantissa, 62 and 63, and the exponent for FAC1 is set to $90. $90 = 16 (decimal) after the excess-128 notation is removed. This exponent is thus saying that this number in FAC1 to be converted back to an exponent of zero would have to have the binary point located at the right of 63.

To convert a one-byte integer to floating point, the one-byte integer is stored in the high-order byte of FAC1, 62, zero is stored in 63, and the exponent for FAC1 is set to $88. $88 = 8 (decimal) after the excess-128 notation is removed. This exponent is thus saying that this number in FAC1 to be converted back to an exponent of zero would have to have the binary point located at the right of 62.

Next, 64 and 65 are stored with zeros. Then the normalization routine is called to convert the integer to its final floating point format. For example, if a one-byte integer, 5, was being converted to floating point, then 62 = 0000 0101 and 61=$88 before the normalization. After normalizing 62 = 1010 0000 and 61 = $83.

Several different specialties of integer to floating point conversion are provided.

One version assumes that 62 and 63 already contain the integer and that the exponent 61 has already been set.

A second version converts the one-byte accumulator to floating point.

A third version converts the accumulator (high-order) and Y-register (low-order) to floating point format.

A fourth version converts the accumulator (high-order) and X-register (low-order) to floating point format and then converts the floating point format to an ASCII string and actually sends each character in the ASCII string to the current output device.

Convert Two-Byte Integer in 62 and 63 of FAC1 to Floating Point B/DC44-B/DC57

Called by:
Fall Through from Convert Accumulator to Floating Point; JMP at B/D39B Convert Accumulator and Y-register to Float; Alternate B/DC49: JSR at B/DDD4 Decimal Output; Alternate B/DC4F: JMP at A/CF81 Locate TI for Expression Evaluation.

Operation:
1. Flip the value of the sign bit (high-order bit of 62) and rotate into carry flag. If the sign is negative (1), carry = 0; if the sign is positive (0), carry = 1.
2. B/DC49: Store $00 into 64 and 65 since only doing a two-byte conversion.
3. B/DC4F: 61 = X-register which contains the exponent for the two-byte number.
4. Store the accumulator of $00 into 70, the rounding byte, and 66, the sign.
5. JMP B/D8D2 to normalize FAC1. If the carry is equal to one convert FAC1 to two's complement form. If 62 is zero, clear all bytes of FAC1 mantissa, shift 70 into 62, and leave 61 = zero. If 62 is not zero, normalize FAC1 by shifting the mantissa left one bit and adding one to the exponent until the high-order bit of the mantissa (the high-order bit of 62) is one, thus indicating the mantissa is normalized.

Convert Accumulator to Floating Point B/DC3C-B/DC43

Called by:
Fall Through from SGN; JSR at B/DD83 Add Accumulator to FAC1; JSR at B/D07B Comparison; JSR at A/CF9D Locate ST for Expression Evaluation.

Operation:
1. Store the accumulator in 62, the high-order byte of the FAC1 mantissa.
2. Store zero into 63, the second byte of the FAC1 mantissa.
3. Set the X-register = $88 to indicate the binary point would be shifted eight places from the left of 62 to the right of 62

to make the accumulator have an exponent of 0 again (which is its normal format).
4. Fall through to B/DC44 to convert the two-byte integer in 62 and 63 to floating point format.

Convert Accumulator and Y-Register to Floating Point B/D391-B/D39D

Called by:
JMP at A/CEE0 NOT; JMP at A/CF6B Locate Integer Variable for Expression Evaluation; JMP at B/D013 AND/OR; BEQ at B/D3A4 POS.

Operation:
1. Store zero into 0D to indicate this is a numeric variable.
2. Store the accumulator in 62, the high-order byte of the FAC1 mantissa. Thus the accumulator contains the high-order byte of the integer.
3. Store the Y-register in 63, the second byte of the FAC1 mantissa. The Y-register contains the low-order byte of the integer.
4. Set the X-register = $90 to indicate that you would need to shift the binary point 16 places to the right, leaving it to the right of 63.
5. JMP B/DC44 to convert the two-byte integer in 62 and 63 to floating point format.

Convert Accumulator and X-Register to Floating Point, Then to ASCII String, Then Send to Output Device B/DDCD-B/DDDC

Called by:
Fall Through from Display IN Message; JSR at E43A/E41C Display Number of BYTES FREE; JSR at A/C6EA LIST; Alternate B/DDDA JMP at B/DDC6 Display IN Message.

Operation:
1. Store the accumulator in 62, the high-order byte of the FAC1 mantissa.
2. Store the X-register in 63, the second byte of the FAC1 mantissa.

   As an example of step 1 and step 2, consider 259 (decimal) = $0103. Accumulator = $01 and X-register = $03.

3. Set the X-register to $90 to correctly set the binary point as if it were at the right of 63.
4. SEC to prevent converting FAC1 to two's complement.
5. JSR B/DC49 to convert the two-byte integer in 62 and 63 to floating point format in FAC1.
6. JSR B/DDDF to convert FAC1 to an ASCII string at 0100 and exit with the accumulator and Y-register pointing to 0100.
7. B/DDDA: JMP A/CB1E to send each character in the string pointed to by the accumulator and Y-register to the current output device until the $00 end-of-string byte is found.

Convert Y-Register to Floating Point B/D3A2-B/D3A5

Called by:
Fall Through from POS; JMP at B/D77F LEN; JMP at B/D821 PEEK; JMP at B/D795 ASC.

Operation:
1. Load the accumulator with zero.
2. Branch to B/D391 to convert the accumulator and Y-register to floating point value in FAC1.

Miscellaneous Floating Point Routines

The various other floating point routines that did not neatly fit into any of the previous categories are described here. These routines include:

- INT.
- Convert FAC1 to two's complement form.
- Increment the mantissa of FAC1.
- Set the accumulator based on the sign of FAC1.
- SGN.
- ABS.
- Add the accumulator to FAC1.
- Store the accumulator into all mantissa bytes of FAC1.
- Compare FAC1 to the Floating Point Number in Memory.
- Display OVERFLOW ERROR.

Several other routines such as the trig functions make use of floating point routines, but these other routines are discussed in their own separate sections.

INT
B/DCCC-B/DCE8

Called by:
Invoke Function for INT; JSR at B/DF8F Exponentiation; JRS at E00E/E00B EXP; JSR at E27A/E277.

The INT for positive numbers returns the integer portion of the number. For negative numbers INT returns the next lower negative integer. For example, INT(83.2) returns 83 and INT(−83.2) returns −84. However, INT also functions with arguments that are outside the range of integers; for example, INT(65205.73) returns 65205. This feature might seem unexpected since INT sounds like it returns the integer portion of a number. Well, it does return the integer portion, but only in the internal floating point format of FAC1, and FAC1 is not converted to integer format. Thus X = INT(65205.73) is valid, but X% = INT(63205.73) produces ILLEGAL QUANTITY since the FAC1 value that results from the INT is outside the range of integers.

Operation:

1. If 61, the exponent of FAC1, >=$A0 or 32 in decimal after excess-128 is removed, the value in FAC1 is too large to have any fractional part so just exit with the accumulator = the exponent and FAC1 unchanged.
2. For exponents <32, JSR B/DC9B to convert FAC1 to a four-byte signed integer in FAC1.
3. Store the zero into 70, the rounding byte.
4. Load the accumulator with 66, the sign of FAC1.
5. Store the zero into 66.
6. Exclusive OR the accumulator, which holds the sign loaded in step 4, with $80, thus setting the accumulator high-order bit to 0 if it's a negative sign or 1 if it's a positive sign.
7. Rotate the high-order bit left into the carry.
8. Set 61, the exponent for FAC1, to 32, to indicate the binary point goes to the right of 65.
9. Store 65, the low-order byte of the FAC1 mantissa, into 07, which now indicates whether this number is even or odd. It is odd if the low-order bit is 1.
10. JMP B/D8D2 to convert FAC1 to two's complement if the carry is clear indicating negative sign, store zero into the

remaining bytes of the FAC1 mantissa if 62 is zero, and normalize FAC1 by shifting to left until the high-order bit of 62 becomes one and adding one to the exponent with each shift.

Convert FAC1 to Two's Complement Form B/D947-B/D96E

Called by:
JSR at B/D8D4 Complement FAC1 If Carry Clear and Normalize; Alternate B/D94B: JSR at B/DCAB Convert Float-to-Fix.

Operation:

1. Reverse 66, the sign byte, by doing an Exclusive OR with $FF.
2. B/D94B: Reverse the mantissa bytes of FAC1—62, 63, 64, and 65 by doing an Exclusive OR with $FF.
3. Reverse 70, the rounding byte, by doing an Exclusive OR with $FF.
4. Increment 70.
5. If 70 is not zero, RTS.
6. If 70 is zero, fall through to B/D96F to increment the mantissa, thus converting FAC1 to two's complement form. Thus, for this routine to produce the two's complement, 70 must be zero at entry to the routine. If 70 is not zero, you end up with just the one's complement when you exit in step 5. If this one's complement conversion is what you want, just set 70 to nonzero and call this routine. It seems that it would be wise in this routine to increment the mantissa no matter what the value of 70 if two's complement conversion is expected.

Increment Mantissa of FAC1 B/D96F-B/D97D

Called by:
Fall Through from Convert FAC1 to Two's Complement; JSR at B/DC23 ROUND.

Operation:

1. Increment the FAC1 mantissa by incrementing the low-order byte 65. If this increment results in 65 becoming zero, then

increment 64. If 64 was incremented, see if it is now zero. If 64 is now zero, then increment 63. Now test if 63 is zero, and if so, increment 62. Finally, if 62 is incremented to zero, just exit with Z=1 (BEQ condition).
2. RTS.

Set Accumulator Based on Sign of Floating Point Accumulator One
B/DC2B-B/DC38

Called by:
JSR at A/C79F FOR—Get Step Sign; JSR at B/D9EA LOG; JSR at B/DC39 SGN; BEQ at B/DC65 Compare FAC1 to Memory; JSR at E097/E094 RND; Alternate B/DC2F: BMI B/DC6B Compare FAC1 to Memory; Alternate B/DC31 JMP at B/DC98 Compare FAC1 to Memory.

Operation:
1. If 61, the exponent of FAC1, is zero, exit with the accumulator equal to zero.
2. B/DC2F: Load the accumulator from 66, the sign for FAC1.
3. B/DC31: ROL, rotating the high-order bit of the accumulator into the carry.
4. Set the accumulator to $FF to set the default of the negative sign until told different.
5. If the carry is set, the sign is negative, so exit with the accumulator = $FF.
6. If the carry is clear, this is a positive signed FAC1. Set the accumulator to $01 and exit.

SGN
B/DC39-B/DC3B

Operation: Invoke Function for SGN.
1. At entry the argument for SGN has been stored in FAC1.
2. JSR B/DC2B to determine the sign of FAC1. If it's positive, return with accumulator = $01; if it's negative, return with the accumulator = $FF; and if it's zero, return with the accumulator = $00.
3. Fall through to B/DC3C to convert the accumulator to a floating point number in FAC1.

ABS
B/DC58-B/DC5A

Operation: Invoke Function for ABS.

1. At entry the argument for ABS has been stored in FAC1.
2. LSR 66, the sign for FAC1, thus shifting a zero into the high-order bit, signifying a positive number.

Add Accumulator to Floating Point Accumulator One
B/DD7E-B/DD90

Called by:
JMP at A/CA29 Add for TI$; JSR at B/DA21 LOG; JSR at B/DD78 ASCII-to-Float Conversion.

Operation:

1. Push the accumulator, which contains a value from $00–$09 to add to FAC1, onto the stack.
2. JSR B/DC0C to copy FAC1 to FAC2 with rounding.
3. Pull the number to be added from the stack.
4. JSR B/DC3C to convert the integer in the accumulator to a floating point format in FAC1 and normalize the value in FAC1.
5. Exclusive OR the sign of FAC1 and FAC2 and store the result in 6F, the sign comparison byte.
6. Load the X-register from 61, the exponent for FAC1.
7. JMP B/D86A to add FAC2 to FAC1.

Store Accumulator into All Bytes of FAC1 Mantissa
B/DCE9-B/DCF2

Called by:
BEQ at B/DC9D—Float-to-Fix Conversion for Zero Value FAC1 Exponent.

Operation:

1. This routine is branched to with the accumulator = zero.
2. Store the accumulator into 62, 63, 64, and 65.
3. Transfer the accumulator to the Y-register.
4. RTS.

Compare FAC1 to Value in Memory Location B/DC5B–B/DC9A

Called by:
JSR at B/D027 Compare Numerics or Strings; JSR at B/D1C9 Float-to-Fixed Point Integer Conversion; JSR at B/DE0F, B/D1A Convert Floating Point to ASCII; JSR at B/DF96 Exponentiation; Alternate B/DC5D: JSR at A/CD57 NEXT.

Exit Conditions:
Accumulator = $00 if the value in memory = FAC1.
Accumulator = $01 if the value in memory < FAC1.
Accumulator = $FF if the value in memory > FAC1.

Operation:

1. Store the accumulator in 24.
2. B/DC5D: Store the Y-register in 25.
   (24) now points to the floating point number in memory to be compared to FAC1.
3. Set the Y-register = $00.
4. Load the accumulator from (24),Y to load the exponent of the floating point number in memory.
5. Increment the Y-register.
6. Transfer the accumulator to the X-register.
7. If the accumulator = zero, branch to B/DC2B to exit with the accumulator = 0 if FAC1 is 0, accumulator = 1 if FAC1 is positive, or accumulator = $FF if FAC1 is negative.
8. Load the accumulator from (24),Y to load the first byte of the mantissa of the floating point number in memory. This floating point number in memory should have been stored with this byte having its high-order bit = 1 if this is a negative number.
9. Exclusive OR the accumulator, which contains the sign of the floating point number in memory, with 66, the sign of FAC1.
10. If the signs are different, branch to B/DC2F to exit with the accumulator = 1 if FAC1 is positive or = $FF if FAC1 is negative.
11. Compare the X-register, which holds the exponent of the number in memory, to 61, the exponent of FAC1.
12. If the exponents are not equal, branch to step 21.

13. For equal exponents, compare the bytes of the respective mantissas.

    Compare the first byte of the mantissa of the number in memory to 62. Before the comparison, the byte from memory is Exclusive OR'd with $80 to restore it to normalized format.
14. If these two bytes are not equal, branch to step 21.
15. Compare the second bytes of the mantissas and branch to step 21 if not equal.
16. Compare the third bytes of the mantissas and branch to step 21 if not equal.
17. If 70, the rounding byte of FAC1, is >= $80, then set the carry; otherwise, clear the carry. This affects the SBC in step 19 by in effect rounding 65 up if 70 is >= $80.
18. Load the accumulator with the fourth mantissa byte of the floating point number in memory.
19. SBC 65, the fourth mantissa byte of FAC1.
20. If the accumulator = $00, indicating no difference in the low-order fourth bytes, BEQ to B/DCBA to RTS with the accumulator = $00, indicating that the two values are equal.
21. If either the exponent or any mantissa bytes are not equal, load the accumulator from 66, the sign of FAC1, which must be positive ($00) if this step is reached.
22. If the carry is clear, branch to step 24. This branch is taken if the byte of the number in memory is less than the byte of FAC1.
23. For the byte of the number in memory being greater than the FAC1 byte, Exclusive OR the accumulator with $FF, thus reversing the sign from $00 to an $FF value in the accumulator.
24. JMP B/DC31: Rotate the high-order bit of the accumulator left into the carry. If the carry is set, exit with the accumulator = $FF (floating point number in memory > FAC1). If the carry is clear, exit with the accumulator = $01 (floating point number in memory < FAC1).

Display OVERFLOW ERROR B/D97E-B/D982

Called by:
BEQ at B/D93A; JMP at B/DADF; JMP at B/DD9D.

Operation:

1. Set the X-register to $0F to index to OVERFLOW ERROR message.
2. JMP A/C437 to display this error message and return to the Main BASIC Loop.

No error message is issued for the underflow condition.

String Operations

A string is a sequence of characters enclosed within quotation marks. If nothing else follows the string, the final quotation mark is not needed. The characters between the quotation marks can be any of the ASCII characters with the exception of a few control keys such as RETURN, RUN/STOP, and INSERT.

A string expression can be a simple constant such as XYZ, a string variable such as A$, a string function such as LEFT$("XY",1), or any combination of these. String expressions are valid operands for the PRINT and PRINT# commands. String expressions can also be used as operands in other commands such as OPEN, CLOSE, LOAD, and SAVE. You can also INPUT or READ string expressions. Comparison of string expressions is also valid. The only arithmetic operator that is valid on string expressions is + for string concatenation.

Functions which take string expressions as arguments and return string expressions after evaluation include LEFT$, RIGHT$, and MID$. CHR$ takes a numeric expression in the range 0–255 and converts it to a string expression. STR$ takes a numeric expression and converts it to a string expression. ASC, VAL, and LEN take string expressions and convert them to arithmetic expressions.

All of these functions must appear on the right side of the equal sign during statement execution (unless they are an operand of one of the commands that can have string operands), and none can serve as a variable.

One string variable function is defined, TI$. TI$ can appear on either side of the equal sign.

You can develop unusual applications with strings, such as using strings to add numbers that are longer than nine digits by aligning the two strings and then adding the respective digits from each string.

The section on variables showed how strings are referenced by string descriptors that can be in either scalar string variables or array string variables. Each descriptor contains the length of the string and a pointer to the string. By keeping string descriptors this fixed size, the scan for variables will always know how many bytes it has to add to locate the next descriptor.

The string itself is not actually included in the string descriptor. Rather, the string descriptor points to strings which are located in either the tokenized program area or in an area at the top of BASIC memory referred to as the active string area. Any string within quotes (or following a single quote) that is entered in a line of program text is stored unchanged in the program text area when the line is tokenized and stored.

The active string area is used to hold strings that are the result of concatenation of string expressions, LEFT$, MID$, RIGHT$, or CHR$. During statement execution, whenever a string variable is referred to, the descriptor for that string is retrieved and from this descriptor the pointer to the string and the length of the string are known.

During LET statement execution when a new string value is assigned to a string variable, the descriptor is created containing the length of this newly referenced string and a pointer to its location.

Another area in page 0 also comes into play during string operations. A temporary string descriptor stack at 19 through 21 (hex) contains space for three string descriptors of three bytes each. A pointer to the next available slot to be used in this temporary stack is maintained in 16 and a pointer to the last entry in this stack that was active is kept in 17. A one-byte pointer is sufficient for pointing to a page 0 location. Whenever string expressions are being evaluated or during any of the string functions, this temporary string descriptor stack maintains pointers to the strings in the expression.

For example, if you had a string expression with two concatenations within it, the temporary string descriptor stack would, at different times during the evaluation of this expression, contain a descriptor for each of the three strings being concatenated. When the concatenation was complete, the final temporary string descriptor would be removed and assigned to the variable that was on the left side of the equal sign.

The temporary string descriptor stack is purged of the last entry after each string function or operation (such as PRINT for a string) completes. It will become full if expressions become too complex and the error message FORMULA TOO COMPLEX is displayed.

Another topic that deals with strings is termed garbage collection. This operation is invoked either explicitly by executing FRE or implicitly whenever free space is requested and not enough free space is available.

For any request for space, garbage collection occurs only once. If after this garbage collection there is still not enough free space, the OUT OF MEMORY error message is displayed. Garbage collection refers to the process of moving all active strings to the top of BASIC memory and then resetting the pointer to the bottom of active strings to this new bottom. The bottom of active strings is the same as the top of available free space plus one.

What exactly is an active string as compared to an inactive string? Well, consider this short example.

```
10 A$ = "LE" + "CANDIRU"
20 A$ = "COMME" + "CA"
```

In this example, "COMMECA" is an active string while "LECANDIRU" is an inactive string. When statement 20 is executed, the string variable A$ gets a new string descriptor that points to this string "COMMECA" which is located in the active string area. However, the string "LECANDIRU" that was created in the active string area in statement 10 still exists in the active string area. It is just sitting there wasting space since there is no string descriptor that refers to it. If garbage collection is invoked, then the "COMMECA" would be moved to the top of the active string area thereby overlaying at least a portion of the "LECANDIRU" string.

While most of the string operations are described in this section, certain other aspects of string operations are described in other sections. Assigning values to string variables can be found in the discussion of LET. Garbage collection and allocation for new strings is found in the section on memory allocations and moves. Function invocation treats the special processing for LEFT$, RIGHT$, and MID$. Expression evaluation details detection of string concatenation. PRINT treats its operations on strings. Mention is also made in the other commands such as OPEN, CLOSE, LOAD, and SAVE.

One thing that can't be done with strings is to create string functions through the DEF FN statement.

Point (50) to Current String Descriptor and Call Allocate Space for String B/D475–B/D47C

Called by:
JSR at A/CA56 LET; JSR at B/D65D Concatenate; JSR at B/D4C0 Scan String, Set Up Descriptor, and Possibly Create String in Active Area.

Operation:
1. Load the X-register from 64 and the Y-register from 65. (64) is the pointer to the string descriptor for this string.
2. Store the X-register into 50 and the Y-register into 51.
3. Fall through to B/D47D to allocate an area for this string.

Allocate Space for String and Length of String and Pointer to Allocated Area B/D47D–B/D486

Called by:
Fall Through from Set (50) to Point to Current String Descriptor; JSR at B/D6F3 CHR$; JSR at B/D70F LEFT$, RIGHT$, MID$.

Entry Conditions: Accumulator has length of string to be allocated.

Operation:
1. JSR B/D4F4 to allocate space for this string or to display OUT OF MEMORY if there's not enough free space for this string.
2. Store the length of the string in 61 and a pointer to the area allocated to the string in (62).

Scan String, Set Up Descriptor, and Create String in Active String Space If String Did Not Originate in Program Text Area B/D487–B/D4C9

Called by:
BEQ at B/D473 STR$; JSR at A/CEC6 Evaluate Expression Within Quotes; JSR at A/CB1E PRINT—for String; JSR at A/CABF PRINT—for Numeric Value That Has Been Converted to String; Alternate B/D48D JSR at A/CC7D GET/INPUT/DATA.

Operation:

1. Store $22, ASCII code for ", in 07 and 08 to serve as delimiters in the search for the end of the string.
2. B/D48D: This is the entry point for GET/INPUT/DATA. If it's GET, 07 and 08 are both set to $00. If it's INPUT or DATA and if the first character scanned was a quote, store $22 (") in both 07 and 08; otherwise, store $3A (:) in 07 and $2C (,) in 08.
3. Initialize (6F) to point to the string using the accumulator and Y-register values at entry.
4. Initialize (62) to point to this same string from the accumulator and Y-register.
5. Set the Y-register to $FF. The first INY in the loop that starts in step 6 makes the Y-register initially zero.
6. INY.
7. Load the accumulator from (6F),Y to load the next byte of data.
8. If this byte is $00, indicate the end-of-string, then branch to step 12.
9. Compare the accumulator to 07. If equal, branch to step 11.
10. Compare the accumulator to 08. If not equal, branch to step 6 to retrieve the next byte of the data.
11. If the accumulator is equal to 07 or 08, compare to $22 ("). If equal, branch to step 13 with the carry being set as a result of this compare. This final quote is not considered part of the string. Remember that the Y-register started its count from minus one.
12. CLC. Branch here if the end-of-string $00 byte is found, or fall through from step 11 if the final character is not a quote.
13. Store the Y-register in 61, thus storing the number of characters in the string (not counting the terminating quote, $00, comma, or colon). Also, the initial quote (or initial comma) is not included. Thus the Y-register only has the actual number of characters in the string excluding the quotes.
14. TYA setting the accumulator to the number of characters in the string.
15. Add the length of the string in the accumulator to (6F), the pointer to the start of the string, and store the result in (71). Thus (71) now points to the character immediately following the string.

16. If 70 is zero, then the string originated in page 0. This is true if the string starts at $00FF, as is the case with STR$ or PRINT for a numeric value that has been converted to a string. Branch to step 19 if the string did originate in page 0.
17. Compare 70 to $02 to see if the string originated in page 2, which is the case if the string was received in the BASIC input buffer, as is done during INPUT.
18. If the string did not originate in page 2 (or in page 0), branch to B/D4CA, the following routine which creates a temporary string descriptor for this string. Thus strings that originate in the program text area are not copied to the active string area in this routine.
19. For strings that did originate in page 2 or page 0, transfer the Y-register to the accumulator to give the length of this string.
20. JSR B/D475 to allocate space for this string, storing a pointer to the top of this allocated area in (35). If not enough free space exists, display OUT OF MEMORY. At exit, the accumulator = length of the string, and the X-register and Y-register are a pointer to the area that was allocated. The subroutine also stores the length of the string in 61 and a pointer to the area allocated in (62).
21. Set the X-register = 6F and the Y-register = 70 to retrieve the pointer to the start of the string.
22. JSR B/D688 to copy the string from the location pointed to by (6F) to the location pointed to by (35), which is the top of the area allocated for the string. This copy routine adds the length of the string found in the accumulator to (6F) to obtain the end of the string and then performs the copy working from the end of the string to the front of the string. The copy does not copy the quote marks; therefore, the active strings are stored without their surrounding quotes.
23. Fall through to B/D4CA to create a string descriptor in the temporary string descriptor stack.

Create String Descriptor in Temporary String Descriptor Stack B/D4CA-B/D4F3

Called by:
Fall Through from Scan String for Page 0 or Page 2 String;

BNE at B/D4BD Scan String for Program Text Strings; JSR at B/D674 Concatenation; JSR at B/D729 LEFT$, RIGHT$, MID$.

Operation:

1. If the temporary string descriptor stack is full, display FORMULA TOO COMPLEX and return to the Main BASIC Loop. The string descriptor stack at 19–21 has room for three string descriptors. The full stack condition is indicated when 16, the pointer to the next area available for the descriptor, is equal to $22.
2. Store 61, the length of the string, and 62 and 63, the pointer to the string, in the string descriptor stack entry pointed to by the X-register which was loaded from 16, a pointer to the next available string descriptor entry.
3. Store the X-register in 64, which now points to the most recent string descriptor stored.
4. Store zero in 65 and in 70.
5. Set 0D to $FF to indicate that a string variable is being processed.
6. Also store the X-register in 17, which now points to the most recent string descriptor stored.
7. Increment the X-register three times to point to the next entry in the string descriptor stack.
8. Store the X-register into 16, which again points to the next available entry in the string descriptor stack.

Return String Length and Pointer to String

If a string descriptor is the last entry on the temporary descriptor stack, delete this entry; and if it is at the bottom of active strings, then also reset the pointer to the active string above this string.

B/D6AA-B/D6DA

Called by:

JSR at B/D041 Compare Strings; JSR at B/D667, B/D671 Concatenate Strings; JSR at B/D716 LEFT$, MID$, RIGHT$.

Entry Conditions:

Accumulator and Y-register point to the string descriptor for this string.

Operation:

1. Store the accumulator in 22 and the Y-register into 23.

2. JSR B/D6DB to see if this string descriptor was the last entry from the temporary string descriptor stack. If so, delete this descriptor from the temporary string descriptor stack. Also reset 16 to point to the just deleted slot, and 17 to point to the previous slot.
3. If the routine at B/D6DB did not find the pointer passed to it equal to the last string descriptor pointer on the stack, the BNE condition of Z=0 is set upon return. Push this status onto the stack.
4. Now using (22), the pointer to the string descriptor, retrieve the length of the string and store in the accumulator. Also retrieve the low-order pointer to the string and store it into the X-register and the high-order pointer into the Y-register.
5. PLP to retrieve the status from step 3.
6. BNE to step 12 if the string descriptor did not point to the last active string used in the temporary string descriptor stack. If this string descriptor did not point to the last active string, continue with step 7.
7. Compare the X-register and the Y-register, now pointing to the string (33), the pointer to the bottom of active strings. If not equal, branch to step 12.
8. If equal, then this string is the last active string in the active string area.
9. Push the length of the string onto the stack.
10. Add the length of the string to (33) and store the result back in (33), thus resetting the pointer to the bottom of active strings to be one position above this string, effectively deleting it from the active string area.
11. Pull the length of the string from the stack.
12. Store the X-register in 22 and the Y-register in 23.
13. RTS with (22) (and also the X-register and Y-register) pointing to this string and with the accumulator containing its length.

Return String Length and Pointer to String from String Descriptor Pointed to by (64) B/D6A6–B/D6A9

Called by:
JSR at A/C9E0 LET for TI$; JSR at A/CB21 PRINT String; JSR at B/D034 Compare Strings; JSR at B/D381 FRE.

Entry Conditions:
(64) points to the string descriptor for this string.

Operation:
1. Load the accumulator from 64.
2. Load the Y-register from 65.
3. Fall through to B/D6AA to return the string length in the accumulator and the pointer to the string in (22) and also in the X-register and Y-register. Also delete the string descriptor from the temporary stack if it's the last entry, and if at the bottom of active strings, reset pointer to the active strings to point above this string.

Type Check for String Variable and If String Then Return String Length and Pointer to String B/D6A3-B/D6A5

Called by:
JSR at B/D782 Check String Type, Return Pointer to String, and Prepare for Converting to Numeric Variable; JSR at E25A/E257 OPEN/CLOSE.

Operation:
1. JSR A/CD8F to type check the most recent variable flag, 0D, to see if a string. If not a string, display TYPE MISMATCH and return to the Main BASIC Loop.
2. Fall through to B/D6A6 to return the string length in the accumulator and the pointer to the string in (22) and also in the X-register and Y-register. Also delete the string descriptor from the temporary stack if it's the last entry, and if at the bottom of active strings, reset the pointer to active strings to point above this string.

Clean Up the Temporary String Descriptor Stack B/D6DB-B/D6EB

Called by:
JSR at A/CA6C LET: JSR at B/D6AE Return String Length and Pointer to String.

Operation:
1. If the accumulator and Y-register point to a different string descriptor than that pointed to by (17), the pointer to the last used string descriptor, then exit with Z=0.

2. If the pointers to the descriptors do match, delete this last entry from the temporary string descriptor stack. Reset 16, the pointer to the next available slot, to point to this string descriptor, and then subtract 3 from 17 to have it point to the previous entry.
   Exit with Z=1.

Check for String Type, Get Pointer to String and Length, and Prepare for Conversion to Numeric Type B/D782-B/D78A

Called by:
JSR at B/D77C LEN: JSR at B/D78B ASC; JSR at B/D7AD VAL.

Operation:
1. JSR B/D6A3 to verify that the argument to LEN, ASC, or VAL is a string. If it isn't, display TYPE MISMATCH and return to the Main BASIC Loop.
2. If the argument is a string, return with the accumulator containing the length of the string and the X-register (as well as (22)) pointing to the string.
3. Store zero into 0D to indicate that the result of LEN, ASC, or VAL will be of the numeric type.
4. Transfer the accumulator to the Y-register, which now contains the length of the string.

String Concatenation B/D63D-B/D679

Called by:
JMP at A/CDE5 Main Expression Evaluation When + Found Following a String Variable or Constant.

Entry Conditions:
64 has pointer to the string descriptor for the left operand.

Operation:
1. Called after the first string (the left operand) has been located, push (64), the pointer to the string descriptor for this left operand, onto the stack.
2. JSR A/CE83 to evaluate the string expression following the +, the right operand. (64) contains a pointer to the string descriptor for this right operand upon return.

3. JSR A/CD8F to verify that this second parameter is a string value. If not, display TYPE MISMATCH and return to the Main BASIC Loop.
4. Restore the string descriptor for the first string by pulling values from the stack that were stored in step 1, and store this pointer to the descriptor for the left operand string into (6F).
5. Add the length of the two strings, the first byte of each string descriptor, and if the sum, which is in the accumulator, is greater than 255, then display the STRING TOO LONG error message and return to the Main BASIC Loop.
6. JSR B/D475 to allocate space for this string from the area directly below the pointer to the bottom of active strings. The pointer to the bottom of active strings is reset to the start of this allocated area, which is enough to contain both strings.

   Also, (50) is set to be equal to the pointer to the descriptor for the second string. (61) is the length of the string (the concatenated string) and (62) is a pointer to the string.
7. JSR B/D67A to copy the left operand string, whose string descriptor is pointed to by (6F), to the start of the allocated area at (35). The copy operation actually works by copying from the end of the string to the start. When the copy finishes, it adds the length of the string just copied to (35), resetting (35) to point to the location just past the end of the copied string.
8. Retrieve the pointer (50) to the string descriptor for the right operand string.
9. JSR B/D6AA to delete the string descriptor for the right operand string from the temporary string descriptor stack, returning with (22) pointing to the string and accumulator containing its length.
10. JSR B/D68C to copy this right operand string pointed to by (22) to the area pointed to by (35), which now points one location past the end of the active string area where the left operand string was stored.
11. Load the accumulator from 6F and the Y-register from 70 to retrieve the pointer to the string descriptor for the left operand string.

12. JSR B/D6AA to delete the string descriptor of the left operand string from the temporary string descriptor stack.
13. JSR B/D4CA to create a new string descriptor in the temporary string descriptor stack taking the length of the string, 61, and the pointer to the string (62) from values set in step 6.

    Upon return, (64) points to the string descriptor just created on the temporary string descriptor stack.
14. JMP A/CDB8 to step 9 of Main Expression Evaluation to evaluate the rest of the expression now that string concatenation is complete. Nothing prevents another concatenation from occurring again in the expression, so concatenation can occur more than once in an expression.

FRE
B/D37D-B/D390

Called by:

Execute FRE Statement.

FRE is included with the string routines because in addition to returning the number of bytes of free area left, it also forces garbage collection of active strings.

Operation:

1. See if the argument to FRE is numeric, and if true, branch to step 3.
2. For a string argument, JSR B/D6A6 to delete the descriptor in the temporary string descriptor stack for this string argument just given.
3. JSR B/D526 to perform garbage collection, moving only strings which have descriptors either in the program text or in the temporary stack descriptor to active string space. This compacts the strings still in active use. It also resets the pointer to the bottom of active strings, (33).
4. Subtract (31), the pointer to the start of the free area, from (33), the pointer to the bottom of active strings (the end of the free area+1). The difference is left in the accumulator and Y-register.
5. Fall through to B/D391 to convert the accumulator and Y-register to a floating point number in FAC1 and then exit.

STR$
B/D465-B/D474

Called by:
Invoke Function for STR$; Alternate B/D46F: JMP at A/CF59 Locate Variable for Expression Evaluation.

Operation:

1. JSR A/CD8D to type check the variable within parentheses, which has been reduced if it's a numeric to a number in FAC1. If it's nonnumeric, display TYPE MISMATCH and return to the Main BASIC Loop.
2. JSR B/DDDF to convert FAC1 to an ASCII string, storing the string starting at $00FF. Because this conversion routine is the normal floating point to ASCII conversion routine, the argument to the string may be stored in scientific format.
3. Remove the return address of this routine.
4. B/D46F: Load the accumulator with $FF and the Y-register with $00 to point to this string.
5. Branch to B/D487 to allocate space for this string, copy the string starting at $00FF to the string memory. Also copy the descriptor for the string to the temporary string descriptor stack. If the normal entry point was used, control returns to the Invoke Function routine.

ASC
B/D78B-B/D797

Called by: Invoke Function for ASC.

This routine will return only the ASCII value of the first character if the string specified is longer than one character.

Operation:

1. The argument for the string has been copied to active string memory if it's a string constant. If it's either a string constant or a string variable, the temporary string descriptor stack holds a descriptor pointing to this string and containing its length. JSR B/D782 to store the length of the string parameter into the Y-register and the pointer into (22). Delete the string descriptor from the string descriptor stack; and if it's a string constant, reset the pointer to active strings past this string so that it is no longer in the active area. (It hasn't been deleted; just the pointer has been reset.)

2. If the length of this string is zero, such as for ASC(""), then display the ILLEGAL QUANTITY error message and return to the Main BASIC Loop.
3. Load the accumulator with the first character from the string and transfer this character to the Y-register.
4. JMP B/D3A2 to convert the integer value in the Y-register into a floating point value in FAC1. Exit back to the Invoke Function routine.

VAL
B/D7AD-B/D7EA

Called by:
Invoke Function for VAL; Alternate B/D7E2: JSR A/CC80 GET/INPUT/READ; JMP at A/CEC9 Expression Evaluation.

Operation:

1. JSR B/D782 to store the length of the string parameter into the Y-register (also into the accumulator) and the pointer to the string in (22). Remove the string descriptor for this string from the temporary string descriptor stack.
2. If the length of the string returned is zero, JMP B/D8F9 to store zero into 61 and 66, setting FAC1 to represent zero.
3. Save (7A), the text pointer that points to the first nonblank character following the argument's right parenthesis, in (71) so that it can later be restored.
4. Store (22), the pointer to this string, into (7A).
5. Add the length of the string to (22) to obtain a pointer to the end+1 of the string and store this in (24).
6. Load the accumulator from (24),0 to retrieve the first byte past the end of the string and push it onto the stack.
7. Store $00 in the byte past the end of the string to force an end-of-string indicator to be later detected.
8. JSR CHRGOT to retrieve the first character of the string pointed to by (7A).
9. JSR B/DCF3 to convert the ASCII string to a floating point number in FAC1. The string ends when the first character is detected in the string that is not a digit, E, ., +, or −. That is, the standard rules for representing a number in normal or scientific notation apply here. The string is forced to end when the $00 end-of-string byte is detected.
10. PLA to restore the saved character past the end of this string and restore it to its position.

11. B/D7E2: Store the pointer to the first nonblank character past the argument or string from (71) into (7A), the text pointer.
12. If the normal entry point was used, the RTS returns to the Invoke Function routine.

CHR$
B/D6EC–B/D6FF

Called by: Invoke Function for CHR$.

Operation:
1. JSR B/D7A1 to see if the parameter for CHR$ evaluates to a numeric value. If not, display TYPE MISMATCH. The floating point to integer conversion routine is used. Also see if it is in the range 0–255; and if it's not, display ILLEGAL QUANTITY. If either of these errors occurs, return to the Main BASIC Loop.
2. Step 1 returns with the X-register holding the integer value from 0–255. Push this onto the stack.
3. Load the accumulator with $01 to indicate a one-byte long string is requested.
4. JSR B/D47D to allocate an area in string memory for this string, returning with (62) pointing to the area allocated.
5. Pull the integer from the stack and store at (62),0.
6. Pull the return address of this routine from the stack and discard.
7. JMP B/D4CA to store descriptor for the string in the temporary string descriptor stack and exit.

LEN
B/D77C–B/D781

Called by: Invoke Function for LEN.

Operation:
1. JSR B/D782 to store the length of the string parameter for LEN into the Y-register. Also, if necessary, remove the string descriptor for this string from the temporary string descriptor stack.
2. JMP B/D3A2 to convert the integer value in the Y-register into a floating point value in FAC1, then return to the Invoke Function routine.

LEFT$
B/D700–B/D705

Called by: Invoke Function for LEFT$.

Entry Conditions:
The second parameter for LEFT$ and the pointer to the string descriptor for the first parameter have been pushed onto the stack.

Operation:

1. The function evaluation routine at A/CFA7 that detected the LEFT$ has already examined the first and second parameters of the argument. The first parameter must be a string, or a TYPE MISMATCH error occurs. The second parameter must be a number in the range 0–255, or an ILLEGAL QUANTITY error occurs. Also display SYNTAX ERROR if the arguments are not separated by a comma.
2. JSR B/D761 to retrieve the two parameters from the stack, returning a pointer to the string descriptor in (50) and the length, which is the second parameter, in the accumulator and also in the X-register. Also set the Y-register to zero.
3. Compare the length of the string specified in the string descriptor to the second parameter now in the accumulator.
4. TYA to set the accumulator to zero.
5. Fall through to B/D706 to perform common LEFT$, RIGHT$, MID$ processing.

Example of LEFT$:

Given the string A$ = "HI THERE"
Determine: X$ = LEFT$ (A$, 2)
Assuming HI THERE is stored at:

50	**H**
51	**I**
52	**space**
53	**T**
54	**H**
55	**E**
56	**R**
57	**E**

In Perform Common LEFT$, RIGHT$, MID$ Processing
After step 11: Accumulator = $00 (offset from which to begin copy)
After step 12: Y-register = $02 (length of string to copy)
After step 13: (22) = 50 + 0 = 50

Step 15 copy operation:
(22),1 = I copied to (35),1
(22),0 = H copied to (35),0.

Perform Common LEFT$, RIGHT$, MID$ Processing B/D706–B/D72B

Called by:
Fall Through from LEFT$; JMP at B/D734 RIGHT$; Alternate B/D70D: BCS at B/D755 MID$; Alternate B/D70E: BCC at B/D75B MID$, BCS at B/D75F MID$.

Operation:

1. BCC to step 4 if the length of the substring (the second parameter) of LEFT$ or RIGHT$ is less than the length of the string.
2. For a substring specified with a length equal to or greater than the length of the string (for LEFT$ or RIGHT$), load the length of the string from the string descriptor and store in the X-register. This value in the X-register is later used to determine how many characters to copy.
3. Set the accumulator to zero. This value is later added to the start of the string to obtain the character from which the LEFT$ or RIGHT$ begins its copy. For LEFT$ or RIGHT$ operation that has specified a longer substring than the string, this zero in the accumulator forces the copy to start from the first character of the string.
4. Push the accumulator onto the stack, which contains the offset into the string from which LEFT$ or RIGHT$ begins its copy. For LEFT$ the accumulator is always zero. For RIGHT$ it is zero if the substring is specified as being longer than the string; otherwise, it is the length of the string minus the length of the substring.
5. B/D70D: Transfer the X-register to the accumulator.

   If the substring is longer, the X-register contains the value of the string's length for LEFT$ or RIGHT$. For MID$, however, if the second parameter, which specifies the starting position of the substring, is greater than the length of the string, the X-register contains zero.

   If the substring is smaller, the X-register holds its value for LEFT$ or RIGHT$. MID$ enters at step 6 if its substring starting location is less than the length of the string.

6. B/D70E: Push the accumulator, which contains the length of the string to be retrieved, onto the stack.
7. JSR B/D47D to allocate space in the active area of string memory for this string whose length is stored in the accumulator, and returning with the length of the string in 61 and a pointer to the area allocated in (62).
8. Set the accumulator from 50 and the Y-register from 51 to retrieve the pointer to the string descriptor for this string.
9. JSR B/D6AA to delete this string descriptor from the temporary string descriptor stack. At exit, (22) is the pointer to the area allocated for this new string.
10. PLA to retrieve the length of the substring to be returned and transfer to the Y-register.
11. PLA to pull the offset into the string from which the copy will begin. For LEFT$ this is always zero.
12. Add the accumulator to (22) to determine where to start retrieving characters, leaving the result in (22).
13. Transfer the Y-register to the accumulator to restore the length of the substring that is to be retrieved.
14. JSR B/D68C to copy the string from the area starting at (22), to the area allocated for this string at (35). The Y-register is used as the offset from (22) and from (35) in this copy. Before each character is copied, the Y-register is decremented. The character is then copied. Then if the Y-register is zero, the copy is complete; otherwise, repeat this loop of decrement, copy, and check for zero.
15. JMP B/D4CA to create a descriptor in the temporary string descriptor stack for this newly created string. Also set (64) to point to this newly created string descriptor. Finally, set 0D to $FF to indicate a string expression.

RIGHT$
B/D72C–B/D736

Called by: Invoke Function for RIGHT$.

Operation:

1. JSR B/D761 to get the two parameters for RIGHT$, returning a pointer to the string descriptor in (50) and the second parameter which specifies the length of the substring to be returned from the right side of the string. This second parameter is returned in the X-register and in the accumulator. Also, set the Y-register to zero.

2. CLC.
   SBC (50),Y.

   Subtract the length of the string plus one from the length for the substring. Note how SBC works with the carry clear. For a carry clear, SBC takes its operand value, adds one, and computes the two's complement.

   Here are a couple of examples assuming that A$ ="ABCDEFGH" and thus has a length of eight.

Example 1:

RIGHT$(A$,2)

$$\begin{array}{lcr}
\text{\$02} & = & 0000\ 0010\\
\text{\$08+1 Then Two's Comp} & = & \underline{1111\ 0111}\\
 & 0 & 1111\ 1001
\end{array}$$

$$\begin{array}{lcrl}
\text{\$08} & = & 0000\ 1000 & \\
+\text{\$01} & = & \underline{0000\ 0001} & \text{(adding complement of carry)}\\
 & & 0000\ 1001 &
\end{array}$$

Convert to two's comp:

$$\begin{array}{lr}
 & 1111\ 0110\\
+ & \underline{\qquad\qquad 1}\\
 & 1111\ 0111
\end{array}$$

Example 2:

RIGHT$ (A$,10)

$$\begin{array}{lcr}
\text{\$0A} & = & 0000\ 1010\\
\text{\$08+1 Then Two's Comp} & = & \underline{1111\ 0111}\\
 & 1 & 0000\ 0001
\end{array}$$

3. EOR $FF (does not alter carry status)
   Continuing with above examples:

Example 1:

$$\begin{array}{lrl}
 & 1111\ 1001 & \\
\text{EOR} & \underline{1111\ 1111} & \\
 & 0000\ 0110 & = \text{\$06 in accumulator}
\end{array}$$

Example 2:

$$\begin{array}{lrl}
 & 0000\ 0001 & \\
\text{EOR} & \underline{1111\ 1111} & \\
 & 1111\ 1110 & = \text{\$FE in accumulator}
\end{array}$$

4. Now JMP B/D706 to the common processing for LEFT$, RIGHT$, MID$. The carry setting from step 2 determines whether to start the copy operation from the first character of the string or from the value in the accumulator. If the carry is clear, the copy starts from the value set in the accumulator above in step 3. If the carry is set, the copy starts from the first position of the string. The length of the string to be copied is passed in the X-register. If you specify a substring for RIGHT$ longer than the length of the string (carry set), B/D706 resets the X-register to the length of the string and resets the accumulator to zero.

   Example of RIGHT$:
   Given the string A$="HI THERE"
   Determine: X$ = RIGHT$(A$,2)
   Assuming HI THERE is stored at:

   50 H
   51 I
   52 space
   53 T
   54 H
   55 E
   56 R
   57 E

   In Perform Common LEFT$, RIGHT$, MID$ Processing
   After step 11: Accumulator = $06 (offset from which to begin copy)
   After step 12: Y-register = $02 (length of string to copy)
   After step 13: (22) = 50 + 6 = 56
   Step 15 copy operation:
   (22),1 = E copied to (35),1
   (22),0 = R copied to (35),0.

MID$
B/D737-B/D760

Called by: Invoke Function for MID$.

If the second parameter for MID$ is zero, an ILLEGAL QUANTITY error message is displayed. An example of such an invalid specification is MID$(A$,0,5).

Operation:

1. Set 65 to $FF to set a default length for the substring of 255 (decimal).

2. JSR CHRGOT to retrieve the last character past the second parameter.
3. If this character is) then no third parameter has been specified, so branch to step 6.
4. JSR A/CEFD to check for a comma following the second parameter; if not found, display SYNTAX ERROR.
5. JSR B/D79E to evaluate the third parameter which specifies the substring length. If it's not a numeric value, display TYPE MISMATCH. If not a value from 0 to 255 (decimal), display ILLEGAL QUANTITY. 65 holds the 0–255 (decimal) quantity upon return.
6. JSR B/D761 to retrieve the first two parameters from the stack. Return a pointer to the string descriptor in (50) and the second parameter, which indicates the starting location from which the copy is to be made, in the accumulator and also in the X-register.
7. If this second parameter is zero, display ILLEGAL QUANTITY. Thus it can't be zero, contrary to the documentation in both the *VIC Programmer's Reference Guide* and the *Commodore 64 Programmer's Reference Guide.*
8. DEX to decrement the second parameter and push onto the stack, thus saving the starting location in the string from which the copy is to begin.
9. CLC.

   LDX #$00 to prepare for returning the null string if the second parameter is greater than the length of the string. SBC (50),Y to subtract the length of the string from the second parameter.

   Here are a couple of examples assuming that A$="ABCDEFGH" and thus has a length of eight.

Example 1:

MID$(A$,4,2)

$03	=	0000 0011
$08 + 1 Then Two's Comp	=	1111 0111
	0	1111 1010

Example 2:

MID$(A$,12,2)

$0B	=	0000 1011
$08 + 1 Then Two's Comp	=	1111 0111
	1	0000 0010

10. If the carry is set as a result of the previous step, you are trying to specify a starting position greater than the length of the string. In this case, BCS to B/D70D with the X-register = $00 to return a null string.
11. For a carry clear in the above operation, you have specified a starting position within the string. Exclusive OR the accumulator with $FF. Thus in example one above, the accumulator is now 0000 0101 = $05.
12. Compare the accumulator to 65, the third parameter which specifies the length of the substring to be returned.
13. If the accumulator is greater than 65, set the accumulator to 65, or leave the accumulator unchanged. Thus, in example 1, the accumulator is changed to $02.
14. Branch to B/D70E to perform the common MID$, LEFT$, RIGHT$ operation.

Example of MID$:

Given the string A$ = "HI THERE"
Determine: X$ = MID$(A$,4,2)
Assuming HI THERE is stored at:

50	**H**
51	**I**
52	**space**
53	**T**
54	**H**
55	**E**
56	**R**
57	**E**

In Perform Common LEFT$, RIGHT$, MID$ Processing
After step 11: Accumulator = $03 (offset from which to begin copy)
After step 12: Y-register = $02 (length of string to copy)
After step 13: (22) = 50 + 3 = 53
Step 15 copy operation:
(22),1 = H copied to (35),1
(22),0 = T copied to (35),0.

Pull Parameters from Stack for LEFT$, RIGHT$, MID$ B/D761–B/D77B

Called by:

JSR at B/D700 LEFT$; JSR at B/D72C RIGHT$; JSR at B/D748 MID$.

Operation:

1. When the Invoke Function routine determines that a function has more than one parameter, it pushes the first and second parameters onto the stack, so these values are on the stack for LEFT$, RIGHT$, MID$.
2. JSR A/CEF7 to check for a) as the next character of the argument and display SYNTAX ERROR if it's not found.
3. Pull the return address for this routine and temporarily save it in 55 and the Y-register.
4. Pull two more bytes from the stack, which is the return address to MID$, LEFT$, or RIGHT$. Thus these functions will return to the instructions past JSR at A/CDB1 or JSR at B/D643.
5. Pull the second parameter from the stack and transfer to the X-register.
6. Pull the next two bytes containing the pointer to the string descriptor for the first parameter and store in (50).
7. Restore the return address for this routine from 55 and the Y-register.
8. Set the Y-register to zero.
9. Transfer the X-register to the accumulator, which now has the second parameter, as well.
10. RTS.

Statement Execution

Statement execution is one of the simpler processes in BASIC. All that is required to prepare for statement execution is that (7A), the text pointer used by CHRGET, point to the start (minus one) of the statement to be executed.

If a direct mode program statement has been entered, (7A) points to 01FF, which is the start (minus one) of the BASIC input buffer at 0200–0258 into which the direct mode statement has been stored in tokenized format.

After RUN is executed, (7A) is reset to point to the start of the program text area (minus one). The reason the address minus one is used is that the next CHRGET always increments (7A) before retrieving a character.

Statement execution retrieves the first nonspace character in the statement to be executed. If this character has its high-order bit off, it is assumed that this statement is an implicit LET statement. An implicit LET statement starts with the name of a variable rather than with the token for LET.

If this character has its high-order bit on, a check is made to see if the value is in the range of tokens that represent BASIC commands. This range is $80 through $A2 inclusive. If the value is not in this range, one final check is made to see if this is a token for GO followed by a token for TO. If the byte with its high-order bit on is neither a command token nor the GO token followed by the TO token, then SYNTAX ERROR is displayed.

If an implicit LET or the GO TO tokens are detected, direct jumps are made to execute the LET or GOTO routines. For all other command tokens, the address of the command is stored on the stack from a table of command addresses that starts at A/C00C. CHRGET is then jumped to, and the RTS from CHRGET passes control to the address of the command routine. The command routine (including LET or GOTO) is responsible for processing all the remaining characters in the statement line, and returning to statement execution control when the command detects an end-of-statement caused by a colon (:) or a $00 end-of-line byte. Also, (7A) points to this final byte of the statement upon return from the command.

If the statement just executed was a direct mode statement, READY is displayed, and control returns to the Main BASIC Loop to await subsequent input.

If the statement just executed was in program mode, then the pointer (7A) is stored to the final byte of this statement in a pointer which can later be used by CONT to resume program execution.

If the statement was a program mode statement, it is determined whether the end of the program has been reached. If the end has been reached, display READY and return to the Main BASIC Loop. If the end has not been reached, set a pointer to the line number of the next BASIC line following the line that was just executed and point (7A) to the first character (minus one) of the tokenized text area in this line. However, if the statement ended with a colon, there is no need to do the action in the previous sentence since (7A) points to the first character (minus one) of the statement following the colon and the line number remains the same.

The normal action for BASIC program execution is to continue in this orderly sequence, retrieving statement after statement in line after line of the program until the end of the program is reached.

Several commands can be used to change which line will be the next one from which statements will be executed. However, only CONT will allow execution to continue from the second or latter statements in a single BASIC line.

The commands that allow you to change the sequence of statement execution are GOTO, GOSUB/RETURN, ON, FOR-NEXT, IF-THEN, and RUN followed by a line number. The various effects of these commands are treated in the next section.

You can intercept the statement execution control routine since it is reached by using a vector at (0308). Normally this vector points to A/C7E4.

Statement Execution Control A/C7E1–A/C7EC

Called by:

JMP at A/C499 Main BASIC Loop for Direct Mode Statements; Fall through and also BEQ at A/C809 from Test for

Break, End-of-Statement End of Program, Prepare Pointers for Executing Next BASIC Statement.

Operation:

1. JMP (0308). The default for the vector at 0308 is A/C7E4, step 2. To intercept statement execution control, modify the vector at 0308 to point to your routine. For example, if you decided to accept tokens with values $CC–$FE, your alternate statement execution control could handle these other tokens.
2. A/C7E4: At this point (7A), the text pointer, is pointing to 01FF if this is a direct mode statement. If RUN was the last direct mode command entered (RUN is also acceptable in program mode although it results in recursion), then (7A) points to the start of the program text area minus one.

   JSR CHRGET to retrieve the next nonspace character in the area pointed to by (7A). Since both direct mode statements and program statements have been tokenized by the time this step is executed, the byte that CHRGET retrieves should be either a one-byte token or the first byte of the name of a variable. If neither of these two conditions is true, SYNTAX ERROR is going to result.
3. JSR A/C7ED to execute this statement by determining which BASIC command starts the statement, pushing this command's return address onto the stack, and then JMP CHRGET to retrieve the first character of the parameter (or the end-of-statement $00 byte). Upon the RTS from CHRGET, the return address to the BASIC command is pulled from the stack. The BASIC command is then executed, and when it finishes, its RTS returns control to step 4 in this routine.

   Normally RUN is entered in direct mode and would be executed from this step. After it is executed, the text pointer is reset to the start of the program text area minus one. Thus direct mode statements are not looked for following a RUN.
4. JMP A/C7AE to test several things. If the end of the program has been reached or if the statement just executed was in direct mode, display READY and go to the Main BASIC Loop to await the next line of input.

   Also, if in program mode, save a pointer to this statement to allow CONT to continue from this statement.

Check for a valid end-of-statement terminator of a colon (:) or $00 and if neither is found, display SYNTAX ERROR. If this statement does end with $00, set (39) to be the line number of the next BASIC line and point (7A) beyond the two-byte link field and the two-byte statement number to the actual start of the tokenized text for this next BASIC statement. Then, following detection of either a colon (:) or $00, fall through to step 1 of this routine to execute the next statement.

Test for Break, a Colon (:) or $00 End-of-Statement Delimiters, Test for End-of-Program, Prepare Pointers for Execution of Next BASIC Statement If Program Mode A/C7AE–A/C7E0 and A/C807–A/C80D

Called by:
JMP at A/CD75 NEXT; JMP at A/C7EA Statement Execution Control; JMP at A/C89D Goto for RUN or GOSUB.

Operation:

1. JSR A/C82C to see if the STOP key was pressed. If so, display BREAK and restart BASIC with a warm start.
2. Load the accumulator from 7A and the Y-register from 7B to get the pointer to the ending delimiter for the last statement.
3. Compare the Y-register to $02 to see if the statement ended in the BASIC input buffer at 0200–0258. If true, this statement was a direct mode statement (except for CONT which resets (7A)), so branch to step 5 as there is no need to save a pointer for CONT.
4. Store the accumulator in 3D and the Y-register in 3E. (3D) can be used by CONT to determine the pointer to the end of the last statement executed.
5. Retrieve this last byte of the statement pointed to by (7A).
6. If it's not $00, branch to A/C807 (step 13) to check for a colon.
7. For the valid end-of-line $00 byte, now determine whether the end of the BASIC program has been reached. Also determine if this program was really just a direct mode statement. In either case the determination is made by

loading the accumulator with the byte two locations past where (7A) now points.

For a statement in program mode this byte just retrieved is the high-order byte of the link field in the next statement. An end-of-program is indicated by a link field with a high-order byte of $00 since no program text can be stored in page 0.

For a direct mode statement, this byte retrieved is always $00 since $00 is stored two bytes past the end-of-line $00 during tokenization of the direct mode line.

8. CLC and BNE to step 10 if this byte is not zero.
9. If this byte is zero, JMP A/C84B to display READY and go to the Main BASIC Loop to await the next line of input.
10. Since the end of the program has not been reached, set (39) to be the line number of the next BASIC line.
11. Advance (7A), the text pointer, past the two-byte link field and the two-byte line number by adding four to it.
12. Fall through to A/C7E1 to execute this statement pointed to by (7A).
13. A/C807: Branch here if $00 is not found as the end-of-statement. Check for a : as the last byte of the statement and display SYNTAX ERROR for any characters in a statement that have not been processed by the command in the statement. If : is found, branch to A/C7E1 to execute the statement in this line following the colon (:).

Test for Null Direct Mode Line or Null Expression Following THEN A/C7ED–A/C7EE

Called by:
JSR at A/C7E7 Statement Execution Control; JMP at A/C948 THEN portion of IF.

Operation:
1. If zero was returned by CHRGET, indicating the end-of-line $00 byte has been found, branch to A/C82B to RTS.
2. Fall through to A/C7EF to execute the BASIC statement if any value other than zero is returned.

Execute the Current BASIC Statement A/C7EF-A/C806 and A/C80E-A/C81C

Called by:

JSR at A/C95C ON; Fall through from Test for Null Direct Mode Line Or Null Expression Following THEN.

Operation:

1. The accumulator contains the first nonspace byte in the statement retrieved by CHRGET.

   Subtract $80 (with carry set) from the accumulator to determine whether this first character has its high-order bit on.
2. If the carry is clear, branch to step 7 to perform LET. If the carry is clear, this first byte is not a token, so it is assumed to be the first character of a variable name.
3. Compare the accumulator to $23. Since $23 + $80 = $A3, the test here is to see if this token is in the range $80–$A2 inclusive. Only tokens in this range are considered to be commands, except for the special case of GO TO.

   If the accumulator is >= $23, branch to step 8. Otherwise, we now have the token for a BASIC command with the high-order bit set to zero.
4. ASL. This shift multiplies the accumulator, which contains the index into the table of addresses of BASIC commands, by two. Since each address of a command routine takes up two bytes in the table, the accumulator now has the correct offset to the address for this command. Transfer the accumulator to the Y-register.
5. Load the accumulator from A/C00D,Y and push onto the stack. Load the accumulator from A/C00C,Y and push onto the stack. Each address entry in the table contains the low-order byte followed by the high-order byte. The high-order byte is pushed on the stack first, followed by the low-order byte. When the RTS is performed, it pulls off these two bytes from the stack and adds one. This value is then the new value of the program counter, and execution continues at this new address in the program counter. Because RTS adds one to the addresses pulled from the stack, the table entries actually are one byte less than the actual address of the routine.

6. JMP CHRGET to retrieve the first character of the parameter for this command or the end-of-line $00 byte. The RTS from this CHRGET pulls off the address stored on the stack in step 5 and thus executes that BASIC command. After the BASIC command does its RTS, control returns to the routine that called this one, which is back to step 4 of Statement Execution Control.
7. JMP A/C9A5 to execute LET if the statement did not start with a token. When a statement does not start with a token, it is assumed that the statement starts with a variable and that this statement is a LET. If the statement does not start with a variable, LET will produce SYNTAX ERROR.
8. Branch here from step 3 if the accumulator is greater than $23. Compare the accumulator to $4B. $4B + $80 = $CB, the token for GO.
9. If not equal, display SYNTAX ERROR.
10. If the GO token is found, load the accumulator with $A4, the token for TO.
11. JSR A/CEFF to do a CHRGET and then check the value returned to see if it is the token for TO. If not, display SYNTAX ERROR.
12. If the GO token and TO token have been found, JMP A/C8A0 to execute GOTO.

    There is also a token for GOTO, $89, so either GOTO or GO TO (with space[s] between GO and TO) is acceptable.

Table of Addresses of BASIC Commands A/C00C–A/C051.

Used by: Step 5 of Execute the Current BASIC Statement.

The addresses in this table are one less than the actual addresses of the locations for the command routines. These addresses are pushed onto the stack and the branch to the routine is made by using an RTS which adds one to the address it pulls from the stack.

The actual addresses of the BASIC commands are listed below.

Token	Keyword	Address (plus one)
80	END	A/C831
81	FOR	A/C742
82	NEXT	A/CD1E
83	DATA	A/C8F8
84	INPUT#	A/CBA5
85	INPUT	A/CBBF
86	DIM	B/D081
87	READ	A/CC06
88	LET	A/C9A5
89	GOTO	A/C8A0
8A	RUN	A/C871
8B	IF	A/C928
8C	RESTORE	A/C81D
8D	GOSUB	A/C883
8E	RETURN	A/C8D2
8F	REM	A/C93B
90	STOP	A/C82F
91	ON	A/C94B
92	WAIT	B/D82D
93	LOAD	E168/E165
94	SAVE	E156/E153
95	VERIFY	E165/E162
96	DEF	B/D3B3
97	POKE	B/D824
98	PRINT#	A/CA80
99	PRINT	A/CAA0
9A	CONT	A/C857
9B	LIST	A/C69C
9C	CLR	A/C65E
9D	CMD	A/CA86
9E	SYS	E12A/E127
9F	OPEN	E1BE/E1BB
A0	CLOSE	E1C7/E1C4
A1	GET	A/CB7B
A2	NEW	A/C642

Statement Execution and Program Flow

The previous section shows how statement execution proceeds once it is started. It also shows how the location of the next statement to be executed is picked up from the text pointer, (7A), and then that statement is executed. Normally, in program mode, (7A) points to the following statement in the program.

The instructions discussed in this section modify (7A) so that the next statement does not have to sequentially follow in the program the previous statement executed.

This section also includes instructions that initiate execution of program statements from direct mode and that halt the program mode statement execution and return to direct mode.

Usually to start execution of a program, you enter RUN from direct mode. This resets (7A) to point to the start of the BASIC tokenized program minus one. RUN also clears all variables.

Program execution can be started by other means, though. RUN followed by a line number starts execution at that line number after clearing all variables. If the line number does not exist, an UNDEF'D STATEMENT error occurs.

GOTO and GOSUB can also be entered from direct mode to begin program execution. Again, if the line number following GOTO or GOSUB does not exist, an UNDEF'D STATEMENT occurs. Neither GOTO nor GOSUB clears the variables before starting program execution. GOSUB can call either a subroutine that ends in a RETURN or an entire program. If GOSUB calls a program, the GOSUB block remains on the stack when the program finishes execution.

CONT is another method of starting program execution. For CONT, though, program execution is really resumed rather than started from the beginning of a program. CONT starts the program execution based on the pointer to the end-of-statement of the last statement executed in the previous program.

Programs can stop execution and return to direct mode through several methods.

If the routine that retrieves the next statement from the following line finds a high-order link field byte of zero in the next line, program execution is considered complete, and control returns to the Main BASIC Loop which waits for the next direct mode command.

Detection of the STOP key between execution of program lines or execution of the STOP command returns control to the Main BASIC Loop after displaying BREAK IN LINE and the line number.

Execution of the END command also returns control to the Main BASIC Loop to await direct mode commands.

Both END and STOP (and the STOP key detection) save pointers to the end-of-statement of the statement just executed and the line number from which this statement was executed for possible use by CONT.

REM and DATA are both commands that are used to skip the text pointer to the next executable statement. For REM, the text pointer is skipped to the end-of-line byte of the REM line. For DATA, the text pointer is skipped to the end-of-line byte of the DATA statement.

IF and ON both provide means to change which statement will next be executed after the IF or ON. First the expression following IF or ON is evaluated.

For IF, an expression that evaluates to zero is considered false and the remainder of the IF line is skipped with statement execution proceeding with the first statement in the following line. For an expression that evaluates to true, the action following the expression is performed. This action can be a line number or GOTO, either of which transfers control to the first statement in that line. The action can also be execution of any other valid BASIC statement.

For ON the expression becomes an index into a list following a GOSUB or GOTO. A line number in the list corresponding to the index receives control with execution proceeding from that line. If the index is higher in value than the number of line numbers in the GOSUB-GOTO list, statement execution continues with the statement following.

GOSUB and RETURN are intended to function as a pair of commands. The same applies to FOR and NEXT. Also, both of these two pairs of commands use the stack, and the execution of these commands is related in some aspects. See the descriptions of these commands for details.

GOSUB is normally used to pass control to the line number which follows the GOSUB. This line usually starts a subroutine that is ended by a RETURN. When RETURN executes, the program execution continues with the statement following the GOSUB.

FOR-NEXT forms a loop that is executed once or numerous times. Execution of the FOR loop ends when the loop variable becomes greater than or equal to the TO value (for positive steps) or less than or equal to the TO values (for negative steps). The FOR loop includes all of the instructions through its corresponding NEXT.

GOTO can also be used in the program mode to transfer control unconditionally to another program line from which statement execution is to continue.

RUN
A/C871-A/C882

Called by: Execute the Current BASIC Statement for RUN.

Operation:

1. Push the status onto the stack. This status is for the CHRGET that retrieved the first nonspace past the RUN token.
2. JSR FF90 to the Kernal SETMSG routine to turn off the Kernal control and error messages, which are thus not displayed during program mode statement execution.
3. Restore the status from the stack. If Z=0, a parameter followed the RUN. If Z=1, RUN was followed by either a colon (:) or the end-of-line $00 byte.
4. If Z=0, branch to step 6 for the condition when a parameter follows RUN.
5. When no parameters follow RUN, then JMP A/C659 to reset (7A) to point to the start of the BASIC tokenized program area minus one. Thus RUN just changes the location from which BASIC statements are interpreted and executed from the BASIC input buffer at 0200–0258 to the start of the tokenized program.

Then A/C659 does a CLR. CLR resets the pointers for the top of available free space, the start of BASIC arrays, the start of the free area, and the pointer to DATA statements. This deletes all variables, arrays, and active strings. It also flushes the temporary string descriptor stack by resetting its pointer. The RTS from CLR returns to step 4 of Statement Execution Control.

6. Branch here from step 4 if RUN is followed by parameters. JSR A/C660 to do CLR, performing all the operations for CLR mentioned in step 5. Thus RUN wipes out all variables whether or not it is followed by a parameter.
7. JMP A/C897 to GOTO the statement number following RUN. If RUN is followed by a nonnumeric value, the value is resolved to zero. If no line number zero exists, an UNDEF'D STATEMENT error occurs. If RUN is followed by a number, that number must also be an actual line number or UNDEF'D STATEMENT is displayed. It appears that RUN followed by a number containing a fraction uses only the integer part of the number in determining the line number. So if line 10 exists, RUN 10.7 is valid.

Do GOTO or GOSUB or RUN When RUN Is Followed by Parameter A/C897-A/C89F

Called by: Fall through from GOSUB; JMP at A/C880 RUN.

Operation:

1. JSR CHRGOT to retrieve the first character of the parameter of RUN or GOSUB pointed to by (7A).
2. JSR A/C8A0 to perform GOTO.

   GOTO attempts to reset (7A) to the start of the BASIC statement (minus one) whose line number is equal to the parameter following RUN or GOSUB. If the line number cannot be found, display UNDEF'D STATEMENT and return to the Main BASIC Loop.

   If the parameter following GOSUB or RUN is a letter or special character, and if a line number 0 exists, reset (7A) to point to line 0 (minus one). Otherwise, display UNDEF'D STATEMENT and return to the Main BASIC Loop.
3. JMP A/C7AE to set the line number for this line now pointed to by (7A) and also reset (7A) to point to the first

character of the tokenized text area in the line. Then fall through to the next procedure at A/C7E1, Statement Execution Control, which sees if this first statement in the line can be executed, and if so, then executes it.

GOTO
A/C8A0–A/C8D1

Called by:
JMP at A/C81A GO token followed by TO token during Execute the Current BASIC Statement; JSR at A/C89A GOTO for RUN/GOSUB; JMP at A/C945 IF-THEN with THEN Followed by Line Number.

Operation:

1. JSR A/C96B to convert the parameter for GOTO to the two-byte line number in 14 (low-order) and 15 (high-order). If GOTO (or RUN or GOSUB) was followed by a letter, return with the carry set and (14) = $0000.
2. JSR A/C909 to determine the offset to the end of the line and store in the Y-register.
3. See whether a forward jump or a backward jump in the program is being made by subtracting (14) from (39), the current line number.
4. If the carry is set after the subtraction, then (39) >= (14) and the GOTO is for a backward jump. Branch to step 9 if the carry is set.
5. For a forward jump with (14) > (39), transfer the Y-register to the accumulator, which now holds the offset to the end of the line containing the GOTO statement.
6. Add the accumulator to 7A, the low-order byte of the text pointer, leaving the result in the accumulator.
7. Load the X-register from 7B, the high-order byte of the text pointer. If step 6 caused the carry to be set, then INX. The accumulator and X-register now point to the line following the GOTO line so that the search begins there for a forward jump.
8. Branch to step 10.
9. For a backward jump, just load the accumulator from 2B and the X-register from 2C. (2B) points to the start of the tokenized BASIC program area.
10. JSR A/C617 to search through the tokenized BASIC program for a BASIC statement whose line number = (14), the

line number for the GOTO parameter. Begin the search at the location pointed to by the accumulator and X-register. If the line number is found, then upon return the carry is set and (5F) contains the address of the line which contains this line number.

11. If the carry is clear, the line number was not found. In this case, display UNDEF'D STATEMENT and return to the Main BASIC Loop.
12. If the line number is found, reset (7A), the text pointer, to be (5F) minus one. Thus the next statement executed will be the one in the line pointed to by (5F).

Convert ASCII Characters to Two-Byte Line Number in Low-Order/High-Order Format A/C96B-A/C9A4

Called by:
JSR at A/C49C Store/Replace BASIC Lines; JSR at A/C8A0 GOTO; JSR at A/C6B6, A/C6A4 LIST; JSR at A/C962 ON.

The maximum line number is 63999. Since CHRGET skips over spaces, line numbers entered with spaces between the numbers are valid; for example, 1 2 3 8 is the line number 1238.

Operation:

1. Store zero into 14 and 15, which hold the resulting line number. 14 holds the low byte and 15 the high byte.
2. If CHRGET returned with the carry set, RTS since the character is nonnumeric. For example, GOTO X returns with 14 and 15 both zero.
3. SBC $2F. With the carry clear, this is the same as subtracting $30 from the accumulator, thus converting the ASCII digit to a byte in the range $00 through $09.
4. Store the accumulator in 07, which temporarily holds each digit picked off from the ASCII line number.
5. Load the accumulator from 15, the high-order byte of the line number so far.
6. Store the accumulator at 22.
7. Compare the accumulator to $19. $1900 = 6400 (decimal). The comparison is done before the next multiplication of the line number by ten. Thus the compare is to see if the line number is going to be 64000 or larger.

8. If the accumulator is >= $19, display SYNTAX ERROR since the largest line number permitted is 63999.
9. Load the accumulator from 14, the low-order byte of the line number so far.
10. The line number is multiplied by 4 as follows: ASL, ROL 22, ASL, ROL 22.
11. Add the original value of the line number from 14 and 15 to now have the line number equal the line number times 5. Store the result back in 14 and 15.
12. Do an ASL 14 and ROL 15 to now multiply the original line number by 10 to make room for adding the next digit from the ASCII line number.
13. Add 07, which contains this next digit in the line number, to 14 and 15.
14. JSR CHRGET to retrieve the next character of the ASCII line number or the character forming its end.
15. JMP to step 2 to check if this next character is numeric and if so to add it to the line number.

Test for STOP Key
A/C82C–A/C82E

Called by:
JSR at A/C6D1 LIST; JSR at A/C7AE Test for End-of-Statement or End-of-Program and Prepare Pointers for Next Statement.

Operation:
1. JSR FFE1 to the Kernal routine to test for the keyboard STOP key. If detected, return with the carry set. Also, if the STOP key is detected, Z=1 flag (BEQ condition) is set.
2. Fall through to A/C82F to do STOP.

STOP
A/C82F–A/C830

Called by:
Fall through Test for STOP Key; Execute Current BASIC statement for STOP.

Entry Conditions:
From test for STOP Key: Carry Set and Z = 1 if the STOP key is detected. From Execute STOP: Carry Set if $00 or colon retrieved and Z=1.

Operation:

1. If the carry is set, branch to A/C832, step 2 of END.
   This branch would not be taken if the STOP key was not detected or if the STOP command was followed by a numeric parameter.

END
A/C831-A/C856

Called by:

Execute Current BASIC Statement for END; Alternate A/C832: BCS at A/C82F STOP.

Operation:

1. CLC to indicate this is END. (Also clears carry if STOP is followed by a numeric parameter or if the STOP key is not detected.)
2. A/C832: BNE (Z=0) to A/C870 to RTS if END or STOP were followed by a parameter or if the STOP key was not detected. If a parameter followed END or STOP, SYNTAX ERROR is displayed and control returns to the Main BASIC Loop. If the STOP key is not detected, control returns to step 3 of Test for Break at A/C7B1.
3. Load the accumulator and Y-register from (7A), the current text pointer to the end of the END or STOP line or to the end of the previous line if checking for the STOP key.
4. Load the X-register from 3A. The Main BASIC Loop set 3A to $FF if this last statement executed was a direct mode statement.
5. INX, and if it's now zero, the statement just executed was a direct mode statement. If true, branch to step 8.
6. Store the accumulator in 3D and the Y-register in 3E to set (3D) to point to the end of the last statement executed (the end of STOP or END or whatever statement preceded detection of the STOP key).
7. Copy (39), the current line number of STOP or END or of the last line before the STOP key was detected, to (3B), the previous BASIC line number that was executed.
8. Remove this routine's return address from the stack and discard it.
9. If carry is set, which indicates STOP was executed or the STOP key was detected, then JMP A/C469 to display

BREAK IN and the line number, READY, and then continue with the Main BASIC Loop.

10. If carry is clear, END was executed. For the VIC: JMP C474 to display READY and continue with the Main BASIC Loop. For the 64: JMP E386 which JMPs to A474 and also allows test for error messages.

CONT
A/C857–A/C870

Called by: Execute Current BASIC Statement for CONT.

CONT should only be entered in direct mode. CONT can continue from any statement even if it is not the first statement in a line. For example, type in and run this program:

```
30 A=5
40 STOP: PRINT A*2
50 PRINT A*3
```

The computer responds with:

```
BREAK IN 40
```

Enter CONT and press RETURN.

The computer responds with:

```
10
15
```

Operation:

1. If a parameter follows CONT, display SYNTAX ERROR and return to the Main BASIC Loop.
2. If 3E is zero, CONT has been disallowed (such as by CLR or error routine). In this case, display CAN'T CONTINUE error message and return to the Main BASIC Loop.
3. Store (3D), which points to the end of the last statement executed (the end of STOP or END or whatever statement preceded detection of the STOP key) into (7A) so that statement execution will continue at the next statement.
4. Store (3B), the previous BASIC line number, into (39), the current BASIC line number.
5. Return to step 4 of Statement Execution Control. The routine at A/C7AE will then use these new values of (7A) and (39) to determine which BASIC statement to execute next.

ON
A/C94B–A/C96A

Called by: Execute BASIC Statement for ON.

ON can be used in place of several IF-THEN statements. For example:

IF X = 1 THEN GOTO 100
IF X = 2 THEN GOTO 200
IF X = 3 THEN GOTO 300

can be replaced with:

ON X GOTO 100,200,300

An ON statement such as ON X GOTO 10,,30 is valid. However, if X = 2, an UNDEF'D STATEMENT error occurs unless a line number 0 exists.

An ON statement such as ON 3.7 GOTO 30,55,80 is also valid. The 3.7 is converted to fixed point binary so its actual value is 3. The line numbers in the GOTO/GOSUB list must be in the range 0–63999.

If the ON expression evaluates to a number which is greater than the number of items in the list, the program continues with the next statement.

If the ON expression evaluates to 0, this is treated as 256 (decimal). Because of the size limitation of BASIC lines, you can't actually have a list of 256 entries, so the program continues with the next statement.

Operation:

1. JSR B/D79E to evaluate the expression following ON. B/D79E converts the expression into its fixed point binary equivalent, thus dropping any fractional part. If the expression is a string, TYPE MISMATCH is displayed. If the expression evaluates to a number outside the range 0–255, ILLEGAL QUANTITY is displayed. The one-byte 0–255 value is left in 65. The expression evaluation does a JMP CHRGOT before exiting so that upon return the accumulator contains the character that stopped expression evaluation.
2. Push the accumulator containing the character that stopped expression evaluation onto the stack.
3. Compare the accumulator to $8D to see if this character was the token for GOSUB. If so, branch to step 5.

4. Compare the accumulator to $89 to see if this character was the token for GOTO. If not, display SYNTAX ERROR, and the byte pushed onto the stack in step 2 is not pulled. However, any error messages reset the stack pointer to $FA so it doesn't matter that the byte was left on the stack.
5. Decrement 65, which was set in step 1 to the ON expression value from 0 to 255.
6. If 65 is not zero, branch to step 9.
7. When 65 finally becomes zero, the matching line number has been reached. Pull the character that stopped expression evaluation in step 1. This character must now be either the token for GOTO or the token for GOSUB.
8. JMP A/C7EF to Execute Current BASIC Statement to execute either GOTO or GOSUB using the last value retrieved in 14 and 15 as the line number. The ON processing is now complete.
9. JSR CHRGET to retrieve the next character in the line number.
10. JSR A/C96B to convert this line number to a two-byte line number in 14 and 15. If the line number is null—for example, if a double comma (,,) appears in the GOSUB/GOTO list—return with 14 and 15 equal to $0000. Return from A/C96B with the accumulator containing the character that stopped the conversion of the line number.
11. If the character in the accumulator is a comma, branch to step 5 to continue until 65 is decremented to zero.
12. If the character following the line number is not a comma, it should be either the $00 end-of-line byte or else a SYNTAX ERROR later occurs. In any case, if the character is not a comma, evaluation of the GOSUB/GOTO list is complete without getting to the line number entry that matches the ON expression value. Just PLA and RTS.

IF
A/C928-A/C93A and A/C940-A/C94A

Called by: Execute Current BASIC Statement for IF.

Operation:

1. JSR A/CD9E to evaluate the expression following the IF. If the expression evaluates to zero, the exponent of Floating Point Accumulator 1 (FAC1), 61, is zero. This state is

the false condition. The true condition is any nonzero value in FAC1. For example, 3 * 7 evaluates to true because it gives a nonzero result in FAC1. Usually the expression following IF is a comparison such as X>7. A comparison sets FAC1 to zero if the result of the comparison is false.

2. JSR CHRGOT to load the accumulator with the character that ended the evaluation of the expression.
3. Compare the accumulator to $89 to see if this character is the token for GOTO. If so, branch to step 5.
4. Check to see if this last character was the token for THEN, $A7. Load the accumulator with $A7 and JSR A/CEFF to check for the THEN token. Display SYNTAX ERROR if the token is not found. If the token is found, then also JSR CHRGET to retrieve the first nonspace character following the THEN token.
5. If either the GOTO token or the THEN token is found, see whether the IF expression was true or false. Load the accumulator from 61, the exponent for FAC1.
6. If 61 is nonzero, the expression was true, so branch to step 8.
7. If 61 is zero, the expression was false, so fall through to A/C93B to skip the remainder of this line by executing the code for REM.
8. A/C940: JSR CHRGOT to retrieve the first nonspace character following the THEN token or the GOTO token. [Note that if the GOTO token was found previously, (7A) still points to the GOTO token when step 8 is reached.]
9. If the character returned by CHRGOT is not a digit, branch to step 11. Such a branch would be made for IF 2=2 GOTO or IF 2=2 THEN X=5.
10. If the character returned by CHRGOT is a digit, this situation is caused by a statement such as IF 2=2 THEN 20. JMP A/C8A0 to execute GOTO.
11. For either the GOTO token or for THEN followed by a nonnumeric character, JMP A/C7ED to execute the command whose token is now in the accumulator. This value is one of the following: the token for GOTO; the token for a command following THEN; the first character of a variable following THEN for an implicit LET; the $00 end-of-line token following a THEN that has nothing else following on this line; or $3A for the colon following a

THEN that has nothing else following it in this statement. If either the $00 or $3A condition just mentioned is true, then the routine Execute the Next BASIC Statement will just pass control to the statement following the IF statement.

REM
A/C93B–A/C93F

Called by:
Execute Current BASIC Statement for REM; Fall through from IF if the expression following IF is false.

Operation:
1. JSR A/C909 to scan for the end of the line containing REM (or IF with false expression value), leaving the Y-register equal to the offset to the end of the line.
2. Branch to A/C8FB to add the Y-register to (7A) to move the text pointer to the end of this BASIC line.

DATA
A/C8F8–A/C8FA

Called by:
Execute the Current BASIC Statement for DATA; JMP at A/CBE7 INPUT; JSR at B/D3DB DEF; Fall through from RETURN.

Operation:
1. JSR A/C906 to scan for the end of the statement. Upon return the Y-register contains the number of characters from (7A) to the end of the line.
2. Fall through to A/C8FB to add the Y-register to (7A) to thus point (7A) to the next BASIC statement minus one. (7A) will be pointing to a $00 end-of-line byte or to a colon for the end-of-statement.

Add Y-register to Text Pointer (7A)
A/C8FB–A/C905

Called by:
Fall through from DATA; BEQ at A/C93E REM; JSR at A/CBF6 INPUT; JSR at A/CCD1 GET/INPUT/READ.

Operation:
Modify (7A), the text pointer, by adding the Y-register to (7A) and storing the result back into (7A).

Scan for Delimiter of Statement or Line A/C906–A/C927

Called by:
JSR at A/C75A FOR; JSR at A/C8F8 DATA; JSR at A/CBF3 INPUT; JSR at A/CCB8 READ; Alternate A/C909: JSR at A/C8A3 GOTO; JSR at A/C93B REM.

Operation:

1. For scanning for the end of a statement, load the X-register with $3A, the ASCII code for : and fall through to step 3.
2. A/C909: For scanning for the end of a line, load the X-register with $00 to indicate the end-of-line $00 byte.
3. Store the X-register in 07.
4. Load the Y-register with $00 and store in 08.
5. Swap the values in 07 and 08. Thus initially if searching for : end-of-statement, then 08 = $3A and 07 = $00.
6. Load the accumulator from (7A) indexed by the Y-register.
7. If the character just loaded is $00, branch to A/C905 to RTS since the end-of-line $00 byte has been read.
8. Compare the accumulator to 08.
9. If equal, the ending delimiter has been reached, so branch to A/C905 to RTS.
10. If not equal, increment the Y-register, which counts the offset to the end of the line.
11. Compare the accumulator to $22, the ASCII for quote. If not found, branch to step 6 to continue the search for the delimiter.
12. If $22, ASCII quote, is found, branch to step 5 to swap the delimiters. Thus a colon within quotes will not end the line since the colon has been swapped to 07. A second quote restores the colon to 08 to again function as a delimiter.

Consider the following example which shows the value the Y-register will have when this routine exits upon finding the delimiter.

Line Being Scanned:

8F	54	49	4E	41	00	equivalent value
REM	T	I	N	A		ASCII characters
04	05	06	07	08	09	Location (hex)

Initially, 7A = 04 at entry to this scan routine. Each time after a character is scanned and found not to be the delimiter, the Y-register is incremented. Thus the Y-register would be incremented for 8F, 54, 49, 4E, and 41, for a final result in the Y-register of 5. This Y-register value can then later be added to 7A to give a sum of 09, thus pointing to the final byte in this line (or final byte in this statement if 3A : is the delimiter).

GOSUB
A/C883-A/C896

Called by: Execute the Current BASIC Statement for GOSUB.

Operation:

1. JSR A/C3FB to see if enough stack space exists to push six bytes onto the stack (although only five bytes are pushed). If not, display OUT OF MEMORY error message and return to the Main BASIC Loop.
2. Push the following onto the stack:

   7B—high-order byte of text pointer (with (7A) pointing to the first nonblank value following the GOSUB token),

   7A—low-order byte of text pointer,

   3A—high-order byte of current BASIC line being executed (the line with this GOSUB statement),

   39—low-order byte of current BASIC line being executed,

   $8D—the GOSUB token to signify a GOSUB block is on the stack.
3. Fall through to A/C897 to perform GOTO for GOSUB/RUN with program flow transferring to the line number following the GOSUB. The line number has to be in the range 0–63999 or SYNTAX ERROR occurs. If the parameter is nonnumeric, a line number 0 must exist or the UNDEF'D STATEMENT error occurs.

RETURN
A/C8D2-A/C8F7

Called by: Execute the Current BASIC Statement RETURN.

Operation:

1. If any parameters follow RETURN, just RTS, which will cause a SYNTAX ERROR when it finds that the command has not processed all the following characters in a statement.
2. Store $FF into 4A to reset the FOR pointer to indicate that a GOSUB block is to be retrieved from the stack and that all FOR blocks are to be purged.
3. JSR A/C38A to scan the stack for any FOR entries. The scan begins at the stack location four bytes above the current stack pointer, thus skipping over the last two return addresses that are on the stack. The X-register is used as an index into the stack after being loaded with the stack pointer plus four. Each time a FOR is found, a comparison is made to see if the FOR block located contains the address of the corresponding FOR variable asked for. Setting 4A to $FF forces a condition where a match is never found. Thus all the FOR entries on the stack are bypassed with each bypass adding 18 to the X-register. Upon return the X-register points to where a GOSUB token should be found and the accumulator holds its value.
4. TXS to reset the stack pointer to where the GOSUB should be found. Everything above on the stack (all of the FOR entries and the two return addresses) are thus purged.
5. Compare the accumulator to $8D to see if this stack entry is the flag for a block of GOSUB information. If it is, branch to step 7.
6. If this entry is not $8D, display RETURN WITHOUT GOSUB error message and return to the Main BASIC Loop.
7. For a found GOSUB block, restore the block information so that execution can continue at the statement following the GOSUB. PLA to remove the 8D from the stack and discard.
8. Pull (39), the line number of the GOSUB statement, from the stack.
9. Pull (7A), the text pointer to the parameter following the GOSUB, from the stack.

10. Fall through the A/C8F8 to skip to the next statement by resetting (7A) to point to the end-of-statement or the end-of-line for GOSUB.

FOR
A/C742–A/C7AD

Called by: Execute Current BASIC Statement FOR.

Operation:

1. Load 10 with $80 to disallow integer or array variables for the FOR loop variable.
2. JSR A/C9A5 to perform LET to create or locate a floating point variable to serve as the FOR loop variable and return a pointer to the data area for this variable in (49). This LET processes the <floating point variable> = <arithmetic expression> section of the FOR. If an integer or array variable is specified, a SYNTAX ERROR results.
3. JSR A/C38A to see if the loop variable is already on the stack in a FOR block. If it is, return with Z=1 (BEQ condition) and (49) pointing to the data area for the loop variable found on the stack.
4. BNE to step 6 if the loop variable was not already active on the stack.
5. If the loop variable was already on the stack, add 16 to the X-register. The X-register returns from the search for FOR blocks pointing to the FOR token that was found. Adding 16 to the X-register value causes the X-register to point to the two bytes that contain the pointer to the end-of-statement byte of the FOR block. TXS to reset the stack pointer to this value.

   Thus, if you specify a FOR using the same variable as a previous FOR block did, all intervening FOR blocks are removed from the stack.

   1 byte—FOR token
   2 bytes—address of Loop Variable Data
   5 bytes—STEP value
   1 byte—sign of the STEP
   5 bytes—TO value
   2 bytes—current line number of FOR
   2 bytes—pointer to end-of-statement byte of FOR

6. Pull two bytes off the stack. If FOR was found in a block on the stack, pull off the pointer to the end-of-statement

of that FOR statement since new values will be provided for all the values in this FOR block. If FOR was not found, pull off the return address of this FOR routine, which normally would return to step 4 of Statement Execution Control. This return address has been removed if the FOR block for this loop variable was found.

7. JSR A/C3FB to see if 18 bytes of space exist on the stack (in addition to the standard reserved 62 bytes). If adequate space does not exist, display OUT OF MEMORY.
8. JSR A/C906 to determine the number of characters from the end of this FOR statement to its delimiter, returning with the Y-register equal to this offset.
9. Add the Y-register to (7A) to determine the pointer to the end-of-statement delimiter for this FOR statement. Push this two-byte pointer onto the stack, first pushing the low-order then the high-order. Notice that (7A) is not modified.
10. Push 3A and 39 onto the stack, the line number of this BASIC line containing the FOR statement.
11. Load the accumulator with $A4, the token for TO.
12. JSR A/CEFF to see if the current character pointed to by (7A) in the FOR statement is the token for TO. If not, display SYNTAX ERROR. If so, JMP CHRGET to retrieve the following character into the accumulator.
13. JSR A/CD8D to verify that the starting value of the loop variable is numeric. If it isn't, display TYPE MISMATCH.
14. JSR A/CD8A to evaluate the TO value, returning this value in FAC1. Note that this TO value is stored as a floating point value even if it is specified as an integer. Also verify that the TO value is numeric and display TYPE MISMATCH if it isn't.
15. Set the high-order bit of 62 to the same value as the sign bit of FAC1 to convert the value to the format of a floating point variable.
16. Point the accumulator and Y-register to A/C78B and store this pointer in (22) to provide a place to return to after the next step.
17. JMP A/CE43 to push FAC1 onto the stack, pushing 65, 64, 63, 62, and 61 in sequence. Return to step 18 by exiting with a JMP(0022).
18. Point the accumulator and Y-register to B/D9BC, which

contains the floating point variable for 1, the default step value.

19. JSR B/DBA2 to copy the floating point number at B/D9BC to FAC1.
20. JSR CHRGET to retrieve the next character in the FOR expression.
21. Compare this character to $A9, the token for STEP.
22. If the STEP token is found, JSR A/CD8A to evaluate the expression following the STEP token and return its value in FAC1. If the expression is nonnumeric, display TYPE MISMATCH. If the STEP token is not found, just use the default of 1 for the STEP size.
23. JSR B/DC2B to determine the sign of the STEP value. Return with the accumulator = $FF if it's a negative value or with the accumulator = $01 if it's a positive value.
24. JSR A/CE38 to remove the return address for A/CE38 from the stack and store in (22). Then push the following onto the stack: the sign of the step value. The step value from FAC1: 65, 64, 63, 62, 61. Exit by doing a JMP(0022) to return to step 25.
25. Push $81, the token for FOR, onto the stack. Thus a total of 18 bytes of information about this FOR have been pushed onto the stack.
26. Fall through to A/C7AE to execute the next BASIC statement following FOR. Thus the FOR loop is executed at least once.

NEXT
A/CD1E-A/CD89

Called by: Execute the Current BASIC Statement for NEXT.

(Jim Butterfield pointed out the following conditions concerning FOR-NEXT and GOSUB-RETURN in an article in *COMPUTE!* magazine, "FOR/NEXT GOSUB/RETURN and the Stack," in November 1981.)

- FOR loops and subroutines are not built until a RUN is done, because the FOR statement is not interpreted and executed until RUN.
- FOR loops remain open if they are not closed, even after a program finishes.
- Adding or editing a line clears all FOR loops.

- FOR loops and subroutines nest within each other.
- Executing RETURN for a subroutine closes all FOR loops within the subroutine.
- Closing a FOR loop closes all FOR loops nested inside.
- Executing NEXT closes all FOR loops nested inside.
- Jumping out of a FOR loop still leaves the loop open.
- Reopening a loop by using the same loop variable again in another FOR statement closes all the FOR loops that were opened since the original FOR loop was opened.

In these points, closing a loop refers to pulling its FOR block data from the stack and resetting the stack pointer to an area above this FOR block, thus deleting reference to it.

Operation:

1. If NEXT is followed by a $00 end-of-line byte or a colon (:) end-of-statement byte, branch to step 3 since no specific FOR variable is associated with this NEXT.
2. If NEXT is followed by a parameter, then JSR B/D08B to return with a SYNTAX ERROR if this parameter does not start with A–Z. If the variable does not already exist, create it. Return with a pointer to the data area for this variable in the accumulator and Y-register.

   Although NEXT allows references to string, integer, and array variables, FOR allows only floating point variables in the FOR loop variable. Thus a NEXT WITHOUT FOR occurs if you use a nonfloating point variable.
3. Store the accumulator and Y-register into (49), which points to data for a NEXT variable or contains $0000 if no variable for NEXT was specified.
4. JSR A/C38A to scan the stack for the FOR loop variable for this NEXT. If NEXT did not have a variable, just scan for the first FOR block. If a FOR block matching the requested FOR block is found (either a specific one or the first one), then return with Z=1 (BEQ condition) and the X-register as the index from 0101 to the start of the FOR block. Also return with (49) pointing to the data of this FOR loop variable.
5. If Z=0, the FOR block was not found, so display NEXT WITHOUT FOR error and return to the Main BASIC Loop.
6. If the FOR block was found, then TXS, thus setting the stack pointer to point to the location on the stack one byte previous to the FOR block.

7. Keep the following chart of the depth of entries in the stack in mind when reading the following steps:

Depth	Value
1	FOR token
2–3	Address of FOR loop variable
4–8	Floating point value of STEP
9	Sign of STEP
10–14	Floating point value of TO
15–16	Line number of FOR statement
17–18	Text pointer to end of FOR statement

Transfer the X-register to the accumulator and add 4, thus pointing to the first byte on the stack of the variable for STEP size. Push the accumulator onto the stack.

8. Add 6 to the accumulator to point to the first byte of the variable for the TO value and store the result in 24.
9. Pull the pointer to the STEP size from the stack. Set the Y-register to $01, setting the high-order byte of the pointer to the STEP size since the stack is in page 1.
10. JSR B/DBA2 to copy the five-byte floating point variable for the STEP size to FAC1, leaving the STEP size on the stack unchanged.
11. TSX to reset the X-register to point to the current stack pointer.
12. Load the accumulator from 0109,X to retrieve the sign of the STEP. The sign is at depth 9 in the FOR block.
13. Load the accumulator from 49 and the Y-register from 4A to retrieve the pointer to the data for the current FOR loop variable.
14. JSR B/D867 to copy the variable pointed to by (49) to Floating Point Accumulator 2 (FAC2), then add FAC2 to FAC1, which contains the STEP size. Thus the STEP size has been added to the loop variable.
15. JSR B/DBD0 to copy FAC1 to the memory located pointed to by (49), thus updating the value of this loop variable.
16. Set the Y-register to $01. JSR B/DC5D to compare the TO floating point number on the stack to FAC1, the updated FOR loop value. If the TO value = loop variable, the accumulator = 0; if TO < loop variable, the accumulator = 1; if TO > loop variable, the accumulator = $FF.
17. TSX to again set the stack pointer to just previous to this FOR block.

18. Subtract 0109,X, which is the sign value of the FOR step from the comparison of the TO value and the loop value.
19. If the result is zero, the loop is done, so branch to step 23. Consider these examples.

    FOR X = 1 to 3 STEP 1

    When X reaches 3, the comparison returns the accumulator of $00. Subtracting the positive sign of $00 from the accumulator leaves a result of $00.

    FOR X = 7 TO 8 STEP −1

    The loop variable is 6 after the step is added. Compare this to the TO value of 8. Since it is less, set the accumulator to $FF. Subtract the sign of the step, $FF, and thus leave a result of $00, indicating that the loop is done.
20. For a loop that is not finished:

    LDA 010F,X and STA 39
    LDA 0110,X and STA 3A

    Thus the line number of the FOR has been restored.
21. LDA 0112,X and STA 7A
    LDA 0111,X and STA 7B

    Thus the text pointer to the end of the FOR statement has been restored.
22. JMP A/C7AE to execute the statement following the FOR statement, thus continuing to execute this FOR loop which is not yet complete.
23. For a completed FOR loop, add 18 to the X-register value which points to the start of this FOR block just completed (minus one). TXS to reset the stack pointer past this FOR block since the FOR block is no longer needed for a closed FOR loop.
24. JSR CHRGOT to retrieve the last character scanned in the NEXT statement.
25. Is this character a comma?
26. If yes, a line such as NEXT X,Y has been detected. JSR CHRGET to retrieve the first character of this next parameter for NEXT. JSR A/CD24 to again execute NEXT for this parameter.
27. If this character is not a comma, NEXT is done. JMP A/C7AE to execute the next BASIC statement.

Find a Specific FOR Block in the Statement Execution and Program Flow Stack
A/C38A-A/C3B7

Called by:
JSR at A/C749 FOR; JSR at A/C8D8 RETURN; JSR at A/CD2B NEXT.

Operation:

1. TSX to thus set the X-register to the current stack pointer.
2. INX four times to move it past the last two return addresses. Consider the following diagram of the stack:

```
0100
0101
....
....
0160          Stack Pointer = $60
0161   4B     Address for C74C return
0162   C7
0163   E9     Address for C7EA return
0164   C7
0165   $81    FOR Token
(17 More Bytes of the FOR block)
```

   After the four INXs, the stack pointer would be 64 in the above example.
3. Load the accumulator from 0101,X, thus loading the accumulator from the stack location of the FOR token.
4. Compare the accumulator to $81, the FOR token. If not equal, then RTS.
5. Load the accumulator from 4A. RETURN sets 4A to $FF to force all consecutive FOR blocks found to be purged.
6. If 4A is nonzero, a specific FOR block is being looked for, so branch to step 8.
7. Load the pointer to the data address of the FOR loop variable from 0102,X and 0103,X and store into (49). This step is taken for any FOR block when NEXT is specified without any parameters.
8. Compare the accumulator to 0103,X. Of course this matches if step 7 was just executed. If it branched here from step 6, it's comparing to see if this FOR block variable data address matches the loop variable data address it is looking for. If not, branch to step 10.

9. If the high byte of the address matches, LDA 49 and compare to the low-byte address at 0102,X. If equal, branch to step 11.
10. Since this is not the FOR block wanted, add 18 to the X-register to point the X-register to the location for the next potential FOR block on the stack.
11. RTS with Z=1 if the FOR block requested was found and also leave (49) pointing to the data area of this FOR loop variable. The X-register contains the offset from the stack location 0101 to the FOR token.

PRINT

PRINT is a command which evaluates the parameters following the PRINT and then sends characters to the current output device, which is normally the screen, based on its evaluation of the parameters. If CMD has been used to associate a file number with a device other than the screen, that device receives the output from the PRINT.

The PRINT command can be analyzed based upon the various types of parameters it finds.

A string expression, whether a string variable or a literal string constant, needs no conversion. Just send each character in the string to the current output device. Special screen control characters can be embedded in the string to cause screen actions such as homing the cursor or changing the color of the screen. See the section on the Screen Editor in my book *The VIC and Commodore 64 Tool Kit: The Kernal* for the actions produced by these screen control characters.

A numeric expression is converted from a floating point value to the ASCII string representing this value with a space at the front of the string reserved for the sign. An extra space also follows a number when it is displayed as a string.

If no parameters follow PRINT, a blank line is displayed.

The TAB character for PRINT is a comma. This processing of the comma is one of the few areas where BASIC differs on the Commodore 64 and the VIC. This is due to the differing number of columns (40 on the Commodore 64, 22 on the VIC). Each comma encountered in the PRINT statement moves the cursor to the next tab position on the screen. For the VIC, only two tab positions exist—columns 0 and 11 (decimal). For the Commodore 64, columns 0, 10, 20, and 30 (decimal) are tab columns.

TAB(and SPC(are very similar. The argument specified by the TAB or SPC must be in the range 0–255. For TAB, send the number of characters needed to TAB from the current cursor position to the column specified in the argument. For SPC, the argument specifies the number of characters to send. If 13, the logical file for CMD, is nonzero, a space is the character sent. If 13 is zero, a cursor right is sent.

If semicolon (;) is detected, just continue printing the next parameter from where the previous one ended.

PRINT finishes when no more parameters are found in the PRINT statement.

This section describes the PRINT command and its various forms, along with other routines closely allied with PRINT. One routine at A/CB1E—to print a string followed by a carriage return—is called by many routines in addition to PRINT.

PRINT
A/CAA0-A/CAC9

Called by:
Execute Current BASIC Statement for PRINT; JMP at A/CA97 CMD; Alternate Entry A/CAA2 Handle Print for TAB, SPC, Comma, or Semicolon.

Operation:

1. The accumulator contains the first character of the parameter following the PRINT token as retrieved by CHRGET. If PRINT is not followed by any parameters, branch (BEQ) to A/CAD7 to send a carriage return and a linefeed. (A linefeed is sent only if 13, the current I/O device for CMD, is > 127.)
2. A/CAA2: Fall through from step 1 on BNE or come here after handling the last TAB, SPC, comma, or semicolon. BEQ A/CAE7 to RTS if the end of the parameters of the PRINT statement have been reached.
3. Compare the accumulator to $A3, the token for TAB(. If found, branch to A/CAF8 to Handle Print for TAB, SPC, comma, or semicolon.
4. Compare the accumulator to $A6, the token for SPC(. Clear the carry after the comparison. If found, branch to A/CAF8 with carry clear to Handle Print for TAB, SPC, comma, or semicolon.
5. Compare the accumulator to $3B, the ASCII character for semicolon (;). If found, branch to A/CB13 to Handle Print for TAB, SPC, comma, or semicolon.
6. If it's none of the above characters, then JSR A/CD9E to evaluate the expression the text pointer now points to. Upon return, 0D is set to $FF if it's a string expression or to $00 if it's a numeric expression. If it's a string ex-

pression, (64) holds a pointer to the descriptor for this string. Since this string expression is also the last one that has been evaluated and it has not been assigned to a string variable, the string descriptor for it is located in the temporary string descriptor stack. The string itself may be either in the program text area or in the active string area. If it's a numeric expression, Floating Point Accumulator 1 (FAC1) holds its value.

7. Test 0D to see which type of expression this is. If it's a string expression, branch to A/CA9A to print this string. For a numeric string, continue with step 8.
8. JSR B/DDDD to convert the number in FAC1 to the ASCII character string representation of this number, leaving the string at 0100 and a pointer to the string in the accumulator and Y-register. Accumulator = $00 and Y-register = $01.
9. JSR B/D487 to create a string descriptor for this string in the temporary string descriptor stack, with (64) pointing to this string descriptor.
10. JSR A/CB21 to remove the string descriptor from the temporary stack. Leave (22) pointing to the string, and the accumulator = length of the string.

    After getting the pointer to the string in (22) and its length in the accumulator, A/CB21 then sends each character in the string to the current output device using the Kernal CHROUT routine.
11. JSR A/CB3B to send either a cursor right (if the screen) or a space (other devices) to the current output device.
12. Branch to step 2 of Print a String from String Expression of a Print Statement to check if the end of the PRINT statement has been reached. This check is done by JSR CHRGOT to see if the last character is the end-of-statement colon or the end-of-line $00. If either condition is true, the PRINT is done. Otherwise, continue at A/CAA0 with another PRINT.

Print a String from String Expression of a PRINT Statement
A/CA9A–A/CA9F

Called by:
BMI at A/CABA PRINT; Alternate A/CA9D: BNE at A/CAC8 PRINT.

When strings are sent to the screen, ASCII characters that represent screen control characters can then take effect. These screen control characters include characters such as HOME and CLR.

Operation:

1. JSR A/CB21 to remove the string descriptor from the temporary string descriptor stack, leaving (22) pointing to the string and accumulator containing its length. The string descriptor removed was created when the string expression in the PRINT statement was evaluated.

   Then A/CB21 uses the pointer in (22) and the length in the accumulator to send each character in the string to the current output device using the Kernal CHROUT routine.
2. A/CA9D: JSR CHRGOT to see if the final character retrieved by CHRGET following the string or numeric expression is the end-of-line $00 byte or the end-of-statement colon. If either is true, Z=1 (BEQ condition) is set.
3. Fall through to step 1 of PRINT at A/CAA0.

Store $00 End-of-Line Byte in BASIC Input Buffer A/CACA-A/CAD6

Called by:

JMP at A/C576 Receive Input from Device and Fill BASIC Buffer.

Operation:

1. This routine is called whenever a carriage return has been received while filling the BASIC input buffer at 0200. The carriage return signifies end-of-line.
2. To signify end-of-line of the statement in the BASIC input buffer, a $00 is stored at the current location in the buffer indexed by the X-register.
3. Load the X-register to $FF and the Y-register to $01 to point to the beginning of the BASIC input buffer (minus one).
4. If 13, the I/O device for CMD, is zero, then RTS. Otherwise, fall through to A/CAD7 to send a carriage return and possibly a linefeed to this device.

Send Carriage Return (and Possibly Linefeed) to Current I/O Device A/CAD7-A/CAE7

Called by:
BEQ at A/CAA0 PRINT with no parameters; JSR at A/C44E Error Message Handler; JSR at A/C6D4 LIST; Fall through from Store $00 End-of-line Byte in BASIC Input Buffer; Alternate A/CAE5: JSR at A/CB35 Print String.

Operation:

1. Load the accumulator with $0D, the ASCII value for RETURN.
2. JSR A/CB47 to send the RETURN to the current output device.
3. If 13, the I/O CMD device, is >= 128, load the accumulator with $0A, ASCII linefeed, and JSR A/CB47 to send the linefeed to the current output device.
4. A/CAE5: EOR $FF.
5. RTS.

Handle TAB (Comma) for PRINT A/CAE8-A/CAF7

Called by: BEQ at A/CAAF PRINT.

On the VIC, commas in a PRINT statement cause a TAB to column 0 or 11, and on the Commodore 64 commas cause a TAB to column 0, 10, 20 or 30, depending on the current location of the cursor when the comma is encountered.

Remember that numbers are printed with a space for the sign.

Operation:

1. SEC.
2. JSR FFF0 to the Kernal PLOT routine to read the current location of the cursor, returning the column number of the cursor in the Y-register.
3. Transfer the Y-register to the accumulator.
4. Subtract $0A (Commodore 64) or $0B (VIC) from the accumulator.
5. If the carry is still set after the subtraction, branch to step 4 to again subtract.

6. EOR $FF and ADC $01 to convert the number from the two's complement negative value back to a positive value. This positive value is the number of spaces or cursor rights to insert between the cursor position and the next TAB position.
7. If the accumulator is nonzero, spaces need to be inserted. The accumulator is always nonzero since if you are already at the tab position, you need to move to the next one. The accumulator is either 1–10 (Commodore 64) or 1–11 (VIC). Branch to A/CB0E to insert the spaces.

Handle PRINT for TAB, SPC, Comma, Semicolon A/CAF8-A/CB1D

Called by:
BEQ at A/CAA6, A/CAAB PRINT; Alternate A/CB13 BCC at A/CB0C PRINT; Alternate A/CB0E Handle Comma (TAB).

Operation:

1. Push status (carry set for TAB or carry clear for SPC) onto the stack.
2. SEC to prepare to read cursor.
3. JSR FFF0 to the Kernal PLOT routine to return the current cursor column location in the Y-register.
4. STY 09.
5. JSR B/D79B to evaluate the next expression following the TAB(or SPC(. If the expression does not evaluate to a number, display TYPE MISMATCH. If a number is not in the range of 0–255, display ILLEGAL QUANTITY. Return with the accumulator containing the character past this expression returned by CHRGET. The expression 0–255 is stored as an integer number in 64 and 65 and also in the X-register.
6. Compare the accumulator to $29, the ASCII character for). This character should have been returned by CHRGET as the character following the numeric expression. If not, then display SYNTAX ERROR.
7. Pull the status stored in step 1.
8. If carry is clear, branch to step 13. This branch occurs for SPC.
9. For TAB(retrieve the X-register which contains the argument for TAB. TAB determines the TAB column using the leftmost column as column zero.

10. Subtract 09, the current character position of the cursor.
11. BCC to step 16 if the current cursor position is greater than the TAB value. For example, PRINT TAB(5)"HI"TAB(5)"HI" displays five spaces then HIHI. The second TAB(5) finds the cursor past the TAB column and thus branches to step 16.
12. A/CB0E: Transfer the accumulator to the X-register. This is the entry point for a comma, with the accumulator containing the number of spaces or cursor rights to be inserted. If falling through from step 11, the accumulator now holds the number of spaces or cursor rights needed between the cursor location and the location specified for TAB.
13. INX.
14. DEX with the X-register containing the number of spaces that need to be inserted.
15. If the X-register is not zero, branch to step 18.
16. A/CB13: Branch here to handle a semicolon. Also fall through from step 15 if no spaces need to be inserted or if you're all done inserting spaces for TAB, SPC, or comma.

    JSR CHRGET to retrieve the next character in the PRINT statement.
17. JMP A/CAA2 to step 2 of PRINT to see if there's anything else left to print.
18. Branch here from step 15 if it's TAB, SPC, or comma and you need to insert spaces or cursor rights. JSR A/CB3B to send a space (if any device except the screen) or display a cursor right (if the screen).
19. Branch to step 14 to see if another space is needed.

PRINT String and Carriage Return A/CB1E-A/CB3A

Called by:

JSR at A/C469 Display ERROR; JSR at A/C478 Display READY; JSR at A/CB6F Display REDO FROM START; JSR at A/CCF8 Display EXTRA INPUT IGNORED; JSR at B/DDDA Decimal Output Routine; JSR at E1AF Display READY after Load; JSR at E191 Display OK for Load; JSR at E42D/E40F Display Version of BASIC; JSR at E441/E423 Display BYTES FREE; Alternate A/CB21: JSR at A/CA9A Print a String Expression of PRINT; JSR at A/CBCB Display Prompt for Input;

JSR at A/CAC2 Print a Number That Has Been Converted to a Numeric String.

Operation:

1. JSR B/D487 to determine the length of the string to be printed. The string is ended by a quote or the $00 end-of-line byte. Also, allocate memory for this string, and copy the string to the memory area allocated. Store a descriptor for the string in the temporary string descriptor stack.
2. A/CB21: JSR B/D6A6 to remove the string descriptor for this string from the temporary string descriptor stack since the last descriptor in that stack refers to this string just processed. Also, if the string is located at the bottom of the active strings, move the pointer to the active strings above this string.

   Upon exit from B/D6A6, (22) contains a pointer to the string and the accumulator holds the length of the string.
3. A/CB24: Using (22) as a pointer to the string and the accumulator as its length, send each character in the string to the current output device by doing a JSR A/CB47, which does a JSR to E10C/E109. At E10C/E109, JSR to the Kernal CHROUT routine. This routine sends the character to the current output device. The default output device is the screen unless specifically changed. Once all characters have been sent, exit this routine.
4. After each character is sent in step 3 see if the character just sent was $0D, the ASCII code for RETURN. If it was, JSR A/CAE5 to EOR $FF, thus converting $0D to $F2. Then continue with the next character in step 3.

Send Space, Cursor Right, or ? to Current Output Device
A/CB3B–A/CB4C

Called by:
JSR at A/CAC5 PRINT; JSR at A/CC00 INPUT; JSR at A/CB19 PRINT for TAB, SPC, Comma, Semicolon; Alternate A/CB45: JSR at A/CC47 Display ? for Input; JSR at A/CBFD Display ? for Input; JSR at A/C451 Error Message Control; Alternate A/CB47 JSR at A/CB2D Print String; JSR at A/CAE2 Send Return; JSR at A/CAD9 Send Linefeed; JSR at A/C73D List Detokenized Keyword; JSR at A/C45B Error Message Display; JSR at A/C6F3 LIST.

Operation:

1. If 13, the file number for CMD, is zero, branch to step 3.
2. If 13 is not zero, load the accumulator with $20, ASCII space. Use a BIT to fall through to step 5.
3. A/CB42: Load the accumulator with $1D, ASCII cursor right, and use a BIT to fall through to step 5.
4. A/CB45: Load the accumulator with $3F, ASCII question mark.
5. A/CB47: JSR E10C/E109 to send this character in the accumulator to the current output device.
6. AND the accumulator with $FF.
7. RTS.

LIST

LIST sends each character in a stored BASIC program to the current output device. Normally the current output device is the screen, but it can be other devices such as the printer, disk, or tape.

The line number(s) you specify as the start and end of the list do not have to exist in the BASIC program. If the starting line number does not exist, the next higher numbered line in the program is used as the starting location. If the ending line number does not exist, then whenever a line is to be listed that has a higher line number than the specified end, the LIST stops.

LIST without any parameters lists the whole program by starting the list from the program text address of the first line number and continuing till all lines have been listed. LIST with no parameters sets the ending line number as 65535, which is higher than any possible actual line number.

The various other formats for LIST are explained in the detailed description.

A few peculiarities about LIST deserve mention.

If you have a LIST command executed from within a program, the program stops executing after the LIST statement. However, if you try to CONT in this situation, only the LIST statement is executed.

If a break occurs in a program and then you do a LIST, CONT will resume program execution with the statement following the one that caused the break.

Quotes around the ASCII characters that represent tokens prevent the tokens from being listed in detokenized format.

REM lists any tokens following it in their detokenized format unless the tokens are enclosed in quotes. Applications using this feature and including screen control commands in the REM statement have been used to produce programs that don't LIST normally.

Tokens are converted to their detokenized format (unless enclosed by quotes) by a separate routine that is reached by using a jump vector at (0306). If you have added new tokens to BASIC you will probably want to add the capability to list these tokens in detokenized format by changing the vector in (0306) to point to your routine.

Normally, a LISTed program whizzes by so fast you have to be a speed reader to read it. The CTRL key slows down the listing. Nothing in the LIST command itself checks for the CTRL key. The scrolling routine checks for the CTRL key and pauses the printing if the CTRL key is detected.

LIST
A/C69C–A/C716

Called by: Execute Current BASIC Statement for LIST.

Operation:

1. If LIST is followed by a number, such as LIST 25, branch to step 4.
2. If LIST is not followed by anything, also branch to step 4.
3. For a parameter following LIST that is not a number, compare the accumulator to $AB, the token for –. If not found, then RTS to produce SYNTAX ERROR. Such an error is caused by LIST X. If found, continue with step 4.
4. JSR A/C96B to convert the ASCII line number for LIST number or LIST– to a two-byte integer in 14 and 15. The number must be in the range 0–63999 or a SYNTAX ERROR occurs. For – the number left in 14 and 15 is $0000.
5. JSR A/C613 to search the tokenized BASIC program for this line number in 14 and 15. This routine sets (5F) to: the address of the line number if a match is found; if a higher-numbered line is found, the address of the higher-numbered line; if no match is found, the address of the final line with $0000 link field that signifies the end of the program.
6. JSR CHRGOT to see if this line number was followed by anything else on the line. Return with the accumulator containing the last character, which is – if that was the first thing following LIST.
7. If the accumulator is zero, indicating the end-of-line byte was read, branch to step 13 as this LIST contains no other parameters.
8. Compare the accumulator to $AB, the token for –.
9. If not equal, RTS to produce a SYNTAX ERROR. Such an error would be caused by LIST 25X.
10. JSR CHRGET to retrieve the first character following the –.

11. JSR A/C96B to convert the line number following –, then 14 and 15 are $0000. Upon exit, the accumulator contains the character that stopped conversion of the line number. LIST 1–X or LIST 1–XXX works.
12. If the accumulator is not zero, RTS to display SYNTAX ERROR. Such an error could be caused by LIST 25–73X.
13. This step is reached for any of the valid LIST formats: LIST, LIST–, LIST number –, LIST– number, LIST number – number, LIST number – letter(s).
14. If 14 and 15 are both zero now, then no line number followed the –, or a LIST with no parameters was specified.

    In this case set 14 and 15 to $FFFF, a line number of 65535. This sets the high range of the list. Since the highest valid line number is 63999, all lines are displayed from the specified start of LIST number or from the start of the program if LIST with no parameters.
15. Now enter a loop that lists all the statements from the starting address, (5F), through the ending line number, (14).
16. Load the Y-register with $01.
17. Store the Y-register in 0F, which is used to indicate whether currently listing from within a quote. If so, detokenization is suppressed.
18. Load the accumulator from (5F),Y. This picks up the high-order byte of the link address of the line to be listed.
19. If the accumulator is zero, the final line of the program text area has been reached, so exit.
20. JSR A/C82C to test for the STOP key. If detected, stop the list and display the BREAK message. Display READY and return to the Main BASIC Loop.
21. JSR A/CAD7 to send a RETURN to the current output device. If the file number is > 127, also send a linefeed.
22. Now see if the end of the list has been reached by comparing the line number from this line addressed by (5F) to the ending line number in (14). The line number for this line addressed by (5F) is loaded into the accumulator and X-register. The Y-register is incremented twice during this step.
23. If the accumulator and X-register are greater than or equal to (14), the end of the list has been reached; it's time to exit.
24. Store the Y-register in 49 to save the index into the statement being listed.

25. JSR B/DDCD to convert the line number in the accumulator and the X-register first to a floating point number in FAC1 and then into an ASCII string at 0100. Then send each character in the ASCII string to the current output device.
26. Load the accumulator with $20, the ASCII space, since a space is displayed in a listing after a line number.
27. A/C6EF: Fall through from step 26. Also branch here from List Detokenized Keywords after a keyword has been listed.
28. Load the Y-register from 49 to restore the index into the statement being listed.
29. AND the accumulator with $7F to remove the high-order bit of any characters to be listed. This removes the high-order bit from the last character of a keyword from the keyword table.
30. A/C6F3: Fall through from step 29. Also branch here from List Detokenized Keyword if character does not have high-order bit on, if $FF for PI, or if quote mode is on.

    JSR A/CB47 to send the character to the current output device.
31. Compare the character just sent to $22, ASCII quote. If found, reverse 0F by doing an Exclusive OR with $FF. 0F is used to determine whether you're within quote mode when you're listing tokens.
32. Increment the Y-register to move to the next character in the line being listed.
33. If the Y-register is zero, exit. The maximum size of the line to be listed appears to be 256 (including the line number and link field). With detokenization, a longer line may actually appear on the screen.
34. Load the accumulator from (5F),Y to get the next character in the line being listed.
35. If this character is not the end-of-line $00 byte, branch to Handle Listing for Tokens to see if it is a token to be listed in its detokenized format.
36. If $00 end-of-line byte is loaded, load the link address from this current line addressed by (5F). Store this link address into (5F) to prepare to list the next line.
37. If the high-order link byte is zero, the end of the program has been reached. In this case, exit.
38. If the high-order link byte is not zero, branch to step 16 to list the next line.

Handle Listing for Tokens A/C717-AC741

Called by: BNE at A/C705.

Operation:

1. JMP (0306), which contains a default vector for A/C71A, step 2 of this routine. If you have added tokens to BASIC, you would want to intercept this list for tokens.
2. If the high-order bit of this character is 0, this can't be a token, so branch to A/C6F3, step 30 of LIST.
3. Is the character $FF? If so, this is the ASCII code for PI. Branch to A/C6F3, step 30 of LIST.
4. If the high-order bit of 0F is 1, the quote mode is on. Branch to A/C6F3 if the quote mode is on, since tokens are not detokenized within quotes.
5. Subtract $80 from the token to reduce it to a value from $00–$7F. Transfer this value to the X-register.
6. Store the Y-register in 49 to save the index into the line being listed.
7. Load the Y-register with $FF to correctly initialize the index into the keyword table.
8. DEX.
9. If the X-register is now zero, the entry in the keyword table matching this token has been found. Branch to step 14.
10. INY.
11. Load the accumulator from A/C09E,Y to get the next character of the keyword currently being examined.
12. If the high-order bit of this character is 0, the end of the keyword has not been reached. Branch to step 10.
13. When the character with the high-order bit is detected, the end of this keyword has been reached. Branch to step 8 to see if the token corresponds to the next keyword.
14. When the keyword table entry has been found, the Y-register points to the last character of the previous keyword.
15. INY to point to the next character of this keyword.
16. Load the accumulator from A/C09E,Y to retrieve the next character in the keyword.
17. If the high-order bit is on, the last character of the keyword has been found. When this occurs, branch to A/C6EF to display this last character of the keyword after removing its high-order bit.

18. For all other characters in the keyword, just JSR A/CB47 to send this character to the current output device.
19. Branch to step 14 to handle the next character in the keyword.

DEF and FN

DEF is used to define a function which returns a numeric value. FN can then later reference this function to evaluate the expression in DEF, possibly using the argument passed from FN.

DEF and FN are both keywords and both have token values. The detection of the DEF FN works by first detecting the DEF and then the FN. DEF is called from Statement Execution and is a command, while FN is called from Evaluate the Operand of an Expression.

String functions cannot be defined.

Here's an example that I'll use to point out a few things about DEF and FN. Enter and run this program:

```
10 A = 3
20 DEF FN XY(A) = 7*A
30 PRINT FN XY(6)
40 PRINT A
```

The computer responds with:

```
42
3
```

In the DEF statement the XY is the name of the function. The function name cannot be a string, integer, or array type name. The *A* is the name of the dependent variable of the function. This variable can appear on the right side of the = in the DEF statement.

As the above example shows, the dependent variable *A* was previously used in the program, which is fine. Also, the dependent variable value is left intact after a DEF or a FN statement, as the PRINT A that displayed 3 illustrated.

Here's how the DEF and FN work. First, DEF must be located in the program previous to any FN reference to it.

The DEF statement creates a function descriptor, which is stored in the area for scalar variables since DEF uses the routine to create variables. This function descriptor has the following format:

Function Name		Text Pointer to First Char Past "=" in DEF		Pointer to Data Area of Dependent Variable		Dummy Value First Non-Space Char Past "="
High-Order Bit On	High-Order Bit Off	Low	High	Low	High	

Notice that the fifth and sixth bytes contain a pointer to the data area of the dependent variable, which is found just above the variable name in that variable descriptor.

When the FN statement references a function, the variables area is searched for a function descriptor by that name. If the function descriptor isn't found, then UNDEF'D FUNCTION occurs. If the function descriptor is found the FN is evaluated through the following steps. Let's again refer to the short program:

```
20 DEF FN XY(A) = 7 * A
30 PRINT FN XY(6)
```

The function descriptor is located, and a pointer to the data area for the dependent variable is stored in (47). The five bytes of data for the dependent variable are then pushed onto the stack. The (6) is then evaluated with the result left in Floating Point Accumulator 1 (FAC1). FAC1 is then copied to the data area for the dependent variable pointed to by (47) so that A is now equal to 6.

Now reset (7A), the text pointer, from the function descriptor; (7A) now points to the 7 in the DEF statement. Call Expression Evaluation to evaluate the 7 * A, using the value for A of 6. This evaluates to 42, with the result left in FAC1. The dependent variable's data is then pulled from the stack and copied back into the data area for the dependent variable, leaving it unchanged upon exit.

The FN evaluation is now complete, with the result in FAC1 of 42. In this case, the PRINT finds a numeric value in FAC1, converts the numeric value to an ASCII string, and prints the string.

DEF
B/D3B3–B/D3E0

Called by: Execute Current BASIC Statement for DEF FN.

Operation:

1. JSR B/D3E1 to verify that the last character retrieved by CHRGET was the token for FN, $A5. If so, then also determine the name of this function. Either locate or create a function descriptor with this name. If the variable for the function name is specified as an integer variable or an array variable, display SYNTAX ERROR. If the variable is a string variable, display TYPE MISMATCH. Return with a pointer to the data area for this function descriptor in (4E) and also in the accumulator and Y-register.
2. JSR B/D3A6 to display ILLEGAL DIRECT if trying to do DEF from direct mode.
3. JSR A/CEFA to check for a (following the function name and display a SYNTAX ERROR if not found.
4. Store $80 in 10 to disallow integer or array variables in the dependent variable of the function.
5. JSR B/D08B to scan the dependent variable following the (. If an integer or array variable, display SYNTAX ERROR. Return with (47), the accumulator and Y-register all pointing to the data area for this dependent variable.
6. JSR A/CD8D to display TYPE MISMATCH if a string variable.
7. JSR A/CEF7 to check for a) following the dependent variable and display SYNTAX ERROR if not found.
8. JSR A/CEFF to check for $B2, the token for =, following the). If not found, display SYNTAX ERROR. If = token is found, push the first character past the = onto the stack. In the various syntax checks just described, spaces can appear between the (, the dependent variable, the), the = etc., since CHRGET skips over spaces.
9. Push 48 and 47 onto the stack, saving the pointer to the data area for this dependent variable.
10. Push 7A and 7B onto the stack, which point to the first nonspace character following the = token in the DEF statement.
11. JSR A/C8F8 to move the text pointer (7A) to the end-of-statement byte of this DEF statement, which is either a colon or the $00 end-of-line byte.

12. JMP B/D44F to store the stacked values for 7A, 7B, 47, 48, and the character following the = token, at the location pointed to by (4E). (4E) points to the data area for this function descriptor.

Check Syntax for FN Descriptor and Return Pointer to Its Data Area
B/D3E1-B/D3F3

Called by: JSR at B/D3B3 DEF; JSR at B/D3F4 FN.

Operation:

1. (7A) is pointing to the first nonspace character following a keyword during statement execution or to the token for FN after scanning for an operand during Expression Evaluation.

   Load the accumulator with $A5, the token for FN.
2. JSR A/CEFF to see if the last character retrieved was the token for FN. If not, display SYNTAX ERROR. For example, DEF x causes this SYNTAX ERROR.

   Also do a CHRGET to load the first character of the function name into the accumulator.
3. ORA $80 and store in 10 to disallow integer or array variables to be specified as the variable name of the function descriptor. This ORA also turns on the high-order bit of the first byte of the function name. The accumulator is stored in 45 in the next step as the first letter of the function name.
4. JSR B/D092 to scan the function name, leaving any second character of the name stored in 46 with the high-order bit off. If the function name is specified as an integer or array variable, display SYNTAX ERROR.

   Return with the accumulator and Y-register pointing to the data area for this function descriptor.
5. Store the accumulator in 4E and the Y-register in 4F to set (4E) as a pointer to the data area of the function descriptor.
6. JMP A/CD8D to display TYPE MISMATCH if the function name was given as a string variable.

FN
B/D3F4-B/D464

Called by:

JMP at B/D3F4 Evaluate Operand of Expression; Alternate B/D44F DEF.

Operation:

1. JSR B/D3E1 to get the pointer to this function into (4E). The syntax check for a FN token is superfluous since the FN token had to be detected for Evaluate Operand to reach here. The syntax check does retrieve the first character of the function name, though, which is essential. If the function name is specified as an integer or array variable, display SYNTAX ERROR. If the function name is specified as a string variable, display TYPE MISMATCH.
2. Push 4F and 4E, the pointer to the data area for the function descriptor, onto the stack.
3. JSR A/CEF1 to check for a (. If it's not found, display SYNTAX ERROR.

   If (is found, evaluate the expression within the parentheses, leaving the result in FAC1. If it's a string expression, set 0D to $FF.

   Check for a) following the expression and display SYNTAX ERROR if not found.
4. JSR A/CD8D to see if this expression within parentheses is a string expression, and if so, display TYPE MISMATCH.
5. Restore 4E and 4F from the stack.
6. Copy (4E),2 and (4E),3 to 47 and 48 and also to the X-register and the accumulator. (47) = pointer to the data area of the dependent variable for this function descriptor.
7. If (4E),3 which was stored in 48 was zero, this function has not been defined yet. This is known because any function descriptor that is defined creates a dependent variable. The dependent variable cannot be located in page 0 because it must be located in the normal variable area. Thus if the high-order byte points to page 0, the dependent variable has not yet been created. If this is true, display UNDEF'D FUNCTION and return to the Main BASIC Loop.
8. If the function has already been defined, a dependent variable already exists. So that we won't destroy the data of this dependent variable, push all five bytes of the data area pointed to by (47) onto the stack.
9. Load the Y-register from 48 and the X-register from 47. (47) = pointer to the data area of the dependent variable.
10. JSR B/DBD4 to copy FAC1 to the data area of the dependent variable. FAC1 contains the expression that was eval-

uated from inside the parentheses. For example, if the statement is Y = FN X(3) then FAC1 = 3. The Y-register is zero upon return.

11. Push 7B and 7A onto the stack.
12. Load the accumulator from (4E),0 and store into 7A. Load the accumulator from (4E),1 and store into 7B. Thus (7A) now points to the first character of the expression following = in the DEF statement.
13. Push 48 and 47, the pointer to the data area for the dependent variable, onto the stack.
14. JSR A/CD8A to evaluate the expression following the = in the DEF statement. Note that if the dependent variable appears on the right side of the expression, the value that it retrieves is the value stored there in step 10. This value in the dependent variable is the value of the expression within parentheses for the FN.

    The result of this evaluation is left in FAC1.

    If the expression evaluates to a string, display TYPE MISMATCH.
15. Pull two bytes off the stack. What is being pulled here are the two bytes pushed in step 13, which were the pointer to (47), the dependent variable data area.

    However, a switch occurs here when these two bytes are restored to (4E). Thus (4E), which is used to point to the data area for the function descriptor, actually points to the data area for the dependent variable now.
16. JSR CHRGOT to retrieve the last character called after the DEF statement. If not the $00 end-of-line byte or the colon end-of-statement byte, display SYNTAX ERROR.
17. B/D44F: This routine does a restoration from the stack to the area pointed to by (4E). However, depending on whether this step is reached from DEF or from FN, the (4E) points either to the data area for the function descriptor or the data area for the dependent variable.

    For FN: Pull the five bytes of the dependent variable data from the stack and restore to (4E),0 through (4E),5. Thus the data area for the dependent variable is now back intact. The (4E) pointer is no longer needed and will be rebuilt the next time FN or DEF appears in the program.

    For DEF: If step 17 is reached by a JMP from DEF, then (4E) points to the data area for the function descriptor.

18. RTS. The value of the FN upon exit is found in FAC1, being stored there in step 14. The remaining steps were just cleanup to get the dependent variable back to its original value.

Function Invocation

During Expression Evaluation the routine to Evaluate Operand or Handle Monadic Operator jumps to the Function Invocation routine if it finds token values of $B4 or greater when trying to evaluate an operand. These token values of $B4 or greater represent functions.

However, the Function Invocation routine makes no check to see what is the highest token value that is a valid function. Normally the values $CC–$FE do not serve as tokens unless you specifically create wedges that allow these values to serve as tokens. The highest normal token value for a valid function is $CA, the token for MID$. One higher token value does exist—$CB for GO. This GO token is normally used to allow you to write GO TO with spaces between the GO and the TO and to call the GOTO routine when this sequence is found.

If the GO token is found in the normal location for an operand during expression evaluation and meets the criteria of having two arguments in parentheses with the first a string value and the second a numeric value in the range 0–255, then function invocation actually tries to execute GO.

How is GO executed when there is no GO routine? Well, it takes the GO token and multiplies it by two to get an index into the table of functions to retrieve the address of the psuedoGO routine. However, the address it picks up is $6979, which is the priority and the low-order address byte of the + operator that follows the valid table of function addresses.

Thus GO goes to $6979. This could easily hang your system unless you have a machine language program at 6979 that does whatever it wants and finally returns control to BASIC. Finally, it seems this works only from program mode. From direct mode it appears to cause a warm start of BASIC.

Examples of how to cause this jump to $6979 are:

```
10 A = 3 + GO("TEST",2)
```

or

```
10 A = GO("", 5)
```

Fortunately for those of you who want to add new tokens above $CB to the language, you do not need to determine the address the function invocation routine picks and put your machine language program for the new function at that location. However, you could if you wanted to. The normal method of picking off the tokens from $CC–$FE is to change the vector at (030A) that has the address of A/CE86 to jump to the Evaluate Operand or Handle Monadic Operator. The vector can be changed to jump to your routine which checks for the tokens $CC–$FE.

When evaluating the recognized functions, a test is made to see whether this function is from a token with a value equivalent to $C8 or higher. These $C8 and higher tokens represent the functions that take two or more parameters in their argument—LEFT$, RIGHT$, MID$. For these three functions, the first argument must be a string expression. The string expression is evaluated, and a pointer to its descriptor is pushed onto the stack. The second parameter is then evaluated, and it must evaluate to a numeric value from 0 to 255. This second parameter is also pushed onto the stack.

For all of the functions from $B4 through $C7, only one parameter is needed in the argument. This parameter is evaluated and the result left in FAC1. If it's a numeric value, FAC1 holds the resulting floating point value. If it's a string expression, (64) points to the string descriptor for this string.

After evaluating the parameters for the function, the address of the function is retrieved from the table of function addresses and moved to (55). Then a JSR 0054 is done. At location 54 is a JMP opcode which JMPs to the address of the function.

The numeric functions then return for one final check to see if the result of the function is numeric and to display a TYPE MISMATCH if the expression is a string.

The string functions don't return to function invocation for this type check. Rather they do their own type checking for string expression values.

Following is the table of functions, their tokens, and the addresses of their routines.

Function	Token	Routine Address
SGN	B4	B/DC39
INT	B5	B/DCCC
ABS	B6	B/DC58
USR	B7	0310/0000
FRE	B8	B/D37D
POS	B9	B/D39E
SQR	BA	B/DF71
RND	BB	E097/E094
LOG	BC	B/D9EA
EXP	BD	B/DFED
COS	BE	E264/E261
SIN	BF	E26B/E268
TAN	C0	E2B4/E2B1
ATN	C1	E30E/E30B
PEEK	C2	B/D80D
LEN	C3	B/D77C
STR$	C4	B/D465
VAL	C5	B/D7AD
ASC	C6	B/D78B
CHR$	C7	B/D6EC
LEFT$	C8	B/D700
RIGHT$	C9	B/D72C
MID$	CA	B/D737

Function Invocation A/CFA7–A/CFE5

Called by:
JMP at A/CEEE Evaluate Operand or Handle Monadic Operator During Expression Evaluation.

Operation:

1. At entry the accumulator contains the token of the function that is to be invoked.

   ASL to multiply the function by 2 to produce an index into the table of functions.

   The possible functions and the indexes (in hex) produced after this ASL are: SGN 68, INT 6A, ABS 6C, USR 6E, FRE 70, POS 72, SQR 74, RND 76, LOG 78, EXP 7A, COS 7C, SIN 7E, TAN 80, ATN 82, PEEK 84, LEN 86, STR$ 88, VAL 8A, ASC 8C, CHR$ 8E, LEFT$ 90, RIGHT$ 92, MID$ 94.

2. Push the accumulator, which contains the index, onto the stack.
3. Transfer the accumulator to the X-register.
4. If the X-register is less than $8F, branch to step 16. The comparison here is to determine if the token is for LEFT$, RIGHT$, or MID$, which are the only functions that need at least two parameters within their argument. Continue with step 5 for any of these three functions.
5. JSR A/CEFA to check for a (and display SYNTAX ERROR if not found.
6. JSR A/CD9E to evaluate the first expression.
7. JSR A/CEFD to check for a comma following this first expression and display SYNTAX ERROR if not found.
8. JSR A/CD8F to see if the expression was a string expression, and if not display TYPE MISMATCH.
9. PLA and TAX to hold the index for this function.
10. Push 65 and 64 onto the stack. (64) points to the string descriptor for the string expression that was evaluated in step 6.
11. TXA and PHA to push the index back onto the stack.
12. JSR B/D79E to evaluate the next expression following the comma. Display TYPE MISMATCH if it's a string expression. Display ILLEGAL QUANTITY if it's not in the range 0–255. Return with the integer value of this expression in the X-register.
13. PLA and TAY so that the Y-register now has the index into this function of LEFT$, RIGHT$, or MID$.
14. TXA and PHA to push the second parameter of the expression onto the stack. MID$ can optionally take a third parameter, but the MID$ function itself checks for this third parameter.
15. Branch to step 18.
16. Branch here for functions that need only one parameter in their argument. JSR A/CEF1 to evaluate the expression within parentheses. If it's a numeric value, FAC1 holds the floating point value. If it's a string value, (64) holds a pointer to the string descriptor.
17. PLA and TAY to restore the index for this function.
18. Y-register now contains the index into the function address table.

    LDA 9/BFEA,Y and STA 55.
    LDA 9/BFEB,Y and STA 56.

(55) now contains the address of this function. Remember that 54 contains the JMP opcode (direct addressing mode).

19. JSR 0054 to JMP to the address for this function. When the function finishes, it returns to the next step unless the function is a string function which has removed the return address to step 20.
20. JMP A/CD8D to see if the function returned a numeric value. If it didn't, display TYPE MISMATCH. If it did, FAC1 now contains the numeric floating point result of the function evaluation.

INPUT/GET/ READ

INPUT/INPUT#, GET/GET#, and READ are all similar in that they scan a location in memory for data and assign this data to the variable(s) in the variable list for the command.

Because they all share this feature of loading data from a memory location to a variable, a common routine has been used to handle this operation. During this common routine, a flag in 11 indicates which statement caused the execution of the common routine. Different processing is done in the common routine, depending on which one of READ, INPUT, or GET is executing. The main differences in the common routine are the location from which this data is read and the amount of data that can be read. For READ, the data is read from DATA statements that are located in the tokenized program text area. For INPUT or GET, the data received from the current input device is stored in a buffer at 0200 and the data is read from this buffer. GET limits the amount of data that can be read from the buffer to one byte.

INPUT# and GET# work very much like INPUT and GET except that they can receive data from an input device other than the keyboard. Since the Kernal CHRIN routine is used for INPUT or INPUT# and the Kernal GETIN routine is used for GET or GET#, see Volume 2 for details of INPUT# and / GET# processing for devices other than the keyboard.

If you try to assign a string to a numeric variable, you don't get any TYPE MISMATCH error. Rather, for GET and READ, you get a SYNTAX ERROR. For INPUT (from the keyboard), you receive the message ?REDO FROM START and are given another chance. The incorrect value assigned when trying to assign a string to a numeric value is discovered when the terminator of the last data item read from the buffer is not $00, a colon, or a comma.

If you try to assign a numeric value that is outside the range of integers to an integer variable, you get the ILLEGAL QUANTITY error.

If you try to assign a variable for READ and can't find another data item from a DATA statement, OUT OF DATA is displayed. This is the error that is displayed if you hit return when the cursor is on the READY line since an attempt is made to execute the READ portion of READY.

Both INPUT and READ allow you to read strings that are either in or not in quotes. If within quotes, everything within the quotes (including commas) can be assigned to a string variable. An example of this is DATA "5,3,7".

The comma that INPUT inserts at 01FF previous to the input buffer is used to force INPUT to get its item of data from the buffer rather than to issue another request to receive data from the current device and fill the input buffer. Receiving data and filling the input buffer is done in the separate code for INPUT, and this code makes certain that the buffer is not null before passing control to the common routine.

How does the common routine know when it is time to search for another DATA statement, do another GETIN or GET, or fill the buffer again for INPUT? It determines this by looking at the text pointer to the data area. If this text pointer points to a $00 or a colon, it's time to do a new scan for DATA, GETIN, or fill the buffer for INPUT. If the DATA pointer points to the end-of-statement or end-of-line of a DATA statement, it's time to scan for the next DATA statement. Also, RESTORE and CLR set the DATA pointer to the $00 byte that precedes the program text area. For GET, a CHRIN is always done, since on entering the common section the text pointer points to the $00 end-of-line byte for GET at 0201. After doing the CHRIN, the text pointer is again reset to 0201. The routine to receive and fill the input buffer for INPUT stores a $00 end-of-line byte in the buffer when a carriage return is received. When this $00 end-of-line byte is pointed to by the text pointer, it's time to do another prompt for INPUT, receive data from current device using CHRIN, and fill the input buffer. This prompt shows up on the screen as ?.

INPUT
A/CBBF–A/CBF8

Called by:
Execute Current BASIC Statement for INPUT; Alternate A/CBCE: JSR at A/CBB2 INPUT#.

Operation:

1. If the next character after the INPUT token is not a quotation mark, branch to step 5. Thus the prompt message must be a literal string and not a string variable.
2. For a quote as the first (nonspace) character following the INPUT token, JSR A/CEBD to scan the string and reset (7A) to the ending quote of the string.
3. JSR A/CEFF to see if a semicolon follows the prompt. If not, display SYNTAX ERROR.
4. JSR A/CB21 to remove the string descriptor for this string from the temporary string descriptor stack. (22) points to the string, and the accumulator contains its length. Then send each character of the string to the current output device by using the Kernal CHROUT routine.
5. A/CBCE: This is an entry point for INPUT#.

   JSR B/D3A6 to display ILLEGAL DIRECT if INPUT or INPUT# is entered in direct mode.
6. Store a comma at 01FF, which is immediately before where the input string will be placed.
7. JSR A/CBF9 to print ? prompt and cursor right. Then receive input from the current input device and fill the BASIC buffer at 0200. When the RETURN is received, an end-of-line $00 byte is stored.
8. If 13, the file number for I/O, is zero, branch to step 11. Also branch to step 11 if there are no read timeout errors.
9. For a nonzero value in 13 and a read timeout, JSR A/CBB5 to call the Kernal CLRCHN routine to close I/O channels. Also reset 13 to $00.
10. JMP A/C8F8 to scan for the end-of-statement byte of INPUT or INPUT# and reset (7A), the text pointer, to this byte in preparation for reading the statement following INPUT#. Then exit.
11. Branch here from step 8 when 13 = 0 or there are no read timeout errors.

    Load the accumulator from 0200, the first byte of the BASIC input buffer. This is where the responses to the INPUT are stored.
12. If the accumulator is not $00, a response was entered to the INPUT prompt that was not a null line. Branch to A/CC0D to the common READ/GET/INPUT routine to read this buffer at 0200.

13. If a null line was entered, see if 13 is zero. If 13 is not zero, branch to step 7 to repeat the prompt.
14. If 13 is zero, indicating keyboard input, JSR A/C906 to scan for the end-of-statement byte colon or end-of-line byte $00 of this INPUT statement. Return with offset to the end of the statement in the Y-register.
15. JMP A/C8FB to add the Y-register to (7A) to reset the text pointer to the end-of-statement byte of this INPUT, thus bypassing any further action for this INPUT from the keyboard.

Prompt for INPUT and Fill BASIC Buffer A/CBF9–A/CC05

Called by:
JSR at A/CBD6 INPUT/INPUT#; JSR at A/CC4A Common READ/GET/INPUT.

Operation:
1. If 13, the file number for CMD, is not zero, branch to step 4.
2. If 13 is zero, which indicates the keyboard is the input device, JSR A/CB45 to send ? to the screen.
3. Also send a cursor-right by JSR A/CB3B.
4. JMP A/C560 to receive characters from the current input device and fill the BASIC input buffer at 0200. When a carriage return is received, store an end-of-line $00 byte. Set the X-register and Y-register to point to 01FF.

INPUT# A/CBA5–A/CBBE

Called by:
Execute Current BASIC Statement for INPUT#; Alternate A/CBB5: JMP at A/CA83 PRINT#; JSR at A/CBE4 INPUT/INPUT#; Alternate A/CBB7: BNE at A/CBA2 GET#.

Operation:
1. JSR B/D79E to evaluate the expression following the #. If it does not evaluate to 0–255, display ILLEGAL QUANTITY or TYPE MISMATCH if it's a string expression. The X-register holds the 0–255 value, the file number, upon return.

2. JSR A/CEFF to see if a comma follows the file number. If not, display SYNTAX ERROR.
3. Store the X-register in 13, the file number for I/O.
4. JSR E11E/E11B to open the channel for this logical file using the Kernal CHKIN routine.
5. JSR A/CBCE to step 5 of INPUT. INPUT prompts with a ? to the device for this file number and fills the BASIC input buffer at 0200 with the response. If the input buffer contains $00, indicating only a RETURN was received from the input device, send another ? and again try to fill the buffer. Continue this loop until the buffer contains a value. The only way out of this loop is if a read timeout occurs from the input device. For a read timeout, 13 is reset to $00 and (7A) is moved to point to the end of the INPUT# statement.

   A/CBCE also handles assigning values from the input buffer to the variables following the INPUT#.
6. Load the accumulator with 13, the I/O file number.
7. JSR FFCC to the Kernal CLRCHN routine to clear the I/O channels.
8. Store $00 into 13.
9. RTS.

GET and GET#
A/CB7B–A/CBA4

Called by: Execute Current Statement for GET.

No separate token exists for GET#.

Operation:

1. JSR B/D3A6 to display ILLEGAL DIRECT if in direct mode.
2. See if the first nonspace character following GET is the ASCII character for #, $23. If not, branch to step 8 to handle GET. Because the # is scanned for by CHRGET rather than part of the token, spaces are permitted between GET and the #.
3. JSR CHRGET to retrieve the next nonspace character following the #.
4. JSR B/D79E to convert this expression following #, which is the file number, into a one-byte integer in the X-register from 0 to 255. If it's out of this range, display ILLEGAL QUANTITY. If it's a string expression display TYPE MISMATCH.

5. Check for a comma following the file number, and if it's not found, display SYNTAX ERROR. If found, leave (7A) pointing to the next nonspace character following the comma.
6. Store the X-register in 13, the file number for I/O. While you can set 13 to zero for GET#, if you attempt to open a file number zero, you'll get a FILE NOT FOUND error.
7. JSR E11E/E11B to open the channel for this logical file using the Kernal CHKIN routine.
8. For GET or GET#: Store a $00 end-of-line byte at 0201 in the BASIC input buffer, thus forcing a limit of one byte that can be returned by GET or GET#. Also set the X-register to $01 and the Y-register to $02 to point to this byte at 0201.
9. Load the accumulator with $40 to set the flag indicating that GET/GET# is being performed.
10. JSR A/CC0F to the common READ/INPUT/GET.
11. If 13 is not zero, branch to A/CBB7 to do the Kernal CLRCHN routine.

READ (and Common Portion for INPUT/INPUT# and GET/GET#)
A/CC06-A/CCFB

Called by:
Execute Current BASIC Statement for READ; Alternate A/CC0D: INPUT/INPUT#; Alternate A/CC0F GET/GET#.

Operation:
1. For READ: Load the X-register from 41 and the Y-register from 42. This (41) is initialized to point to the beginning of the BASIC tokenized program by CLR, RESTORE, or NEW. Otherwise, (41) points to the ending delimiter of the last DATA value read.
2. Load the accumulator with $98, the flag that indicates READ is being done. Fall through to step 4 by using a dummy BIT instruction.
3. A/CC0D: Entry point for INPUT/INPUT#. Load the accumulator with $00 to set the indicator that INPUT/INPUT# is being performed.
4. A/CC0F: Entry point for GET/GET# with the accumulator = $40.

Store the accumulator in 11, the flag which indicates whether READ $98, INPUT $00, or GET $40 is active.

5. Store the X-register and Y-register at (43), which is used as a temporary read pointer. For GET, this points to 0201, the $00 end-of-line byte. For INPUT, (43) points to 01FF, which is the comma preceding the input buffer. For READ, it points to $00 byte previous to the tokenized program or to the ending delimiter of the last data item.
6. Also branch to step 6 from step 51 after each variable in the list has been processed. JSR B/D08B to obtain the address of the next variable in the variable list for GET, INPUT, or READ. Leave (7A) pointing to the character following this variable, which should be either a comma or the end-of-statement or end-of-line byte. Display SYNTAX ERROR if a variable is not found. Return with the accumulator and Y-register pointing to the data area for this variable descriptor.

   Store the accumulator and Y-register into (49), a pointer to the data area for this variable descriptor.
7. Save the (7A) text pointer to the list of variables in (4B). Store (43), the temporary read pointer, in (7A).
8. JSR CHRGOT to retrieve the last character that stopped the data scan.
9. If $00 or a colon, branch to step 10. If not the end-of-line $00 or end-of-statement colon, branch to step 21—for example, if (43) points to the comma in a DATA statement that ended a previous READ. If the DATA pointer points to the start of the tokenized program, the initial $00 byte is retrieved. Also branch for INPUT if it's pointing to the comma at 01FF, which is the case for the initial INPUT and for following INPUTs that are separated by a comma.
10. Now that new data is needed, BIT 11 to see whether READ, INPUT, or GET is active.
11. If not GET, branch (BVC) to step 16.
12. If GET, then JSR E124/E121 to retrieve the next character from the current input device using the Kernal GETIN routine.
13. Store the character returned in 0200.
14. Set the X-register and Y-register to point to 01FF.
15. Branch to step 20.
16. If READ, branch (BMI) to A/CCB8 at step 52.

17. For INPUT: If 13 is not $00, INPUT# is active, so branch to step 19.
18. For INPUT, send ? to the screen device using the Kernal CHROUT routine.
19. JSR A/CBF9 to send ? and cursor right to the current output device. ? now appears if the end-of-line $00 of the input buffer has been detected while more variables need values. Then A/CBF9 does a JMP A/C560 to receive characters from the current input device and store them in the BASIC input buffer at 0200. If more than 88 characters are received, display STRING TOO LONG. When a carriage return is received, a $00 end-of-line byte is stored, returning with the X-register and Y-register pointing to 01FF.
20. Store the X-register in 7A and the Y-register in 7B.
21. A/CC51: Branch here if a DATA statement is found. Also branch here from step 9 if continuing the READ from a previous DATA statement or if INPUT is starting at 0200 after finding a comma at 01FF.

    JSR CHRGET to increment (7A) and then retrieve the byte pointed to by (7A). This is the byte at 0200 for GET, which is the only byte it requires.
22. BIT 0D to check the list of variables for READ, INPUT, or GET to see what type of variable is looking for a value.
23. If it's a numeric variable, branch to step 40 at A/CC89.

Steps 24–39 String Variable Processing

24. For a string variable, again BIT 11 to see whether this routine was called by GET, INPUT, or READ.
25. If not GET, branch to step 28.
26. For GET: INX and STX 7A.

    GET has already retrieved the byte at 0200 in step 21. This step moves the text pointer to the $00 end-of-line byte that was stored at 0201 during GET preparation.
27. Branch to step 32.
28. If READ or INPUT, store the accumulator into 07.
29. If this character is a quotation mark ("), branch to step 33 with 07 having quote as a scan terminator.
30. If this character is not a quotation mark, store a colon (:) in 07 as the scan terminator.
31. Load the accumulator with $2C, ASCII comma.
32. CLC.

33. Store the accumulator in 08. Thus if a quote was detected, the only terminator for the scan is a quote or the end-of-line $00. This allows string variables to be assigned values such as X$ = "3,2,5". If a quote is not found, either a colon or a comma terminates the scan.
34. Load the accumulator from 7A and the Y-register from 7B.
35. If the carry is set as a result of the quote being found, INY to move past the initial quote character.
36. JSR B/D48D to scan the string pointed to by the accumulator and Y-register. Since the string starts in page 2, allocate space in the active string space and copy the string there. Create a string descriptor in the temporary string descriptor stack and leave a pointer to this string descriptor in (64).
37. JSR B/D7E2 to store the pointer to the end of the string (71) in (7A) for use by CHRGET in scanning the next item of data in the input buffer.
38. JSR A/C9DA to copy the temporary string descriptor for this string to the data area at (49) of this string variable from the variable list.
39. JMP A/CC91 to step 43.

Steps 40–42 Numeric Variable Processing

40. JSR B/DCF3 to convert the ASCII string for this number into a floating point number in FAC1.
41. Load the accumulator from 0E, the integer/floating point flag.
42. JSR A/C9C2 to copy FAC1 into the data area of the variable descriptor pointed to by (49). If 0E indicates an integer value, FAC1 is first converted to a two-byte integer in 64 and 65, and this integer value is stored in the data area of the integer variable descriptor. If it indicates a floating point variable, the floating point number from FAC1 is stored as a floating point variable in the data area, with the high-order bit of the fraction serving as a sign.

Steps 43–51 See If Done with Variable List, Save Text Pointer to Input Buffer, and Reset Text Pointer to Variable List

43. JSR CHRGOT to see what character stopped the scan in the input buffer.
44. If the scan was stopped by $00, a colon, or a comma, branch to step 46.

45. For the scan stopped by any other character, JMP A/CB4D to the Bad Input routine.
46. Save (7A) in (43), the temporary READ pointer for resuming the scan of the input buffer from the $00, a colon, or comma.
47. Store (4B), the saved pointer to the text pointer scan of the variable list, into (7A) to prepare for reading the next variable.
48. JSR CHRGOT to retrieve the last character scanned in the variable list.
49. If the character is the end-of-statement colon or the end-of-line $00, branch to A/CCDF at step 64.
50. If not the end of the variable list, JSR A/CEFD to check for a comma and display SYNTAX ERROR if it's not found. Also, return with the accumulator containing the first nonspace character following the comma and the text pointer (7A) pointing to this character.
51. JMP A/CC15, step 6, to handle the next variable in the list.

Steps 52–63 READ: Scan for DATA Statement

52. Branch here from step 16 when a new DATA statement needs to be located. JSR A/C906 to scan for the $00 or colon that terminates the current statement. (7A) has been reset from 43 to point to the current DATA statement the first time this routine is reached. This could be pointing to the start of the program if a RESTORE has been done. This scan returns with the Y-register equal to the length to the end of this statement.
53. INY to point to the first byte of the next statement.
54. Transfer the accumulator, which contains the last character that ended the statement, either a colon or $00, to the X-register.
55. If it's not zero, branch to step 60 since a colon ended this statement.
56. If a zero ended the statement, INY to point to the high-order byte of the link field of this next line.
57. Load the accumulator from (7A),Y, the high-order link byte.
58. If this link byte is $00, the end of the program has been reached without finding another DATA statement. Display OUT OF DATA and return to the Main BASIC Loop.

59. Load the next two bytes of this line, which contain its line number, and store into (3F). (3F) is used to get the line number if a SYNTAX ERROR occurs during READ, such as trying to assign a string to a numeric variable.
60. JSR A/C8FB to modify (7A) by adding the Y-register to the current contents of (7A). (7A) now points to the data area for the next statement. This statement can either be the first one on a line or can follow a colon.
61. JSR CHRGOT to retrieve the first byte of this statement.
62. If this byte is not $83, the token for DATA, branch to step 52 to continue the scan for a DATA statement.
63. Once a DATA statement is found, JMP A/CC51 to step 21 to assign the values in this DATA statement to the variables in the READ statement.

Steps 64–70 End of Variable List Handling

64. Branch here from step 49 when the character in the variable list is the end-of-statement colon or the end-of-line $00.
65. Load the accumulator from 43 and the Y-register from 44. (43) was the temporary read pointer.
66. If 11 has its high-order bit off, then GET or INPUT was active—branch to step 68.
67. For READ: JMP A/C827 to store the accumulator in 41 and the Y-register in 42, thus saving the pointer to the ending delimiter of DATA that was read.
68. For GET or INPUT: If (43),0, which is the final byte scanned in the input buffer, is $00, then RTS. This is forced to be the case for GET.
69. If 13, the I/O file number, is nonzero, then RTS.
70. However, if the current input device is the keyboard, and if the input buffer was not fully read by INPUT, display the message EXTRA IGNORED.

Bad Input Routine
A/CB4D–A/CB7A

Called by:

JMP at A/CC9A READ/INPUT/GET When the scan in the input buffer was stopped by any character other than a colon, comma, or the $00 end-of-line.

Operation:

1. See which routine was active by loading the accumulator from 11 and doing a BIT test. If INPUT, branch to step 6. If READ, branch to step 3. For GET, continue with step 2.
2. Load the Y-register with $FF and branch to step 4.
3. For READ, load the accumulator from 3F and the Y-register from 40. (3F) contains the line number of the last DATA statement being scanned.
4. Store the accumulator in 39 and the Y-register in 3A. (39) is the current line number.
5. JMP A/CF08 to display SYNTAX ERROR. For READ, also display IN and the line number. For GET, the $FF in 3A suppresses the IN and line number message.
6. For INPUT: If 13, the file number for I/O, is zero, branch to step 8.
7. If 13 is not zero, the input device was other than the keyboard. JMP A/C437 to display the message FILE DATA and then exit.
8. For INPUT from the keyboard, display the message ?REDO FROM START.
9. For keyboard input, reset (7A) from (3D), which is the pointer to the end-of-statement byte of the last statement that was executed previous to the INPUT.

I/O Routines

The I/O routines for BASIC include the BASIC commands OPEN, CLOSE, SAVE, LOAD, VERIFY, INPUT#, GET#, CMD, and PRINT#.

CMD, PRINT#, GET#, and INPUT# require a previous OPEN so that these commands can relate their logical file number parameter to the corresponding entry in the device number table.

OPEN from BASIC calls the Kernal SETLFS routine to specify the logical file number, device number, and secondary address for a file. The OPEN information for these three items is stored in three tables that can each contain ten entries. One table is for logical file numbers, another for device numbers, and a third for secondary addresses.

The three values for any particular file are stored in the same relative-order in the three files. When one of the I/O commands requests a particular logical file number, that file number is searched for in the logical file number table. If it is found, the corresponding entries in the device number and secondary address tables can be located.

If the logical file number isn't found, the FILE NOT FOUND error occurs. If you try to OPEN a file and ten logical files have already been OPENed, you get the TOO MANY FILES error. It is not possible to OPEN a logical file number zero.

OPEN also does certain other things besides managing the logical file number, device number, and secondary address tables. For a serial device, OPEN commands the serial device to listen and then sends a secondary address for OPEN to this serial device. The filename information can also be sent to the serial device. For tape, OPEN checks for a tape header of a sequential file if reading or writes a tape header for a sequential file if writing to tape. For RS-232 certain handshake procedures are performed, and two 256-byte buffers are stolen from the top-of-BASIC memory to provide room for the buffer to transmit and receive characters.

Once the OPEN is completed, it is possible to do I/O to a logical file number that corresponds to a logical file that is open. Simply specify an INPUT#, CMD, PRINT#, or GET#, and BASIC handles the rest. BASIC calls the appropriate Kernal I/O routines as described below. BASIC itself performs no direct I/O operations. For details about the Kernal I/O routines, see the Kernal volume of *The Tool Kit.*

First a channel to the device is established. The CMD (and PRINT# which calls CMD) calls the Kernal CHKOUT routine to open an output channel. For tape, this involves checking the secondary address for a valid write specification and displaying NOT OUTPUT FILE if you gave a secondary address for READ. For serial and RS-232 devices, handshake procedures are performed. If the serial device doesn't handshake as expected, the DEVICE NOT PRESENT error message occurs. For INPUT# and GET#, the Kernal CHKIN routine to open an input channel is called. If the current input device for this logical file has a secondary address for a write and you try to do INPUT# or GET#, the NOT INPUT FILE error occurs. The handshake is performed for RS-232 or serial devices and the DEVICE NOT PRESENT can occur for a serial device.

Now that the channel is established, you can start transmitting or receiving characters over it. The Kernal CHROUT routine is used by PRINT. Unless you change things through an OPEN, the default channel is the screen, and that's where PRINT normally sends its characters. For other devices, CMD and PRINT# call PRINT to use this CHROUT routine.
The Kernal CHRIN routine is used by INPUT# to fill the BASIC input buffer at 0200. This same CHRIN routine is also used to fill the buffer when you are just entering characters on the screen. The Kernal GETIN routine is called by GET or GET# to receive one character from the current input device. GETIN doesn't have the possibility of hanging the system like CHRIN does since GETIN always returns with a $00 if it retrieves a null character or with a $0D if a serial I/O error occurs.

One feature of BASIC that is at times confusing is the difference between PRINT# and CMD. Also, how are LIST and PRINT related to CMD? Below is a short list which should help summarize the relationships:

PRINT#: Calls CMD; Calls CLRCHN to close all I/O channels and reset current output device to be the screen.
CMD: Calls CHKOUT to set new current output device; Calls PRINT.
PRINT: Sends characters to the current output device.

LIST: Calls PRINT.

The INPUT# and GET# commands are discussed in the section on INPUT/GET/READ. All of the other I/O commands are discussed in this section.

SAVE and LOAD/VERIFY both use the Kernal SAVE and LOAD/VERIFY routines to do all of the I/O work. The SETLFS and SETNAM that are required for the Kernal SAVE and LOAD/VERIFY are done by the BASIC SAVE and LOAD/VERIFY commands when these commands parse the parameters following the command.

One of the features for SAVEing to tape that is frequently specified incorrectly is the information about the secondary address. So far neither the VIC-20 nor Commodore 64 *Programmer's Reference Guide* has the information all straight. One source that does is Russ Davies' *Mapping the VIC.* Here's what we found for the tape SAVE secondary address: First, any even secondary address results in a tape ID of 1 to produce a relocatable program tape (examples: $00, $02, $04). (Relocatable means it can be loaded back into a different area of memory than the area it was saved from.) Any odd secondary address gives a tape ID of 3 for a nonrelocatable program tape (examples: $01, $03, $05). Also, if you have bit 1 on (the second bit from the right) in the secondary address, an end-of-tape header with a tape ID of 5 is written following the program (examples: $02, $03). If you don't specify anything for a secondary address for SAVE, a default of $00 is used, thus producing a relocatable program tape. This is the kind of tape that you should create when saving BASIC programs so that they can be loaded back to wherever BASIC currently finds the area that is reserved for its tokenized program. So my advice is: Leave the secondary address alone and everything works fine.

The logical file number you specify in an I/O command can be any numeric expression (including floating point and scientific notation) as long as the final integer result is in the range of 1–255.

The *Programmer's Reference Guide* says, "When PRINT# is used with tape files, the comma, instead of spacing by print zones, has the same effect as a semicolon." The *Guide* also says, "The data items are written as a continuous stream of characters." There is nothing in PRINT that check for a tape device when commas are detected. The only device specific item with PRINT connected with spaces is that PRINT sends a cursor-right to the screen for a space and sends actual spaces to other devices. When I tried PRINT# to a tape file using a list of variables separated by commas and then read the tape file back in using GET#, I found that the spaces were indeed sent just like PRINT was tabbing to the next TAB position for a comma.

One of the pleasant surprises when investigating the tape SAVE was that the restriction against SAVEing from $8000 or above that had existed on the VIC-20 has been removed on the Commodore 64. This fix was done by eliminating the check in tape SAVE that had used the same byte both for an indication that the SAVE was done (high-order bit on) and for the high-order byte of the area being SAVEd (when it reached $80 it forced an end-of-SAVE). Changing this limitation was a necessity on the Commodore 64 since the default BASIC memory area goes past $8000. If this change had not been made, any very large programs that went past $8000 couldn't have been completely saved.

OPEN
E1BE/E1BB–E1C6/E1C3

Called by: Execute the Current BASIC Statement for OPEN.

Operation:
1. JSR E219/E216 to handle the parameters for OPEN.
2. JSR FFC0 to the Kernal OPEN routine to open this logical file to allow use of the file in CMD, PRINT#, GET#, or INPUT#.
3. If an error occurred during OPEN, branch to E1D1/E1CE which jumps to E0F9/E0F6.

CLOSE
E1C7/E1C4–E1D3/E1D0

Called by: Execute the Current BASIC statement for CLOSE.

Operation:
1. JSR E219/E216 to handle the parameters for CLOSE, leaving the file number in 49.
2. Load the accumulator with this file number from 49.
3. JSR FFC3 to the Kernal CLOSE routine to close this logical

file, removing the entries for it from the logical file number, device number, and secondary address tables.

4. If there are no errors during CLOSE, exit.
5. If an error occurred during CLOSE or if it's a CLOSE for an RS-232 device, JMP E0F9/E0F6.

Handle Parameters for OPEN/CLOSE E219/E216–E263/E260

Called by: JSR at E1BE/E1BB OPEN; JSR at E1C7/E1C4 CLOSE.

Operation:

1. Load the accumulator with $00. JSR FFBD to the Kernal SETNAM routine to set a default filename size of zero in B7.
2. JSR E211/E20E to see if any parameters follow the OPEN/CLOSE. If no parameters, display SYNTAX ERROR.
3. JSR B/D79E to evaluate the first parameter following the OPEN/CLOSE. If it's nonnumeric, display TYPE MISMATCH. If it's not in the range 0–255, display ILLEGAL QUANTITY. Return with the value in the X-register.
4. Store the X-register in 49 to save the file number in case more than one parameter is present. Also transfer the X-register to the accumulator.
5. LDX $01 to set default device number of 1, the tape cassette.
6. LDY $00 to set default secondary address of zero for tape READ.
7. JSR FFBA to the Kernal SETLFS routine to set the logical file number from the accumulator, the device number from the X-register, and the secondary address from the Y-register.
8. JSR E206/E203 to see if the end of the OPEN/CLOSE statement has been reached. If so, just exit after processing this parameter for the file number.
9. JSR E200/E1FD to check for a comma following the first parameter. If not found, display SYNTAX ERROR. If a comma is found, evaluate the second parameter. Again return the value in the X-register of 0–255 and display TYPE MISMATCH if it's a string expression or ILLEGAL QUANTITY if it's not 0–255.

10. Store the X-register in 4A to save the device number in case more than two parameters are present.
11. LDY $00 to set a default secondary address of zero for read.
12. LDA 49 to retrieve the first parameter that gave the file number.
13. If 4A, the device number, is greater than three and thus indicating a serial device, DEY to set secondary address of $FF.
14. JSR FFBA to the Kernal SETLFS routine to set the logical file number from the accumulator, the device number from the X-register, and the secondary address from the Y-register.
15. JSR E206/E203 to see if the end of the OPEN/CLOSE statement has been reached. If so, just exit after processing these first two parameters for the file number and the device.
16. JSR E200/E1FD to check for a comma following the second parameter. If not found, display SYNTAX ERROR. If a comma is found, evaluate the third parameter. Again return the value in the X-register of 0–255 and display TYPE MISMATCH if it's a string expression or ILLEGAL QUANTITY if it's not 0–255.
17. TXA and TAY to move this third parameter to the secondary address register.
18. LDX 4A to retrieve the device number previously set.
19. LDA 49 to retrieve the logical file number.
20. JSR FFBA to the Kernal SETLFS routine to set the logical file number from the accumulator, the device number from the X-register, and the secondary address from the Y-register.
21. JSR E206/E203 to see if the end of the OPEN/CLOSE statement has been reached. If so, just exit after processing these first three parameters for the file number, the device, and the secondary address.
22. JSR E20E/E20B to check for a comma following the third parameter. If not found, display SYNTAX ERROR.
23. If a comma is found, evaluate the fourth parameter by JSR A/CD9E. Return with (64) pointing to the string descriptor for this fourth parameter, the filename. This fourth parameter contains the filename, which can also include the type such as SEQ and the mode such as W.

24. JSR B/D6A3 to see if the expression from step 23, the fourth parameter, was a string. If not, display TYPE MISMATCH. If it was a string, remove the string descriptor from the temporary string descriptor stack with (22) pointing to the string and the accumulator containing its length.
25. LDX from 22 and LDY from 23.
26. JMP FFBD to the Kernal SETNAM routine to set the filename using the pointer to the name in the X-register and Y-register and the length of the name from the accumulator.

Check for Comma in Parameter List and Evaluate Following Parameter to Integer Value in X-Register E200/E1FD-E205/E202

Called by:
JSR at E1E9/E1E6, E1F6/E1F3 LOAD/SAVE Parameters; JSR at E231/E22E, E245/E242 OPEN/CLOSE Parameters.

Operation:
1. JSR E20E/E20B to check for a comma and display SYNTAX ERROR if it's not found.
2. JMP B/D79E to evaluate the expression following the comma. If it's nonnumeric, display TYPE MISMATCH. If it's not 0–255, display ILLEGAL QUANTITY.

Check If End of Parameter List E206/E203-E20D/E20A

Called by:
JSR at E1E0/E1DD, E1E6/E1E3, E1F3/E1F0 LOAD/SAVE Parameters; JSR at E22E/E22B, E242/E23F, E251/E24E OPEN/CLOSE Parameters.

Operation:
1. JSR CHRGOT to load the accumulator with the last character scanned in the parameter list.
2. If not the $00 end-of-line or colon end-of-statement, then RTS.
3. If end-of-line or end-of-statement, pull this routine's return address off the stack and then RTS. Thus this RTS will serve as the RTS for the LOAD/SAVE Parameters Routine or the OPEN/CLOSE Parameters Routine.

Check for Comma Between Parameters E20E/E20B–E218/E215

Called by:
JSR at E200/E1FD Evaluate Following Parameter to Integer; JSR at E254/E251 OPEN/CLOSE Parameters; Alternate E211/E20E JSR at E21E/E21B OPEN/CLOSE Parameters.

Operation:
1. JSR A/CEFD to check for a comma and display SYNTAX ERROR if not found. A/CEFD then calls CHRGET to retrieve the following nonspace character.
2. E211/E20E: JSR CHRGOT to load this last character retrieved back into the accumulator.
3. If it's not zero or colon, then RTS.
4. If it's the $00 end-of-line or colon end-of-statement, display SYNTAX ERROR since a comma appears without a following parameter.

SAVE E156/E153–E164/E161

Called by: Execute the Current BASIC Statement for SAVE.

Operation:
1. JSR E1D4/E1D1 to handle the SETNAM and SETLFS procedures for SAVE.
2. Set the end address plus one for the SAVE. Load the X-register from 2D. Load the Y-register from 2E. (2D) points to the end of the tokenized program plus one.
3. Set the starting address for the SAVE. LDA $2B to set the offset to (2B). (2B) contains the starting address of the tokenized program.
4. JSR FFD8 to the Kernal SAVE routine.
5. If the carry is set upon return, an error occurred during SAVE, so branch to E0F9/E0F6 to handle the error.
6. If the carry is clear, the SAVE did not have any errors, so exit.

VERIFY E165/E162–E167/E164

Called by: Execute the Current BASIC Statement for VERIFY.

Operation:
Load the accumulator with $01 to indicate that this is a VERIFY operation, and then fall through to the LOAD routine, step 2, through the use of a BIT instruction.

LOAD
E168/E165–E1BD/E1BA

Called by:
Execute the Current BASIC Statement for LOAD; Alternate E167/E164: Fall through from VERIFY.

Operation:

1. Load the accumulator with $00 to indicate LOAD.
2. E167/E164: Store the accumulator into 0A, which is $00 for LOAD or $01 for VERIFY.
3. JSR E1D4/E1D1 to do the SETNAM and SETLFS for LOAD/VERIFY.
4. Load the accumulator from 0A. LDX from 2B. LDY from 2C. (2B) points to the start of the area for a tokenized BASIC program.
5. JSR FFD5 to the Kernal LOAD/VERIFY routine.
6. If the carry is set upon return, branch to E1D1/E1CE to handle the error.
7. If there's no error, load the accumulator from 0A.
8. If the accumulator is zero, a LOAD was done, so branch to step 13.
9. For VERIFY: JSR FFB7 to the Kernal READST routine to read the I/O status.
10. If there are no I/O errors, branch to step 12.
11. For any I/O errors during VERIFY, display VERIFY ERROR.
12. If the VERIFY was done from program mode, just exit. If it was done from direct mode, display OK and exit.
13. For LOAD: JSR FFB7 to the Kernal READST routine to read the I/O error byte into the accumulator.
14. AND the accumulator with $BF. Thus the end-of-file for cassette READ error is not detected.
15. If the accumulator is now zero, branch to step 17.
16. For the accumulator of nonzero, display LOAD ERROR and exit to the Main·BASIC Loop.
17. If LOAD was issued from program mode, branch to step 21.

18. For a LOAD from direct mode: Store the X-register in 2D and the Y-register in 2E. (2D) points to the end of the BASIC program plus one.
19. Display READY.
20. JMP A/C52A to reset (7A) to the start of the BASIC program, relink BASIC lines, and go to the Main BASIC Loop.
21. For a program mode LOAD: JSR A/C68E to reset (7A) from the (2B) pointer to the start of the BASIC program.
22. JMP A/C533 to relink the BASIC lines.
23. JMP A/C677 to restore DATA pointer to the start minus one of the text area, reset the stack pointer, reset the temporary string descriptor stack pointer, disallow CONT, and clear the flag that prevents subscripts. The pointers to the variable area, the arrays, the free area, and the active string area are not reset, so all variables are still retained from the previous program. If the program that is LOADed is longer than the original program, the newly LOADed longer program will overlay the variable area.

    Note: Steps 22 and 23 are found at E476 on the VIC-20 in a patch area that was put back into the section for LOAD on the Commodore 64.

Handle LOAD/SAVE Parameters for SETNAM and SETLFS E1D4/E1D1-E1FF/E1FC

Called by:
JSR at E156/E153 SAVE; JSR at E16C/E169 LOAD/VERIFY.

Operation:

1. Load the accumulator with $00 and JSR FFBD to the Kernal SETNAM routine to set a default filename of length zero for null filename.
2. LDX $01 to set the default device number. LDY $00 to set default secondary address for READ (LOAD).
3. JSR FFBA to the Kernal SETLFS routine to set the default secondary address and device number.
4. JSR E206/E203 to see if there are any parameters for LOAD/SAVE. If not, just exit.
5. JSR E257/E254 to evaluate the first parameter, the filename. If not a string, display TYPE MISMATCH. Set the accumulator to the length of the name, the X-register

and the Y-register to its location, and then use the Kernal SETNAM routine to use this filename for the LOAD/ SAVE.

6. JSR E206/E203 to see if any parameter follows the filename. If not, just exit.
7. JSR E200/E1FD to check for a comma following the filename. If none is found, display SYNTAX ERROR. After the comma, evaluate the second parameter, the device number. If this second parameter is a string, display TYPE MISMATCH. If it's not in the range 0–255, display ILLEGAL QUANTITY. Return with the number in the X-register.
8. Load the Y-register with $00 to set the default secondary address for READ.
9. STX in 49 to save the device number in case a third parameter is present.
10. JSR FFBA to the Kernal SETLFS routine to use this device number just specified and the secondary address of zero.
11. JSR E206/E203 to see if any parameter follows the device number. If not, just exit.
12. JSR E200/E1FD to check for a comma following the device number. If none is found, display SYNTAX ERROR. After the comma, evaluate the third parameter, the secondary address. If this third parameter is a string, display TYPE MISMATCH. If it's not in the range 0–255, display ILLEGAL QUANTITY. Return with the number in the X-register.
13. LDX from 49, the same device number parameter.
14. JMP FFBA to the Kernal SETLFS routine.

PRINT#
A/CA80–A/CA85

Called by: Execute the Current BASIC Statement for PRINT#.

Operation:

1. JSR A/CA86 to execute CMD, which opens a channel for output for the logical file specified and then executes PRINT.
2. JMP A/CBB5 to call the Kernal CLRCHN routine to clear all channels and reset 13, the I/O command file number, to zero.

CMD
A/CA86–A/CA99

Called by:
Execute the Current BASIC Statement for CMD; JSR at A/CA80 PRINT#.

Operation:

1. JSR B/D79E to convert the expression following the CMD or PRINT#, which should be the file number of a previously opened file, into an integer from 0 to 255. If it's out of this range, display ILLEGAL QUANTITY. If it's a string expression, display TYPE MISMATCH. Return with the integer in the X-register.
2. If nothing followed the file number, branch to step 4.
3. If something did follow the file number, see if this character was a command. If not, display SYNTAX ERROR.
4. PHP to save the BEQ status if nothing followed the file number. PRINT uses this status information to know if a blank line is to be printed.
5. STX in 13, the I/O file number for PRINT and LIST.
6. JSR E118/E115 to execute the Kernal CHKOUT routine to open the channel for this logical file number.
7. PLP to restore a possible blank line status indicator.
8. JMP A/CAA0 to execute PRINT. If no parameters followed the file number, PRINT just sends a carriage return. If the file number is >=128, PRINT also sends linefeed.

BASIC's CHROUT
E10C/E109–E111/E10E

Called by: JSR at A/CB47 PRINT.

Operation:

1. JSR FFD2 to the Kernal CHROUT routine to send the character in the accumulator to the current output device, 9A.
2. If carry is set, branch to E0F9/E0F6 to handle Kernal call errors.

BASIC's CHRIN
E112/E10F-E117/E114

Called by:
JSR at A/C562 Receive Characters and Fill BASIC Input Buffer.

Operation:
1. JSR FFCF to the Kernal CHRIN routine to load a character into the accumulator from the current input device, 9A.
2. If carry is set, branch to E0F9/E0F6 to handle Kernal call errors.

BASIC's CHKOUT
E118/E115-E11D/E11A

Called by: JSR at A/CA93 CMD.

Operation:
1. JSR FFC9 to the Kernal CHKOUT routine.
2. If carry is set, branch to E0F9/E0F6 to handle Kernal call errors.

BASIC's CHKIN
E11E/E11B-E123/E120

Called by: JSR at A/CB8F GET#; JSR at A/CBAF INPUT#.

Operation:
1. JSR FFC6 to the Kernal CHKIN routine.
2. If carry is set, branch to E0F9/E0F6 to handle Kernal call errors.

BASIC's GETIN
E124/E121-E129/E126

Called by: JSR at A/CC35 GET/GET#.

Operation:
1. JSR FFE4 to the Kernal GETIN routine.
2. If carry is set, branch to E0F9/E0F6 to handle Kernal call errors.

Handle Kernal Call Errors and RS-232 CLOSE E0F9/E0F6–E10B/E109

Called by:
The BASIC Kernal Call Routines shown above and by a JMP at E1D1/E1CE for OPEN/CLOSE.

Operation:

1. Compare the accumulator to $F0, which is the value set by the RS-232 CLOSE routine.
2. If not equal, branch to step 5.
3. For an RS-232 CLOSE, STY 38 and STX 37 to reset the pointer to the end of BASIC memory since the RS-232 buffers have been released.
4. JMP A/C663 to execute CLR.
5. Transfer the accumulator to the X-register to be used to retrieve an error message.
6. JMP A/C437 to display the error message and return to the Main BASIC Loop.

Comparison of Operands

Two operands can be compared during evaluation of any expression as long as the two operands are of the same type. That is, both string operand comparison and numeric operand comparison are valid.

The comparison can be for <, =, > or any combination of these three tests. If any of the conditions of the comparison are found to be true, the result in the Floating Point Accumulator 1 (FAC1) upon return from the comparison is −1. For an untrue comparison, FAC1 is 0.

Comparison of Operands B/D016–B/D02D

Called by:
Prepare for Stacked Operator Execution and RTS to Stacked Operator Routine During Expression Evaluation.

Entry Conditions:
FAC2 has the left operand value, FAC1 has the right operand value.

These values are string descriptor pointers if the operands are strings. 12 has the condition or combination of conditions being tested: For left operand < right operand—00000100; for left operand = right operand—00000010, for left operand > right operand—00000001. Any combination of these comparisons is valid, even <=>. The carry is set when performing a comparison of strings, and it's clear when comparing numeric expressions.

Operation:

1. JSR A/CD90 to see if the right operand matches the type of the left operand, and if not, display TYPE MISMATCH.
2. BCS to B/D02E to do comparison of strings if the carry was set at entry to this routine.
3. For a comparison of numeric expressions, set the high-order bit of 6A based on the sign of FAC2, 6E.

4. Load the accumulator with $69 and load the Y-register with $00 to point to FAC2, which contains the left operand.
5. JSR B/DC5B to compare the floating point number in FAC2 to FAC1. Return with accumulator = 0 if FAC2 (left operand) = FAC1 (right operand), accumulator = $FF if FAC2 < FAC1, accumulator = $01 if FAC2 > FAC1.
6. Transfer the accumulator into the X-register.
7. JMP B/D061 to step 20 of String Comparison. At B/D061 do the following:

   If the accumulator = $01 or $00, clear the carry.

   INX so now the X-register is $01 if =, $00 if >, or $02 if <.

   TXA and ROL, which rotates the carry back into the low-order bit. Thus accumulator = 00000010 if =, 00000001 if <.

   AND the accumulator with 12, which holds the conditions being tested.

   If the AND left the accumulator with a nonzero value, at least one condition matched. In this case, set a result in FAC1 of $FF or −1.

   If the AND left the accumulator with a zero value, nothing matched, so set a result in FAC1 of 0.

String Comparison
B/D02E-B/D07D

Called by:
BCS at B/D019 Comparison of Operands; Alternate B/D061: JMP at B/D02B Comparison of Operand (for numerics).

Operation:

1. Set 0D to zero to indicate the result of this comparison of strings is a numeric value.
2. Decrement 4D. Picture the before and after values for the different comparison conditions:

	Less than	**Equal**	**Greater than**
Before	0000 1001	0000 0101	0000 0011
After	0000 1000	0000 0100	0000 0010

3. JSR B/D6A6 to delete the temporary string descriptor for the right operand string from the temporary string descriptor stack and return with the accumulator = length of the string and (22) pointing to the string.

4. Store the accumulator in 61 to save the length, and store (22) in (62) to save the pointer to the right operand.
5. Load the accumulator from 6C and load the Y-register from 6D to load the string descriptor for the left operand from the value that was saved in FAC2 when preparing to execute the comparison operator routine.
6. JSR B/D6AA to delete the temporary string descriptor for this left operand string and return with the accumulator = length of the string and (22) (also the X-register and the Y-register) pointing to the string.
7. Save the X-register and Y-register in (6C) which now contains a pointer to the left operand string.
8. TAX to store the length of left operand string in the X-register.
9. Subtract the length of the left operand string from the length of the right operand string.
10. If the strings are equal in length, branch to step 13 with accumulator = $00.
11. If the right operand string is longer, load the accumulator with $01 and branch to step 13.
12. For a left operand string longer than the right operand string, LDX 61, the length of the right operand string, and load the accumulator with $FF.
13. Store the accumulator in 66. 66 is $00 if the length of the left string = the length of right string; 66 is $01 if the left string length < right string length; 66 is $FF if the left string length > right string length.
14. Load the Y-register with $FF so that the initial value after the first INY is zero.
15. INX so that the X-register is now the length of the smaller string plus one.
16. INY.
17. DEX, which holds the length of the smaller string.
18. If the X-register is not zero, branch to step 22.
19. If the X-register is zero, LDX 66. This step occurs if the smaller string is a null string, or if the strings are equal for the entire length that was compared but one of the strings was shorter. It also occurs when the strings are equal the entire way through. Fall through to step 20.
20. B/D061: Entry point for numeric compare. BMI to step 27.
21. CLC and BCC to step 27.

22. Load the accumulator from (6C),Y (the left operand string) and compare to (62),Y (the right operand string).
23. If these two characters are equal, branch to step 16 to check the next two characters.
24. LDX $FF for the character of the left string being > the character of the right string.
25. BCS if > to step 27.
26. LDX $01 if the left string character < the right string character.
27. INX, TXA, and ROL.
    The following diagram shows what the accumulator now is for string comparisons (significant bits only):

Comparison Ended Because of:

Unequal length or Matched Strings			**Unlike Characters**	
L=R	L<R	L>R	L>R	L<R
010	100	001	001	100

28. Now that the result of the comparison is known, see if it matches the comparison that was specified. 12 holds the comparison that was specified, which was (significant bits only) 010 if =, 100 if <, 001 if >, or any combination of these. AND the accumulator with 12.
29. If the result of the AND is zero, then no conditions requested were met in the comparison test. Branch with the accumulator = 0 to step 31.
30. If the accumulator is nonzero, at least one condition called for was found to be true. Load the accumulator with $FF.
31. JMP B/DC3C to convert the 0 or $FF (−1) to a floating point value in FAC1, which is thus returned in the expression evaluation for this comparison.

PEEK/POKE/ WAIT

PEEK, POKE, and WAIT all reference a memory location. PEEK returns the value in the location, POKE puts a new value there, and WAIT waits for a certain value to appear in the memory location.

PEEK
B/D80D-B/D823

Called by: Invoke Function for PEEK.

Operation:
1. Save 14 and 15 on the stack.
2. FAC1 contains the result of evaluating the argument for PEEK, which contains a memory location to be examined. JSR B/D7F7 to convert FAC1 to a two-byte integer value in 14 and 15. If FAC1 is greater than 65535, display ILLEGAL QUANTITY.
3. Load the accumulator from (14),0. (14) points to the memory location you specified in the argument for PEEK. The value in the accumulator can range from $00–$FF or 0–255 (decimal). Transfer the accumulator to the Y-register.
4. Restore 14 and 15 from the stack.
5. JMP B/D3A2 to convert the Y-register to a normalized floating point number in FAC1, which is thus forwarded on to expression evaluation.

Convert Address in Floating Point Format to Two-Byte Fixed Value
B/D7F7-B/D80C

Called by:
JSR at B/D7EE Get POKE/WAIT Address; JSR at B/D813 PEEK; JSR at E12D/E12A SYS.

Operation:
1. If the sign of FAC1 is negative, display ILLEGAL QUANTITY.

2. If the exponent of FAC1, 61, is > 17 (decimal [$91 with excess-128]), the number in FAC1 is longer than two bytes and is thus greater than 65535, the maximum address. Display ILLEGAL QUANTITY if true.
3. JSR B/DC9B to convert the floating point number in FAC1 to a four-byte signed integer in FAC1. The check in step 2 insures that only the two low-order bytes will contain significant digits.
4. Load the accumulator from 64 and the Y-register from 65.
5. Store the accumulator in 14 and the Y-register in 15.

POKE
B/D824-B/D82C

Called by: Execute Current BASIC Statement for POKE.

Operation:
1. JSR B/D7EB to convert the first parameter for POKE, the address, to a two-byte integer in (14) in the range 0–65535. The second parameter is converted to an integer in the range 0–255 in the X-register. This routine also does value- and type-checking.
2. TXA so that the accumulator now contains the value to be POKEd.
3. Store the accumulator at (14),0.

Get Parameters for POKE and WAIT
B/D7EB-B/D7F6

Called by:
JSR at B/D824 POKE; JSR at B/D82D WAIT; Alternate B/D7F1: JSR at B/D839 WAIT.

Operation:
1. JSR A/CD8A to evaluate the first parameter of POKE or WAIT and display TYPE MISMATCH if this expression is a string. Leave the numeric expression value in FAC1.
2. JSR B/D7F7 to convert FAC1 to an integer in the range 0–65535 with the result stored in (14).
3. B/D7F1: JSR A/CEFD to check for a comma. If it's not found, display SYNTAX ERROR.
4. JMP B/D79E to evaluate the expression following the comma. If it's a string expression, display TYPE MIS-

MATCH. If it's not in the range of 0–255, display ILLEGAL QUANTITY. Store the 0–255 value into the X-register and exit.

WAIT
B/D82D–B/D848

Called by: Execute Current BASIC Statement for WAIT.

Operation:

1. JSR B/D7EB to evaluate the first two parameters of WAIT, leaving the address of 0–65535 in (14) and the AND mask of 0–255 in the X-register.
2. Store the X-register in 49, the AND mask.
3. Load the X-register with $00, the default for the third parameter, the EOR value.
4. JSR CHRGOT to see if the second parameter was followed by end-of-statement or end-of-line, and if so branch to step 6.
5. JSR B/D7F1 to convert the third parameter to an integer from 0 to 255 and store in the X-register.
6. Store the X-register in 4A, the EOR value.
7. Load the accumulator from (14), 0 to get the byte from the memory location specified.
8. EOR 4A.
9. AND 49.
10. BEQ to step 7.
11. If the EOR and AND ever result in the accumulator becoming nonzero, then RTS.

POS

POS returns the column location of the cursor within the logical screen line. For the Commodore 64, the value returned is 0–79. For the VIC, the possible values are 0–87. The argument to POS is a dummy argument.

POS
B/D39E–B/D3A1

Called by: Invoke Function for POS.

Operation:
1. SEC to ask for reading the cursor position.
2. JSR FFF0 to the Kernal PLOT routine, which returns in the Y-register the current column of the cursor within the logical screen line.
3. Fall through to B/D3A2 to convert the Y-register to a floating point number in FAC1.

RND

Whenever a BASIC cold start occurs, the routine at E453/E45B to Initialize Vectors copies the seed for RND, the floating point value for .811635157 to 8B-8F. RND returns a random number, but depending on the argument, the method of generating the random number varies.

If you specify a positive argument to RND, a new random number is generated based on the previous seed that exists in 8B–8F. This previous seed is multiplied by 11,879,546.4 (decimal) and then 3.92767778E−8 is added. The value of the positive argument is not used in the calculation, just the fact that it is positive.

For a negative argument, the argument, which has been converted to a floating point number, is used to generate a new random number.

For a zero argument, a random number is generated based on the VIA#1 Timers for the VIC or the CIA#1 Timers for the 64.

For any one of these arguments, once the initial value in FAC1 is established, 62 and 65 are swapped and 63 and 64 are swapped. These are the four bytes of the fraction. Then the exponent for FAC1 is set to $80 to force a value between zero and one into FAC1. FAC1 is then normalized, and the resulting normalized FAC1 is copied to the RND seed area at 8B–8F for the next call of RND. FAC1 is also returned as the value of the function call.

If you specify a negative argument and then later specify the same negative argument, you will get the same random number since the random number is generated based only on the argument.

If you specify a positive argument, you generate a random number based on the previous seed in 8B–8F. If you start from the same seed, a series of random numbers will also produce the same values. If you specify an argument of zero, then for any one call of RND you get what appears to be a truly random number based on the timers. Generally, using a zero argument would be a good choice for generating a series of random numbers. See Raeto Collin West's book *Programming the PET/CBM* for one caution about using a zero argument.

Since RND also returns a value based on some previous state, if you know the previous state you can predict the random number that is generated. For an extensive discussion of random numbers, see Donald Knuth's *The Art of Computer Programming,* Vol. 2: *Seminumerical Algorithms.*

RND
E097/E094–E0F8/E0F5

Called by:
Invoke Function for RND; Alternate E0F6/E0F3: JSR at E2C2/E2BF TAN.

Operation:

1. JSR B/DC2B to determine the sign of FAC1 which contains the evaluated argument for RND.
2. If it's negative, branch to step 15 to use the argument, which is in FAC1, as the seed.
3. If it's not equal to zero, branch to step 9 to use the old seed.
4. If it's zero, create a random number based on the VIA (VIC-20) or CIA (Commodore 64) timers.
5. JSR FFF3 to the Kernal IOBASE routine to get the base address of VIA or CIA registers. For the VIC, this base address is 9110, and for the Commodore 64, it is DC00. Return this address in the X-register and Y-register.
6. Store the X-register and Y-register into (22) to serve as a pointer to the VIA/CIA registers.
7. For the VIC: Store VIA#1 Timer 1 low-order byte in 62, store VIA#1 Timer 1 high-order byte in 64, store VIA#1 Timer 2 low-order byte in 63, and store VIA#1 Timer 2 high-order byte in 65. Timer 1 for VIA#1 is initialized to free running mode, and Timer 2 for VIA#1 is initialized to one-shot mode.

   For the Commodore 64: Store CIA#1 Timer A low-order byte in 62, store CIA#1 Timer A high-order byte in 64, store CIA#1 time-of-day clock: 1/10 seconds in 63, and seconds in 65.
8. JMP to step 16.
9. For a positive argument to RND, point the accumulator and Y-register to 8B, the address of the RND seed.
10. JSR B/DBA2 to copy the RND seed to FAC1.

11. Point the accumulator and Y-register to E08D/E08A, the floating point value for 11,879,546.4 (decimal).
12. JSR B/DA28 to multiply this number by FAC1, the RND seed, and leave the result in FAC1.
13. Point the accumulator and Y-register to E092/E08F, the floating point value for 3.92767778E−8 (decimal).
14. JSR B/D867 to add this value to FAC1. Continue with step 15.
15. Branch here if it's a negative argument. Reverse 62 and 65. Reverse 63 and 64.
16. Set the sign of FAC1, 66, to $00 to force a positive sign.
17. Store 61, the exponent, in 70, the rounding byte, to insure that a zero won't be the final value in FAC1 after normalization.
17. Set the exponent, 61, to $80, to force the value in FAC1 to be in the range of 0–1 (not including 0 or 1).
18. JSR B/D8D7 to normalize FAC1.
19. Point the accumulator and Y-register to 008B, the location for the RND seed.
20. E0F6/E0F3: JMP B/DBD4 to copy FAC1 to 8B–8F for RND. For SIN save during TAN computation, copy FAC1 to 4E–53.

SYS and USR

SYS and USR both allow you to execute machine language programs from BASIC. They each do this in a different way.

SYS is a command. SYS evaluates its argument into an address and then jumps to that location. SYS allows you to pass parameters to and receive parameters from your machine language program. The parameters are passed in 030C–030F (780–783 decimal), as shown below with their decimal addresses, since you will probably be using PEEK and POKE to access these locations:

780—Accumulator
781—X-Register
782—Y-Register
783—Status Register

Before the machine language program is jumped to, the registers are loaded from these locations. After the program finishes, the register values are stored back into the appropriate locations.

USR is a function. USR does a JMP 0000 (VIC) or a JMP 0310 (Commodore 64). Normally, at 0000 (VIC) and at 0310 (Commodore 64) a direct jump opcode is located and a jump is made to the address specified in location (01) for the VIC or (0311) for the 64. If you don't modify these locations (normally modified by using POKE), USR returns ILLEGAL QUANTITY. Once you place the address of your machine language program in (01) or (0310), USR jumps to that address and returns to the final step of function invocation when your program does an RTS. This final step of function invocation checks that the value returned in FAC1 is a numeric value.

The jump opcode that is normally found at 00 or 0310 can also be changed. For example, you could put an RTS there which would immediately return to your BASIC program. On the VIC (but not the Commodore 64), you can do a couple of other unusual things. Locations 3 and 4 normally contain a vector to the floating-point-to-integer conversion routine at B/D1AA. Locations 5–6 normally contain a vector to the integer-to-floating-point conversion routines. Since a JMP 0000 is executed by function invocation (actually function invocation does a JSR 0054 and the JMP 0000 is done from there), you can place a short machine language program there. Consider a couple of examples:

Example 1:
JMP to the floating-point-to-integer conversion routine passing an argument in FAC1.

Location	Object Code	Instruction
0000	24 00	BIT$00
0002	4C AA D1	JMP$D1AA

The required POKEs are:

POKE 0,36
POKE 1,0
POKE 2,76

The address at 3 and 4 already exists.

Example 2:
JMP to the integer-to-floating-point conversion routine passing an argument in FAC1.

Location	Object Code	Instruction
0000	24 00	BIT$00
0002	EA	NOP
0003	EA	NOP
0004	4C 91 D3	JMP$D391

The required POKEs are:

POKE 0,36
POKE 1,0
POKE 2,234
POKE 3,234
POKE 4,76

The address at 5 and 6 already exists.

Example 3.
Multiply location FF (255 decimal) by 8. In such an example, the argument to USR would be a dummy argument.

Location	Object Code	Instruction
0000	06 FF ASL FF	
0002	06 FF ASL FF	
0004	06 FF ASL FF	
0006	60	RTS

The required POKEs are:

POKE 0,6
POKE 1,255
POKE 2,6
POKE 3,255
POKE 4,6
POKE 5,255
POKE 6,96

Finally, it is possible to pass a string expression to USR. Function invocation does no type checking of the expression before calling a function. Rather, the type check for a numeric value is done after return from the function.

How do string functions such as LEFT$, RIGHT$, and MID$ get around the type check for a numeric result? They just throw away the return address to the function invocation routine and thus return to the previous calling steps in expression evaluation. The string functions also create a string descriptor for the new string and place it in the temporary string descriptor stack. The address of the string descriptor is returned in (64). See the string functions for examples on which to base your USR string function.

SYS
E12A/E127–E155/E152

Called by: Execute Current BASIC Statement for SYS.

Operation:

1. JSR A/CD8A to evaluate the expression following SYS. If it's a string expression, display TYPE MISMATCH. Leave (7A) pointing to the character that stopped the expression evaluation. This character is normally the end-of-line $00 byte or the end-of-statement colon. However, if you specified SYS 3350,M,N then the (7A) text pointer would be pointing to the comma (,) following 3350.
2. JSR B/D7F7 to display ILLEGAL QUANTITY if the number is negative or greater than 65535. Thus only valid addresses are acceptable. Convert the FAC1 value to a two-byte address in 14 and 15.
3. Push $E1 and $46 (Commodore 64) or $E1 and $43 (VIC) onto the stack, so that when the SYS machine language routine does its RTS, control will return to step 10 at E147/E144.

4. Load the accumulator from 030F, the SAVE area for the status register and PHA.
5. Load the accumulator from 030C, the SAVE area for the accumulator.
6. Load the X-register from 030D, the SAVE area for the X-register.
7. Load the Y-register from 030E, the SAVE area for the Y-register.
8. PLP to restore the SAVEd status.
9. JMP (0014). This indirect jump uses the memory location at (0014) as the new program counter. Presumably you have your machine language program at that address. You could also be doing a SYS to a ROM address in BASIC or the Kernal.
10. PHP to save the status that was returned from the machine language program.
11. Store the accumulator in 030C.
12. Store the X-register in 030D.
13. Store the Y-register in 030E.
14. PLA to get this status and STA 030F.
15. RTS.

USR
0310/0000-0312/0002

Called by: Invoke Function for USR.

Entry Conditions:
FAC1 contains the evaluation of the expression within parentheses for USR. This expression can be either a numeric expression or a string expression. If it's a string expression, (64) holds a pointer to the string descriptor for this string expression.

Operation:
1. JMP 0310/0000.
2. JMP (default instruction of direct mode JMP unless you modify it) to the address specified in (0311)/(01). If you have not put an address in (0311)/(01), the default address is B/D248, which displays the error message ILLEGAL QUANTITY.

   Normally you place an address in (0311)/(01) (low-order byte/high-order byte form) that points to the location of your machine language program. Your program can then use the parameter data passed in FAC1. When your machine language program returns by doing an RTS, the FAC1 result from your program is passed back.

Execution continues after your RTS with step 20 of Function Invocation. Function Invocation checks to see that 0D still indicates that the value in FAC1 is a numeric value. If not, TYPE MISMATCH is displayed. Since function invocation does not check the argument type for string or numeric before calling the function, you can receive arguments that are either string or numeric expressions. However, if you receive a string expression, you don't want to return to step 20 of Function Invocation for the type check. Instead, do like the other string functions, such as LEFT$, and pull off the two bytes of the return address that point to step 20. If you have created a new string, you will probably also want to create a new string descriptor for this string and push the string descriptor onto the temporary string descriptor stack. See the LEFT$, RIGHT$, and MID$ routines for how to do this.

Check for ASCII A-Z in Accumulator

This routine is used to determine whether the accumulator contains an ASCII value for A–Z. If the carry is set at exit, the accumulator did contain A–Z at entry.

Check for ASCII A-Z in Accumulator B/D113–B/D11C

Called by:
JSR at A/CE92 Evaluate Next Operand of Expression; JSR at B/D097, B/D9AA, B/D085 Create/Locate Variable.

Operation:

1. At entry the accumulator contains the value to be checked for A–Z. Compare the accumulator to $41, the ASCII code for A.
2. If it's less than A, branch with carry clear to step 5 to RTS.
3. SBC $5B, the ASCII code for the character one above Z. If carry is clear after subtraction, a borrow was required and the accumulator was >= $5B.
4. SEC and SBC $A5 so that the accumulator remains unchanged and the carry is given its correct value. Note that $A5 + $5B = $00.
5. RTS. If the carry is set, the accumulator had an ASCII value of A through Z at entry.

Example if the accumulator had T, ASCII $54, at entry

```
T =  $54   = 0101 0100
SBC  $5B   = 1010 0101  (two's comp form)
             -----------
             0 1111 1001

SBC  $A5   = 0101 1011  (two's comp form)
             -----------
             1 0101 0100 <= accumulator unchanged and carry set
```

Check If in Direct Mode

This routine is used to determine whether direct mode or program mode is active. The GET/GET#, INPUT/INPUT#, and DEF commands use it to display ILLEGAL DIRECT if you try to enter these commands in direct mode. No other commands cause an error if entered from direct mode.

Check If in Direct Mode
B/D3A6–B/D3B0

Called by:
JSR at A/CB7B GET/GET#; JSR at A/CBCE INPUT/INPUT#; JSR at B/D3B6 DEF.

Operation:

1. Load the X-register from 3A. 3A is set to $FF by the Main BASIC Loop routine whenever a line is entered without a line number.
2. Increment the X-register so it's now zero if in direct mode.
3. If the X-register is not zero, exit since currently in program mode.
4. Display ILLEGAL DIRECT error message and return to the Main BASIC Loop.

Get Expression at Next Text Pointer Location

This routine is used to move the text pointer (7A) past the current character and then to evaluate a following expression into an integer from 0 to 255 which is returned in the X-register.

Get Expression at Next Text Pointer Location B/D79B-B/D79D

Called by: JSR at A/CAFF TAB/SPC.

Operation:

1. This routine is called by TAB/SPC to move the text pointer past the TAB or SPC token and then to evaluate the following expression.

   JSR CHRGET to increment the text pointer and return with the next character in the accumulator.
2. Fall through to B/D79E to evaluate the expression that starts at the text pointer, returning with an integer value in FAC1 in the range 0–255. Display ILLEGAL QUANTITY if it's not in this range. Display TYPE MISMATCH if it's a string expression.

Issue IN Line Number Message

This routine is used by the error message and BREAK message routines to display the message IN and the current line number.

Issue IN Line Number Message B/DDC2–B/DDCC

Called by: JSR at A/C471 ERROR/BREAK Message Routine.

Operation:

1. JSR B/DDDA to JMP to A/CB1E to display the message IN.
2. Load the accumulator from 3A. Load the X-register from 39. (39) = line number of the last BASIC line that was executed when the error or BRK occurred.
3. Fall through to B/DDCD to convert this number into normalized floating point format in FAC1. Next convert the floating point format to an ASCII string. Then print the ASCII string.

Syntax Check for Parentheses, Comma, or ASCII Character in Accumulator

Depending on the entry point, this routine does a check to see if the text pointer (7A) is currently pointing to the requested character. If it is, then JMP CHRGET to retrieve the following nonblank character into the accumulator. If the text pointer is not pointing to the requested character, display SYNTAX ERROR.

Syntax Check for Requested Character A/CEF7–A/CF07

Called by: 22 routines at the various entry points.

Operation:

1. A/CEF7: Check for). Load the accumulator with $29, the ASCII code for), and use a BIT instruction to fall through to step 4.
2. A/CEFA: Check for (. Load the accumulator with $28, the ASCII code for (, and use a BIT instruction to fall through to step 4.
3. A/CEFD: Check for ,. Load the accumulator with $2C, the ASCII code for , and fall through to step 4.
4. A/CEFF: Check for an ASCII character in the accumulator.

   Compare the accumulator to (7A),0 to see if the text pointer is pointing to the character requested. If not, display SYNTAX ERROR. If it is, then JMP CHRGET to load the accumulator with the next nonblank character and set flags indicating if an end-of-line or end-of-statement is reached.

Error Messages and READY

Whenever an error occurs in BASIC, an error message is displayed. The error returns a number that is used to index into the table of errors at A/C328 to retrieve the address of the error message for this number. The characters in the error message are then sent to the screen with a final character in the error message signified by having its high-order bit on. Also, when an error occurs, the serial bus is cleared and various other resets occur. See the error message handler for details. After the error message is displayed, READY is displayed, and the Main BASIC Loop is reentered to await the next entry from the keyboard.

The error message routine contains a vector at (0300) which you can use to do your own error message handling.

Many routines that find errors call the error message handler directly with the error in the X-register. Other routines will call a short preparatory routine that loads the X-register with the error number and then calls the error message handler. Examples of this latter type of short preparatory routine include the routines at B/D245 to prepare the BAD SUBSCRIPT error, B/D248 to prepare the ILLEGAL QUANTITY error, B/D3AE to prepare the UNDEF'D FUNCTION error, and A/CF08 to prepare the SYNTAX error.

Error Message Handler A/C437-A/C468

Called by: 17 routines.

Operation:

1. JMP (0300) to the error message handler routine whose address is in the vector at (0300). Normally for the VIC this is C43A, and for the Commodore 64 it is E38B.
2. E38B (Commodore 64 only):
   Transfer the X-register to the accumulator.
   If the high-order bit is on, then JMP A474 to display READY.

If the high-order bit is off, then JMP A43A to handle the error message.

3. A/C43A: The error message index is in the X-register at entry. Multiply this value by two and leave the result in the X-register. Each error message table entry contains a two-byte address.
4. Load (22) from the table containing addresses of error messages that is found starting at A/C328. Since no error number of zero should occur, there is no error message address for such an error. Since the first entry in the table is for error message 1, use A/C326 as the base when loading the low-order byte of the address, and A/C327 when loading the high-order byte.
5. JSR FFCC to the Kernal CLRCHN routine to clear any activity on the serial bus and then to set the input device to be the keyboard and the output device to be the screen.
6. Reset 13, the I/O file number, to zero.
7. JSR A/CAD7 to send a carriage return.
8. JSR A/CB45 to display ?.
9. Now display the message by loading each character that is addressed by (22) and, making sure the high-order bit is 0, then JSR A/CB47 to print the character to the screen. When finally a character with its high-order bit on is loaded from the string at (22), the message is complete (after sending this final character with its high-order bit off).
10. JSR A/C67A to reset the stack pointer to $FA, reset the temporary string descriptor stack pointer to $19 to point to the first entry, disallow CONT, and allow subscripts.
11. Load the accumulator and Y-register with A/C369 to point to the error message.
12. Fall through to A/C469 to display ERROR, possibly followed by IN and the line number. Then display READY and continue with the Main BASIC Loop.

Display IN Line Number Message If Program Mode A/C469–A/C473

Called by:
Fall through from Error Message Handler; JMP at A/C851 STOP.

Operation:

1. JSR A/CB1E to display ERROR.
2. If not in direct mode (if 3A is not $FF), JSR B/DDC2 to display IN and the line number of the last statement executed.
3. Fall through to A/C474 to display READY and go to the Main BASIC Loop.

Display READY
A/C474-A/C47F

Called by:

Fall through from Error Message Handler and Display IN Line Number; JMP at A/C714 LIST; JMP at A/C854 STOP/END; JMP at E472 VIC Warm Start; JMP at E384 VIC Cold Start; JMP at E391 Commodore 64 Cold or Warm Start.

Operation:

1. Point the accumulator and Y-register to the READY message, and JSR A/CB1E to display the message on the current output device.
2. JSR FF90 to the Kernal SETMSG routine to allow control messages.
3. Fall through to the Main BASIC Loop at A/C480.

Polynomial Computation

A polynomial consists of an expression such as the following example:

$5X^3 - 7X^2 + 3X + 28$

The highest exponent value of the variable is the order of the polynomial. As you can see from the example, the number of terms of the polynomial is the order plus one.

The way almost everyone is taught to do polynomials is shown in the following example, using the polynomial example above with an X value of 3:

```
= 5(3*3*3) − 7(3*3) + 3(3) + 28
= 5(27)    − 7(9)   + 3(3) + 28
=   135    − 63     + 9    + 28
=   109
```

This conventional method is not the fastest way of computing polynomials. For each term, except for X to the power of 0 or of terms with coefficients of zero, you have to compute the power of X. In addition to computing the powers of X, if the order of the polynomial is N, then N − 1 multiplications are required and N additions.

Another much faster method that requires only N multiplications and N additions is used in BASIC. This routine is based on Horner's Rule, which was stated by W.G. Horner in 1819. Horner's Rule can be pictured as follows, using the same example as before with X again = 3:

$(((5X - 7)X + 3)X) + 28$

The actual computation using Horner's method is:

```
3*5 = 15
15 − 7 = 8
8*3 = 24
24 + 3 = 27
27*3 = 81
81 + 28 =109
```

The various routines that use this polynomial computation routine all use tables that contain the coefficients to be used in the computation. The table is preceded by a one-byte field that has the order of the polynomial, with the order one less than the number of values in the table. The values immediately follow the order byte, with the values arranged from the coefficient for the highest-order term to the coefficient for the lowest-order term. Each coefficient occupies five bytes and is stored in the standard floating point variable format with the high-order bit of the fraction serving as the sign bit.

The only routine that directly calls polynomial computation is EXP. The others (LOG, SIN/COS/TAN, ATN) call a routine which in turn calls this one. The routine that LOG, SIN/COS/TAN, and ATN call is Evaluate Series for (X Squared)*X. This routine takes the value it receives in FAC1 and stores it in a temporary floating point variable. Then FAC1 is squared and the result is used as the argument for calling the Compute Polynomial Using Horner's Rule routine. Upon return from the polynomial computation, the FAC1 result is multiplied by the saved original value of FAC1.

The series values that are actually used by the various routines are optimized to give the most accurate answer with a reasonably small order of the polynomial. When certain series are computed, the value produced by the series starts converging towards a limiting value. As more terms of the series are computed, the convergence gets closer to the limiting value. When sufficient accuracy has been obtained the series evaluation can stop. The longest series used is ATN which has a polynomial of order 11.

To see an example of why series are optimized, consider the following calculation of EXP(2) and compare its result after eight terms of the computation to the result you get from BASIC for EXP(2) of 7.3890561. BASIC also uses only eight terms, but its series is optimized.

Here is the standard unoptimized series for EXP(X):

$$\mathrm{EXP}(X) = 1 + X + (X^2/2!) + (X^3/3!) + (X^4/4!) + (X^5/5!) + (X^6/6!) + (X^7/7!)$$

With X = 2 this becomes:

$$\mathrm{EXP}(2) = 1 + 2 + \frac{4}{2} + \frac{8}{6} + \frac{16}{24} + \frac{32}{120} + \frac{64}{720} + \frac{128}{5040}$$

$$\mathrm{EXP}(2) = 7.38095238$$

Compare this unoptimized answer to the optimized answer that EXP(2) gives of 7.3890561 and you can see that the unoptimized series needs more terms to produce as accurate an answer.

Discussions of the actual techniques that are used to optimize the series can be found in books on numerical methods and numerical analysis. The topic of how optimization works is beyond the scope of this book. What I'll try to show in the various series for SIN, ATN, etc., is the standard series for the function and the optimized series used in BASIC. Also, the range for which the series is optimized is sometimes given.

One of the nice features about the polynomial computation is that you can use it to execute a polynomial that you might want to define. For example, if you had a polynomial that computed $25X^3 - 17.3X^2 + 88X + 5$, you would place the order three byte at the beginning of your series, followed by the five-byte floating point variables for 25, −17.3, 88, and 5. An example of how to do this is found in the section on direct use of the floating point routines.

Polynomial Computations Using Horner's Rule E059/E056-E08C/E089

Called by:
JSR at E037/E034 EXP; Alternate E05D/E05A: JSR at E04F/E04C Evaluate Series for FN(X Squared)*X.

Operation:

1. Store the accumulator in 71 and the Y-register in 72 to set a pointer to the order byte for the EXP series values.
2. E05D/E05A: If entered from the JSR for Evaluate Series for FN(X Squared)*X then (71) already points to the order byte of the series values to be used.

   JSR B/DBC7 to copy FAC1 to 5C–60. This value will be used to add the FAC1 value when doing the Horner's Rule computation in steps 4–9.
3. Load the accumulator from this order byte and then save the accumulator in 67. The order is one less than the number of values in the series.

   Now set the accumulator and Y-register to (71) plus one to point to the first five-byte floating point number for this series. FAC1 contains whatever was in FAC1 at entry to this routine.

4. JSR B/DA28 to multiply FAC1 by the number pointed to by the accumulator and Y-register, leaving the result in FAC1.
5. Add 5 to (71) and store the result back into (71) so that it now points to the next series value.
6. JSR B/D867 to add the number pointed to by (71) to FAC1.
7. Load the accumulator with $5C and the Y-register with $00 to point to the original saved value of FAC1.
8. Decrement 67, which was initialized to the order of the polynomial.
9. If 67 is not zero, branch to step 4.
10. When 67 becomes zero, the final result of this polynomial computation is in FAC1. RTS.

Evaluate Series for (X Squared) E043/E040–E058/E055

Called by:
JSR at B/DA16 LOG; JSR at E328/E325 ATN; JSR at E2B1/E2AE SIN.

Operation:
1. Store the accumulator in 71 and the Y-register in 72 to set a pointer to the order byte.
2. JSR B/DBCA to copy FAC1 to 57–5B and return with the Y-register = 0.
3. Load the accumulator with $57.
4. JSR B/DA28 to copy the floating point number at 57–5B to FAC2, and then multiply FAC2 by FAC1, leaving the result in FAC1. Thus what has happened here is that the original value in FAC1 is now squared.
5. JSR E05D/E05A to do the series evaluation on FAC1 using the series pointed to by (71). Return with the result of the series evaluation in FAC1.
6. JMP B/DA28 to copy 57–5B to FAC2, then multiply FAC2 by FAC1 and leave the result in FAC1. What this routine has thus done is to perform a series evaluation on FAC1 squared and to multiply the result by the original value of FAC1. If you represent FAC1 by X and the series evaluation as a function, you could picture what happens as follows:

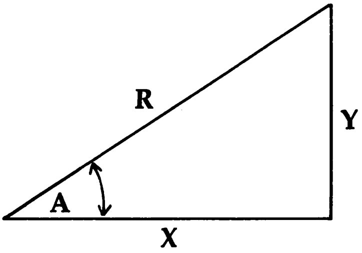
   X = F(X) * X

Example of How to Display Floating Point Variables for a Series: LOG on the Commodore 64

```
1Ø S = 47553
2Ø N = PEEK(S)
3Ø FOR Z = Ø TO N
4Ø C = Z*5
5Ø X = S+1+C
6Ø XP = PEEK(X)
7Ø F1 = PEEK(X+1)
8Ø F2 = PEEK(X+2)
9Ø F3 = PEEK(X+3)
1ØØ F4 = PEEK(X+4)
11Ø W=(-1)↑(F1AND128)*2↑(XP-129)*(1+((F1AND127)+(F
    2+(F3+F4/256)/256)/256)/128)
13Ø IF F1 >= 128 THEN W = -W
14Ø PRINTW
15Ø NEXT
```

Sine, Cosine, and Tangent

The sine, cosine, and tangent are all computed based on the sine of an angle. The cosine of an angle is equal to the sine of that angle plus 90 degrees. The tangent of an angle is the sine divided by the cosine. The meanings for sine, cosine, and tangent are simply demonstrated by picturing a triangle:



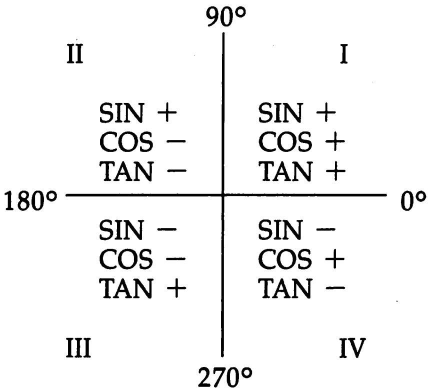
SIN (A) = Y/R
COS (A) = X/R
TAN (A) = Y/X

In BASIC, the arguments of these trig functions are given in radian measurement rather than in degrees. A radian is the measure of the central angle of a circle that subtends (extends below) an arc of that circle with the length of the arc equal to the radius.

Since the sine of any angle above 360° is equal to the sine of the corresponding angle less than 360°, the latter is actually used in computing the sine (for example, the sine of 370° is the same as the sine of 10°). To convert the angle to a value less than 360°, mod 360 arithmetic is used in which the angle is divided by 360°. 360° is equivalent to 2*PI radians, which is the actual value used to divide into the original argument which is specified in radians. Although the trig routines work in terms of radians, I find it easier to relate to degrees and thus the examples in this section all use degrees.

In discussions of trig functions, you often find a rectangular coordinate system used. The following diagram shows the quadrants in this system and the signs of the TAN, SIN, and COS functions in the various quadrants. The computations for SIN, COS, and TAN all correctly set the sign.

Quads and Signs of Functions



Rather than look up the values of the SIN in a table, a series is used to calculate values for the SIN. The series upon which the SIN computation is based is:

$$\mathrm{SIN}(X) = X - \frac{X^3}{3!} + \frac{X^5}{5!} - \frac{X^7}{7!} + \frac{X^9}{9!} \quad \ldots$$

The ! is the factorial symbol which means that 5! = 1 * 2 * 3 * 4 * 5 and 7! = 1 * 2 * 3 * 4 * 5 * 6 * 7.

What BASIC has done is to go ahead and compute the value of the SIN using this series for an X value of 360 degrees (or 2*PI). The SIN routine then relates its argument by a ratio to this 360 degrees (or 2*PI) and computes the series for SIN by running the ratio through the series using the precalculated values of the SIN for 360 degrees (or 2*PI).

SIN
E26B/E268–E2B3/E2B0

Called by:
Invoke Function for SIN; Alternate E29D/E29A: JMP at E2DD/E2DA.

Operation:

1. At entry from function invocation, FAC1 contains the radian value from inside the parentheses.
   JSR B/DC0C to copy FAC1 to FAC2 with rounding.
2. Load the accumulator and the Y-register with values to point to the floating point number for 2*PI.
3. Load the X-register from 6E, the sign of FAC2.

4. JSR B/DB07 to divide FAC2 by 2*PI and leave the result in FAC1. 2*PI radians is the same as 360°, which is a complete circle. All values greater than 360° have the same SIN values as their corresponding value less than 360° (for example, the sine of 380° is the same as the sine of 20°, and the sine of 810° is the same as the sine of 90°).
5. JSR B/DC0C to copy FAC1 to FAC2 with rounding so that FAC2 now contains the value of the number of radians after dividing by 360°.
6. JSR B/DCCC to round FAC1 down to the nearest integer, but leave the result in floating point format. If the exponent of FAC1 is >=32, leave FAC1 unchanged since it is already in integer format.
7. Set 6F to zero.
8. JSR B/D853 to subtract FAC1 from FAC2, leaving a result between zero and one in FAC1. This result is a ratio of the angle to 360°. For example:

```
450/360 = 1.25 1.25 − 1 = .25
 90/360 =  .25  .25 − 0 = .25
```

9. Point the accumulator and Y-register to the floating point representation of .25.
10. JSR B/D850 to move .25 to FAC2 and then subtract FAC1 from FAC2, leaving the result in FAC1.

    Here are some examples for the SIN in each of the four quadrants:

Quad and Degrees		Value After Step 8	Value After Step 10	Sign (66) After Step 10
I	30	.083	+.167	+
II	120	.333	−.083	−
III	210	.583	−.333	−
IV	300	.833	−.583	−

11. Push the sign of FAC1, 66, from step 10 onto stack.
12. If the sign is positive, then it's in the first quadrant, so branch to step 16.
13. If it's a negative sign, then in quadrants II, III, or IV. JSR B/D849 to add .5 to FAC1, with the following example using the same original angles as the previous example.

Quad and Degrees		Value After Step 10	Value After Step 13	Sign(66) After Step 13
II	120	−.083	+.417	+
III	210	−.333	+.167	+
IV	300	−.583	−.083	−

14. If 66 is negative then branch to step 17, thus branching if in quadrant IV.
15. For a positive sign in 66 the flag in 12 is Exclusive OR'd with $FF. 12 is initialized to $00 by TAN. This saved sign of the SIN is used together with the sign of the COS to set the sign of the TAN.
16. E29D/E29A: Entry point for computing COS during the TAN routine.

    JSR B/DFB4 to reverse the sign of FAC1.
17. Point accumulator and Y-register to the floating point value of .25.
18. JSR B/D867 to copy .25 to FAC2 and then add FAC2 to FAC1. Here are the results after steps 16 and 18 using the same angles as in the previous examples:

Quad and Degrees		Value After Step 16	Value After Step 18	Sign(66) After Step 18
I	30	−.167	+.083	+
II	120	−.417	−.167	−
III	210	−.167	+.083	+
IV	300	−.083	+.167	+

19. Pull the sign pushed onto the stack in step 11.
20. If this sign is positive, branch to step 22. The branch is taken for angles from quadrant I.
21. If it's a negative sign, JSR B/DFB4 to reverse the sign of FAC1. After this step the quadrant II sign is positive while quadrants III and IV are negative. Thus the signs for the SIN for all four quadrants have now been correctly determined.
22. Point the accumulator and Y-register to the series for SIN.
23. JMP E043/E040 to compute the series for the SIN.

COS
E264/E261–E26A/E267

Called by: Invoke Function for COS.

Operation:

1. Point the accumulator and Y-register to the floating point value of PI/2. Since 2*PI is 360°, PI/2 is 90°.
2. JSR B/D867 to copy PI/2 to FAC2 and then add FAC2 to FAC1.
3. Fall through to the SIN Routine to compute the value for the SIN of FAC1. The SIN routine can be used to determine the value of the cosine since COS(X) = SIN(X+90°).

TAN
E2B4/E2B1–E2DB/E2D8

Called by: Invoke Function for TAN.

Operation:

1. JSR B/DBCA to copy FAC1 to 5C–60.
2. Set 12, the indicator of the sign of the SIN, to zero.
3. JSR E26B/E268 to determine the SIN of FAC1 for TAN, leaving the result in FAC1.
4. Set the pointer in the X-register and Y-register to 004E.
5. JSR E0F6/E0F3 which JMPs to B/DBD4 where FAC1 is copied to 4E–52, which now holds the result of computing the SIN.
6. Set the pointer in the X-register and Y-register to 0057. At 0057 is stored the argument that was passed by the SIN routine to the series evaluation routine. This value can be reused in computing the COS as long as the equivalent of 90° is added.
7. JSR B/DBA2 to copy 57–5C to FAC1.
8. Set 66, the sign of FAC1, to zero.
9. Load the accumulator from 12, which was set to $FF if the SIN was determined for an angle in the third quadrant.
10. JSR E2DC/E2D9 to compute the COS of the angle + 90° for TAN, leaving the result in FAC1.
11. Point the accumulator and Y-register to 004E.
12. JMP B/DB0F to copy the SIN value from 004E to FAC2 and then divide FAC2 by FAC1. Since the TAN = SIN/COS, the result in FAC1 now contains the value for the TAN. The sign of TAN is also correctly set. If the SIN and COS have like signs (quadrants I and III), the sign of the TAN is positive. For unlike signs of the COS and SIGN (quads II and IV), the TAN is negative.

Compute COS for TAN
E2DC/E2D9-E2DF/E2DC

Called by: JSR at E2D2/E2CF TAN.

Operation:
1. PHA to save the 12 value from SIN.
2. JMP E29D/E29A to compute the COS by using the same value used to compute the SIN and adding .25, the equivalent of 90°. The sign of the COS is also determined.

ATN

ATN is an inverse trig function for TAN. Consider this example: TAN(1.5) = 14.10142; ATN(14.10142) = 1.5. TAN is given an angle and returns a ratio, while ATN is given a ratio and determines an angle. The series upon which the ATN computation is based is:

$$\text{ATN}(X) = X - \frac{X^3}{3} + \frac{X^5}{5} \ldots + \frac{X^{11}}{11} \ldots \text{if absolute value of } X < 1$$

Because the denominator of this series does not contain a factorial, the series converges more slowly than the series for SIN and thus 12 terms are used to obtain sufficient accuracy. The ATN series has been computed using a value for X of 1 (the tangent of 45°). The ATN routine determines whether the tangent argument is less than 1. If not, the reciprocal of the argument is computed. Then either the reciprocal or the original value of < 1 is run through the series using the precalculated values of the ATN for 1. The answer returned is an angle from −90° to +90° (of course expressed in radians).

ATN
E30E/E30B-E33D/E33A

Called by: Invoke Function for ATN.

Operation:

1. Load the accumulator from 66, the sign of FAC1, and PHA. If it's a value from the first or third quadrant, the sign is negative.
2. If it's a positive sign, branch to step 4.
3. For a negative sign, JSR B/DFB4 to reverse the sign of FAC1.
4. Load the accumulator from 61, the exponent of FAC1, and PHA.
5. Compare the exponent to $81. If the exponent is less than $81, the argument specified was for an ATN of 0 to 1 (or less than 45°), so branch to step 8.
6. For ATN of >= 1, point the accumulator and Y-register to the floating point value of 1.

7. JSR B/DB0F to copy 1 to FAC2 and then divide FAC2 by FAC1, leaving the reciprocal of the argument to ATN in FAC1.
8. JSR E043/E040 to do the series evaluation using FAC1. Return the result in FAC1.
9. PLA to pull the original exponent value. If >$81, branch to step 11 since no readjustment is needed.
10. For those values that passed a reciprocal to the series evaluation, now point the accumulator and Y-register to PI/2. PI/2 is equivalent to 90°. Then JSR B/D850 to copy PI/2 to FAC2 and subtract FAC1 from FAC2.
11. PLA to retrieve the original sign of the argument. If it's positive, exit.
12. For a negative argument, JMP B/DFB4 to negate the sign of FAC1.

Consider the following example for ATN(1.7321) which is equal to 1.04720985, or an angle of 60°. This argument had to be converted to its reciprocal:

ATN (1.7321)
FAC1 = 1.7321
1/1.7321 = .5773
ATN(.5773) = 30°
Then step 10 does 90° − 30° = 60°

EXP

EXP(X) computes e^x with e = 2.718281828..., the base of the natural or Naperian system of logarithms.

The series upon which the EXP computation is based is:

$$EXP(X) = 1 + X + \frac{X^2}{2!} + \frac{X^3}{3!} + \ldots \text{ for all values of } X$$

The EXP series has been computed using a value for X of LOG(2). The argument to EXP is divided by LOG(2). The exponent of the quotient is saved. This exponent is what power you need to raise 2 to reach this number. The remainder of this division is sent through the series and returns a result in FAC1. The saved exponent of the quotient is then added back to the exponent in FAC1. The EXP of a number is thus determined by relating the number to 2 and using the series that has already been computed for a value of LOG(2).

EXP
BFED–BFFF and E000/E042(Commodore 64)/DFED–E03F (VIC)

Called by: Invoke Function for EXP.

On the Commodore 64 the code for EXP jumps from BFFD to E000 since BASIC is not contiguous. The extra JMP to get to E000 adds three bytes to BASIC which explains why the code from E000 on is usually three bytes higher than its counterpart on the VIC.

Operation:

1. Point the accumulator and Y-register to the floating point variable for 1/LOG(2).
2. JSR B/DA28 to multiply the argument for EXP in FAC1 by 1/LOG(2), thus obtaining the power to which 2 must be raised to be the argument number.
3. Add $50 to 70, the rounding byte for FAC1, and if the carry is set as a result, then JSR B/DC23 to round up FAC1 by adding one to the low-order bit of the fraction for FAC1.
4. Store the accumulator in 56.
5. JSR B/DC0F to copy FAC1 to FAC2.

6. If the exponent of FAC1 is >= $88, display OVERFLOW ERROR and return to the Main BASIC Loop.
7. JSR B/DCCC to perform INT on FAC1, leaving FAC1 as an integer but still represented in floating point format.
8. Load the accumulator from 07, the low-order byte of the integer computed by INT. If adding $81 gives an answer of zero, then again an OVERFLOW ERROR occurs.
9. Subtract 1 from the accumulator to restore the excess-128 exponent value and push it onto the stack.
10. Now swap FAC1 (the integer portion) and FAC2 (the real number value).
11. JSR B/D853 to subtract FAC1 (the real number portion) from FAC2 (the integer portion), leaving a negative value from 0 to 1.
12. JSR B/DFB4 to negate FAC1.
13. Point the accumulator and Y-register to the series for EXP.
14. JSR E059/E056 to compute the series for EXP on this fractional part, returning the value in FAC1.
15. Set 6F to zero.
16. Pull the power of two to the original argument from the stack.
17. JSR B/DAB9 to add this power of two to the exponent of FAC1.

LOG

LOG(X) computes the logarithm of X to the base e with e = 2.718281828.... LOG(X) is the opposite of EXP(X). For example, LOG(148.413159) = 5 and EXP(5) = 148.413159.

The series upon which the LOG computation is based is:

$$\mathrm{LOG(X)} = 2 * \left[\left(\frac{X-1}{X+1}\right) + \frac{1}{3} * \left(\frac{X-1}{X+1}\right)^3 + \frac{1}{5} * \left(\frac{X-1}{X+1}\right)^5\right]$$

The LOG series has been computed using a value for X of LOG(128). Numbers are represented in floating point format using an excess-128 exponent.

LOG
B/D9EA-B/DA27

Called by: Invoke Function for LOG.

Operation:

1. JSR B/DC2B to determine the sign of FAC1 and whether FAC1 is zero. Return with the accumulator = 0 if FAC1 is zero, = $FF if FAC1 is negative, or = $01 if FAC1 is positive.
2. If it's negative or zero, display ILLEGAL QUANTITY and return to the Main BASIC Loop.
3. If it's positive, load accumulator from 61, the exponent of FAC1, and subtract $80 to remove the excess-128 notation. Push the result onto the stack.
4. Set 61 to $80 to force FAC1 to a value from 0 to 1.
5. Point the accumulator and Y-register to a floating point value of 1/SQR(2), or 1/1.414.
6. JSR B/D867 to add 1/1.414 to FAC1. I let X represent FAC1 in the following examples. This step thus results in (1.414X + 1)/1.414.
7. Point the accumulator and Y-register to SQR(2) or 1.414.
8. JSR B/DB0F to divide SQR(2) by FAC1, or continuing our example, now we have (1.414 * 1.414)/(1.414X+1) after actually doing the division.
9. Point the accumulator and Y-register to the floating point value of 1.

10. JSR B/D850 to subtract FAC1 from 1. Our example is now (1.414X−1)/(1.414X+1).
11. Point the accumulator and Y-register to B/D9C1, the series for LOG.
12. JSR E043/E040 to evaluate the series.
13. Point the accumulator and Y-register to −.5, which is an approximation of (LOG(LOG2))/LOG(2).
14. JSR B/D867 to add −.5 to FAC1.
15. PLA to retrieve the original exponent of FAC1 (minus the excess-128) from the stack that was pushed there in step 3.
16. JSR B/DD7E to add the exponent to FAC1.
17. Point the accumulator and Y-register to LOG(2).
18. Fall through to B/DA28 to multiply FAC1 by LOG(2).

Appendices

Appendix A

ASCII Codes

ASCII	CHARACTER	ASCII	CHARACTER
5	WHITE	50	2
8	DISABLE	51	3
	SHIFT COMMODORE	52	4
9	ENABLE	53	5
	SHIFT COMMODORE	54	6
13	RETURN	55	7
14	LOWERCASE	56	8
17	CURSOR DOWN	57	9
18	REVERSE VIDEO ON	58	:
19	HOME	59	;
20	DELETE	60	<
28	RED	61	=
29	CURSOR RIGHT	62	>
30	GREEN	63	?
31	BLUE	64	@
32	SPACE	65	A
33	!	66	B
34	"	67	C
35	#	68	D
36	$	69	E
37	%	70	F
38	&	71	G
39	'	72	H
40	(	73	I
41	)	74	J
42	*	75	K
43	+	76	L
44	,	77	M
45	–	78	N
46	.	79	O
47	/	80	P
48	0	81	Q
49	1	82	R

ASCII	CHARACTER	ASCII	CHARACTER
83	S	120	♣
84	T	121	▕
85	U	122	♦
86	V	123	┼
87	W	124	▒
88	X	125	│
89	Y	126	π
90	Z	127	◥
91	[	129	ORANGE
92	£	133	f1
93	]	134	f3
94	↑	135	f5
95	←	136	f7
96	─	137	f2
97	♠	138	f4
98	│	139	f6
99	─	140	f8
100	▔	141	SHIFTED RETURN
101	▔	142	UPPERCASE
102	▁	144	BLACK
103	▏	145	CURSOR UP
104	▕	146	REVERSE VIDEO OFF
105	╮	147	CLEAR SCREEN
106	╰	148	INSERT
107	╯	149	BROWN
108	□	150	LIGHT RED
109	╲	151	GRAY 1
110	╱	152	GRAY 2
111	□	153	LIGHT GREEN
112	□	154	LIGHT BLUE
113	●	155	GRAY 3
114	▁	156	PURPLE
115	♥	157	CURSOR LEFT
116	▏	158	YELLOW
117	╭	159	CYAN
118	╳	160	SHIFTED SPACE
119	○	161	▌

ASCII	CHARACTER	ASCII	CHARACTER
162	▄	200	▕
163	□	201	╮
164	□	202	╰
165	□	203	╯
166	▒	204	□
167	□	205	╲
168	▒	206	╱
169	◤	207	□
170	▐	208	□
171	├	209	●
172	▗	210	□
173	└	211	♥
174	┐	212	▏
175	▂	213	╭
176	┌	214	╳
177	┴	215	○
178	┬	216	♣
179	┤	217	▕
180	▎	218	♦
181	▍	219	┼
182	▐	220	▒
183	▔	221	│
184	▀	222	π
185	▃	223	◥
186	□	224	SPACE
187	▖	225	▌
188	▝	226	▄
189	┘	227	□
190	▘	228	□
191	▚	229	□
192	─	230	▒
193	♠	231	□
194	│	232	▒
195	─	233	◤
196	─	234	▐
197	▔	235	├
198	▁	236	▗
199	▏	237	└

ASCII	CHARACTER
238	┐
239	▃
240	┌
241	┴
242	┬
243	┤
244	▎
245	▍
246	▐
247	▔
248	▀
249	▃
250	▕
251	▖
252	▝
253	┘
254	▘
255	π

Note: 0–4, 6, 7, 10–12, 16, 21–27, 128, 130–132, and 143 are not used.

The VIC-20 does not use 149–155.

Appendix B

Screen Codes

POKE	Uppercase and Full Graphics Set	Lower- and Uppercase	POKE	Uppercase and Full Graphics Set	Lower- and Uppercase
0	@	@	31	←	←
1	A	a	32	-space-	
2	B	b	33	!	!
3	C	c	34	"	"
4	D	d	35	#	#
5	E	e	36	$	$
6	F	f	37	%	%
7	G	g	38	&	&
8	H	h	39	'	'
9	I	i	40	(	(
10	J	j	41	)	)
11	K	k	42	*	*
12	L	l	43	+	+
13	M	m	44	,	,
14	N	n	45	–	–
15	O	o	46	.	.
16	P	p	47	/	/
17	Q	q	48	0	0
18	R	r	49	1	1
19	S	s	50	2	2
20	T	t	51	3	3
21	U	u	52	4	4
22	V	v	53	5	5
23	W	w	54	6	6
24	X	x	55	7	7
25	Y	y	56	8	8
26	Z	z	57	9	9
27	[	[	58	:	:
28	£	£	59	;	;
29	]	]	60	<	<
30	↑	↑	61	=	=

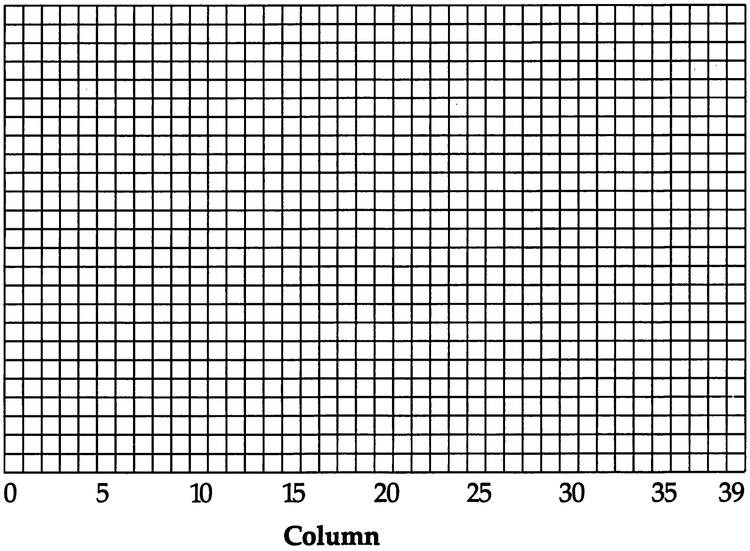
Appendix B

POKE	Uppercase and Full Graphics Set	Lower- and Uppercase	POKE	Uppercase and Full Graphics Set	Lower- and Uppercase
62	>	>	99	▔	▔
63	?	?	100	▁	▁
64	─	─	101	▎	▎
65	♠	A	102	▒	▒
66	│	B	103	▕	▕
67	─	C	104	▒	▒
68	─	D	105	◤	▨
69	▔	E	106	▐	▐
70	▁	F	107	├	├
71	▏	G	108	▗	▗
72	▕	H	109	└	└
73	╮	I	110	┐	┐
74	╰	J	111	▂	▂
75	╯	K	112	┌	┌
76	└	L	113	┴	┴
77	╲	M	114	┬	┬
78	╱	N	115	┤	┤
79	┌	O	116	▎	▎
80	┐	P	117	▍	▍
81	●	Q	118	▐	▐
82	─	R	119	▔	▔
83	♥	S	120	▀	▀
84	▏	T	121	▃	▃
85	╭	U	122	┘	✓
86	╳	V	123	▖	▖
87	○	W	124	▝	▝
88	♣	X	125	┘	┘
89	▕	Y	126	▘	▘
90	♦	Z	127	▚	▚
91	┼	┼			
92	▒	▒			
93	│	│			
94	π	▒			
95	◥	▨			
96	-space-				
97	▌	▌			
98	▄	▄			

128–255 reverse video of 0–127

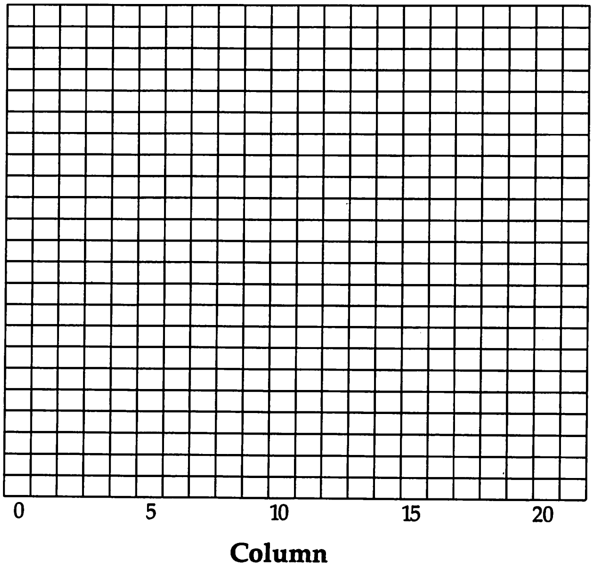
Screen Location Table (64)

Row	
0	1024
	1064
	1104
	1144
	1184
5	1224
	1264
	1304
	1344
	1384
10	1424
	1464
	1504
	1544
	1584
15	1624
	1664
	1704
	1744
	1784
20	1824
	1864
	1904
	1944
24	1984



Screen Location Table (VIC)

Row	
0	7680 (4096)
	7702 (4118)
	7724 (4140)
	7746 (4162)
	7768 (4184)
5	7790 (4206)
	7812 (4228)
	7834 (4250)
	7856 (4272)
	7878 (4294)
10	7900 (4316)
	7922 (4338)
	7944 (4360)
	7966 (4382)
	7988 (4404)
15	8010 (4426)
	8032 (4448)
	8054 (4470)
	8076 (4492)
	8098 (4514)
20	8120 (4536)
	8142 (4558)
22	8164 (4580)



Note: Numbers in parentheses are for VICs with 8K or more of memory expansion.

Appendix D

Keycodes (64)

Key	Keycode	Key	Keycode
A	10	6	19
B	28	7	24
C	20	8	27
D	18	9	32
E	14	0	35
F	21	+	40
G	26	–	43
H	29	£	48
I	33	CLR/HOME	51
J	34	INST/DEL	0
K	37	←	57
L	42	@	46
M	36	*	49
N	39	↑	54
O	38	:	45
P	41	;	50
Q	62	=	53
R	17	RETURN	1
S	13	,	47
T	22	.	44
U	30	/	55
V	31	CRSR ↑↓	7
W	9	CRSR ⇄	2
X	23	f1	4
Y	25	f3	5
Z	12	f5	6
1	56	f7	3
2	59	SPACE	60
3	8	RUN/STOP	63
4	11	NO KEY	
5	16	PRESSED	64

The keycode is the number found at location 197 for the current key being pressed. Try this one-line program:

```
10 PRINT PEEK (197): GOTO 10
```

Values Stored at Location 653

Code	Key(s) pressed
0	(No key pressed)
1	SHIFT
2	Commodore
2	Commodore
3	SHIFT and Commodore
4	CTRL
5	SHIFT and CTRL
6	Commodore and CTRL
7	SHIFT, Commodore, and CTRL

Appendix D

Keycodes (VIC)

Key	Keycode	Key	Keycode
A	17	6	58
B	35	7	3
C	34	8	59
D	18	9	4
E	49	0	60
F	42	+	5
G	19	–	61
H	43	£	6
I	12	CLR/HOME	62
J	20	INST/DEL	7
K	44	←	8
L	21	@	53
M	36	*	14
N	28	↑	54
O	52	:	45
P	13	;	22
Q	48	=	46
R	10	RETURN	15
S	41	,	29
T	50	.	37
U	51	/	30
V	27	CRSR↑↓	31
W	9	CRSR⇄	23
X	26	f1	39
Y	11	f3	47
Z	33	f5	55
1	0	f7	63
2	56	SPACE	32
3	1	RUN/STOP	24
4	57	NO KEY	
5	2	PRESSED	64

The keycode is the number found at location 197 for the current key being pressed. Try this one-line program:

```
10 PRINT PEEK (197): GOTO 10
```

Values Stored at Location 653

Code	**Key(s) pressed**
0	(No key pressed)
1	SHIFT
2	Commodore
2	Commodore
3	SHIFT and Commodore
4	CTRL
5	SHIFT and CTRL
6	Commodore and CTRL
7	SHIFT, Commodore, and CTRL

Index

abbreviations for keywords 104
ABS 21, 109, 135, 189, 235, 248, 252
accumulator
  adding to FAC1 252
  check for A–Z value 380
  convert accumulator and X-register to floating point to ASCII string 247
  convert accumulator and Y-register to floating point 247
  convert accumulator to floating point 246
  convert floating point to integer in accumulator and Y-register 243
  passing to/from SYS 375
  propagate into FAC1 fraction 253–54
  scanning for character in 384
accuracy of floating point arithmetic 190–92
addition of polynomials. *See* Horner's Rule
addition. *See* floating point addition
addition of numeric strings 256
address, convert from floating point to two-byte integer 368–69
allocate space
  for line 115, 120
  for string 122, 129, 142, 259–61
  for variable 120
AND 21, 109, 189, 225–26
arithmetic. *See* floating point addition, subtraction, multiplication, division
arrays 164
  check for redimensioning 172–73
  clearing 133
  computing size 178–79
  creating 179–82
  deleting 160
  descriptor. *See* variable descriptor, array
  dimensions 126
  element size 179
  floating point 164, 179
  integer 164, 179
  maximum elements per dimensions, default 179
  string 164, 179
  variable area 122
ASC 268
ASCII-to-float conversion 226–33
ASCII-to-two-byte-line-number 292
assignment operation. *See* LET
ATN 399–400
autostart cartridges 89
available
  free space 121
  space for variable or line 120, 123
  stack space 120, 122–23, 142
bad READ/INPUT/GET error handling 348–49
bank switching memory 41, 42
BASIC. *See* differences between Commodore 64 and VIC-20
  flaws 99, 190–92, 203–204, 229, 333–34
  input buffer 96–98, 103, 109, 338–41
  lines 97
  major areas 161
  Microsoft, Preface
  mixed with machine language 84
  modifying 41
  pointers to areas 161
block memory moves 123–24
Boolean operations. *See* AND, NOT
bottom of BASIC 84
bottom of memory 84
bottom of stack, used to store strings 233
BRK 89
bugs. *See* BASIC, flaws
Butterfield, Jim 305
carriage return 96, 97, 98, 315, 322
cassette buffer 84
cassette program tapes, relocatable and nonrelocatable 352
channel. *See* I/O channels
chip select. *See* bank switching memory
CHKIN, BASIC's 362
CHKIN, Kernal's 342, 343, 351, 362
CHKOUT, BASIC's 362
CHKOUT, Kernal's 351, 361, 362
choice of seed. *See* RND
CHRGET 99–102
CHRGET, copying to page zero 89–90
CHRGOT 99–102
CHRIN, BASIC's 98, 362
CHRIN, Kernal's 96, 98, 338, 351, 362
CHROUT, BASIC's 361
CHROUT, Kernal's 318, 340, 345, 351, 361
CHR$ 270
CIA chip 372
CLOSE 353–56
CLR 290, 339, 343, 363
CLRCHN 340, 342, 343, 360
CMD 311, 350–51, 361

coefficients of polynomials 388–89
cold start 89, 90, 372
colon to separate statements. *See* end-of-statement
column major order 164
columns on screen 371
command dispatch table. *See* tables, command dispatch
comma
  between parameters 356–57
  in PRINT 311, 315
  preceding input buffer for INPUT 339, 340, 344
compacting memory. *See* garbage collection
comparison, testing for >, >=, =, <=, <, <> 364–65
  numeric 364–67
  string 365–67
compressed text. *See* tokenization
COMPUTE! magazine 305
concatenation 265–67
CONT 280, 281, 288, 295
  after LIST 320
conversions 21
COSINE 393–94, 396–97
  for TAN, 398
creating CHRGET in page zero 89
cursor
  screen 371
DATA 288, 299–300, 338–39, 347–48, 359
Davies, G. Russ Preface, 352
DEF 258, 326–31, 331
delimiters, scan for 300
dependent variable of function 326, 327
descriptor. *See* variable descriptor
detokenization 321, 324
detokenization vector 41
device numbers 350, 354–55, 360
differences between Commodore 64 and VIC-20 315, 353, 372, 373, 375, 377
DIM 164, 172, 180, 182–83
  for floating point variables 164
dimensions of an array 170–83
direct mode 159, 281, 283
  check for 381
  disallowing 328, 340, 342
disallowing array or integer variables 166–67, 328
disallowing CONT 359
e (2.71828183) 401–403
elements of an array 174–78
embedding machine language programs in BASIC statements 84–85
END 288, 294
end-of:
  line 85, 280, 281, 314, 323, 339, 340, 344, 356
  program 283, 323
  statement 280, 282, 328, 356, 357
  string 260
error message vector 385–86
error messages 385–86
  BAD SUBSCRIPT 173, 175
  BREAK 322
  CAN'T CONTINUE 295
  DEVICE NOT PRESENT 351
  DIVISION BY ZERO 218, 220
  EXTRA IGNORED 348
  FILE NOT FOUND 343, 350
  FORMULA TOO COMPLEX 257, 262
  ILLEGAL DIRECT 328, 340, 342, 381
  ILLEGAL QUANTITY 170, 184, 185, 187, 225, 242, 244, 296, 316, 336, 338, 342, 354, 356, 360, 361, 368, 369, 370, 375, 377, 378, 403
  Kernal 363
  LOAD ERROR 358
  NEXT WITHOUT FOR 306
  NOT INPUT FILE 351
  OUT OF DATA 339, 347
  OUT OF MEMORY 178, 179, 181, 304
  OVERFLOW 211, 218, 222, 229, 232, 402
  REDIM'D ARRAY 165, 172, 182
  REDO FROM START 338, 349
  RETURN WITHOUT GOSUB 302
  STRING TOO LONG 266, 345
  SYNTAX ERROR 165, 166, 169, 183, 278, 281, 283, 285, 293, 297, 303, 316, 321, 328, 329, 330, 331, 336, 338, 342, 343, 344, 347, 349, 354, 355, 356
  TOO MANY FILES 350
  TYPE MISMATCH 184, 243, 244, 264, 266, 304, 316, 329, 330, 331, 334, 336, 337, 342, 354, 356, 360, 361, 364, 369, 370, 377, 379
  UNDEF'D FUNCTION 330
  UNDEF'D STATEMENT 287, 290
  VERIFY ERROR 358
evaluation,
  of expressions 184
  of polynomials 388–92
excess-128 notation 163–64, 189–90, 207, 208
execute
  current BASIC statement 284–85
  program 288–89
  token vector 41
EXP 223, 401–402

exponentiation 35, 223–24
exponent of floating point number 163
expressions
  creating array descriptors but not variable descriptors 164–65
  string 256, 311
  string for USR 377
filenames 355, 356, 360
fix-to-float conversion 244–48
float-to-ASCII conversion 22, 233–41
float-to-fixed-point-integer conversion 22, 239–44
floating point
  accumulators 20, 163, 185
  adding exponents for multiplication or division 213–14
  addition 33–34, 196–201
  comparison 253–54
  converting FAC1 to two's complement 250
  complementing FAC1 203
  direct use from machine language 19–40
  division 21, 30–33, 214–21
  inaccuracies 190–92
  moves 192–96
  multiplication 21, 29, 30, 207–14
  normalization 204–205
  numeric expression value 136–37
  overflow 177, 211
  remainder during division 215
  rounding 222
  saving FAC1 on stack 148
  subtraction 21, 33–34, 196–201, 206–207
  underflow 211
  variables 183–88
FN 326–32
FOR 109, 286, 303–305
FOR block 309–10
FRE 109, 267
function descriptor 326–27
function invocation 333, 335–37
functions
  arithmetic 334–37
  dispatch table. *See* tables, function dispatch
  string 334–37
garbage collection 258, 267
GET 342–43
GET# 342–43
GETIN, BASIC's 351, 362
GETIN, Kernal 362, 344, 351, 362
GO   Preface, 333
GOSUB 287, 289, 290–91, 301, 302
GOSUB block 302
GOTO 290–92, 297–98
GO TO 333–34
hanging the system with CHRIN 351
Hesmon   Preface, 23, 38, 160
hierarchy of operators. *See* precedent, operator
highest token 104
Horner's Rule for polynomial evaluation 39, 388–92
I/O channels 89
I/O routines 350–63
IF 297–99
immediate mode. *See* direct mode
implicit LET 183–87
IN message 383
initial operator priority for expressions 135–41
initialize
  BASIC 89
  vectors 91
INPUT 338–41, 343–44
INPUT# 338–41, 344, 345
insertion of line into program text area 109–15
INT 249–50
integer, largest possible 170
integer-to-float conversion 244–48
interpreted language   Preface, 3
interpreted statements   Preface, 4
IRQ vector 41
jiffy clock 154–55
Johnson, J. C. 9
Kernal vectors 89
keyboard 91, 92, 133
keywords 3, 103–109
  dispatch vector table. *See* tables, command dispatch table
  table 103–109
Knuth, Donald Ervin 190, 373
LEFT$ 256–59, 262, 271–73
LEN 270
length of tokenized line 103–104
LET 135, 167, 183–87, 279, 285
line field 110
line numbers 3, 85, 96, 103, 109–11, 115–19, 287–93, 295–97
  largest possible 107, 116, 292, 301, 322
  for LIST 320–23
  to-ASCII-string conversion 323
linefeed 98, 315, 322
lines
  deleting 96–97, 109–15
  relinking 3, 110, 118–19, 354
  sequential storing 110
  storing 3, 96–99, 103–15
link address 85, 110–19, 132, 321–23

linked lines 3
LIST 320–25, 351–52
LOAD 350, 352, 358–60
loading an overlay program 359
LOAD/SAVE parameters 359–60
LOG 189, 403–404
logarithm, natural 403–404
logical expressions 135
logical file number 350, 354–56, 359–60
logical line 3, 371
  maximum size 96, 112
logical operations 21, 35–36
machine language, mixed with BASIC 84–85
main BASIC loop 3, 97
MEMBOT 84, 92
memory
  allocations 120–31
  moves 120–31
MEMTOP 84, 93
MID$ 275–77, 256–58, 272, 333–36
modifying BASIC 41
monadic
  minus 136, 150–52, 224–25
  operators 136, 142–43, 150–52
  plus 136, 150–52
move memory 123–24
moving program text 111–15
negative exponents 163–64, 189–90, 229, 232
NEW 89, 93–94, 132–33, 158, 343
NEXT 289, 305–309
next statement for execution 3
nonrelocatable program tapes 352
NOT 135–37, 150–52, 222–25
numeric expression, convert from float to fixed point integer 243
numeric only functions for DEF 326
ON 280, 286, 288, 296–97
OPEN 350–51, 353–57
OPEN/CLOSE parameters 353–57
operand
  evaluate 150–51
  left 147, 149
  right 149
  stacking 147
optimal coding of variables 168–69
order of a polynomial 388
overflow byte 92
parameter passing 5–18
partial products 208–11
partial quotients 214–15
patch areas 4
PEEK 368–70
PI 105, 142, 150, 323–24
PLOT 315–16, 371
pointer
  BASIC 84
  bottom of active strings 120, 124–25, 129, 160–61
  bottom of BASIC memory 123
  end of last statement executed 294–95
  program text area 84, 93, 118–19, 132–34, 160–61
  resets 132–34
  start of arrays 125, 133, 160–61, 169
  start of free space 115, 124, 160–61
  start of variables 115, 132–33, 160–61
  top of free space 89, 91, 93, 121, 128
  top of memory 160–61
POKE 368–70
polynomials 39–40
  computation 388–92
POS 371
Power-On 89–90
  message 89–90, 93–94
power, raising to a. *See* exponentiation
precedent, operator 135–41, 147
PRINT 311–19, 351–53, 361
  a string 313–14, 316–17
PRINT# 350–53, 360–61
program mode 103, 280, 287
program flow 287–310
program
  deleting from memory 89, 132
  statements 99
  text area 3, 89–93, 97, 110–15, 117–19, 338, 343
  text area initial $00 byte 93, 110
  text area pointer 89, 93, 359
programs, machine language 84, 89
propagating negative sign 240
question mark 103, 105, 318, 340–41
quotation mark 104–107, 256–57, 318
quotes
  around strings for INPUT and READ 339, 345
  effect during LIST 320, 322–23
radian 393
RAM 42
READ 99, 134, 338–39, 347–48
READST 154, 358
READY message 96–97, 387
recursion 145, 147–48
relational operators. *See* comparison, testing for
relocatable program tapes 352
REM 103, 107, 288, 299
  to hold machine language programs 85
  effect during LIST 320
reset 89
RESTORE 132, 134, 339, 343

RESTORE key 89
RETURN 289, 302–303
reverse polish notation 137
RIGHT$ 272–75, 277–78, 334–37
RND 92, 372–74
ROM 42
rounding 21
RS-232 350–51, 363
RUN 3, 157, 160, 279, 287, 289–91
SAVE 352–53, 357, 359–60
  to tape from >= $8000 353
scalar variables 161–62
  area 121
scan for delimiter of statement or line 300–301
scientific notation 227, 229
search for BASIC line 117–18
secondary addresses 350, 352, 354–55, 360
serial devices 133, 351
SETNAM 352, 356, 358–60
SETLFS 350, 352, 354–55, 357–60
SGN 21, 251
sign of floating point accumulator 190–92
SIN 393–97
spaces
  between line numbers 116
  printing 318
  retained in tokenized line 103, 105
  skipped, by CHRGET 99, 102
SPC(311, 316
SQR 21, 224
stack 122
  and for block 303–304
  and GOSUB block 301
  direct access 170–71, 307–10
  direct modification 92, 233
  pointer 89, 91, 94, 115, 122, 133–34
  use of bottom 233
stacked operator
  priority 137, 149
  execution 149–50
start of tokenized program area 90–91
statement
  interpretation 3–4
  execution 284–85
step 303–305
stepsize
  default 42, 305
  modifying default 42
STOP 288, 293–94
STOP key 89, 94, 282, 288, 293, 322
string(s)
  active 257, 259–61
  argument for USR 14
  comparison 256
  concatenation. *See* concatenation
  constants 256
  copy to allocated area 121, 130
  deleting 160, 258, 263
  expressions. *See* expressions, string
  for INPUT and READ 338
  for INPUT prompt 340
  functions, not allowed in DEF 326
  length 263–64
  location in memory 257
  moving to allocated area 128–29
  operations 256–78
  pointer 263–64
  printing 313–14
string descriptor 135, 257
  creating 259–62
  deleting 264–65
  size 91–92, 125–26, 128
  temporary stack 120, 135, 151, 187, 257, 318, 346, 365
  temporary stack pointer 89, 94, 125, 134, 359
STR$ 268
subtraction. *See* floating point, subtraction
syntax check 3
syntax errors. *See* error messages, SYNTAX
SYS 5–12, 375–79
TAB(311
tabbing for commas in PRINT 4, 315–17
tables
  command dispatch 108–109, 285–86
  device number 350, 354
  function dispatch 333–35
  initial vector values 91
  keyword 103, 108–109, 324–25
  logical file number 350, 354
  math, logical, relational operator dispatch 145, 147
  secondary address 350, 354
TAN 393–94, 397–98
tape. *See* cassette
tape buffer. *See* cassette buffer
ten
  FAC1 * ten 30, 212
  FAC1 / ten 30–31, 221
text pointer 96–97, 103, 107, 110, 115, 132–34, 142
THEN 298–99
  following null expression or null direct mode line 283
TI 152–55, 183,
TI$ 99, 101, 154–55, 183, 185–86, 233, 235, 256
time-outs 340–42

tokenization 3, 10, 96–97, 99, 103–109
  vector 41
tokens 3, 10, 103–104, 106, 324–25, 333–35
top of
  BASIC 84, 91, 350, 363
  free space 91
  memory 90, 350, 363
trailing zeroes 236
trig routines 393–400
true/false comparison results 364–67
two's complement 190, 197, 200
type checking 264
types of expressions 135
unary operators. *See* monadic
USR 5–17, 375–79
VAL 256, 269–70, 335
variable descriptor
  array 156–88, 256
  floating point 153
  integer 153–54
  string 256
variables 156–88
  array 174–78
  assigning values to 135
  clearing 133, 290
  creating 156–59, 165–68
  deleting 165–68
  descriptors 256–62, 364, 377
  floating point 163, 185
  integer 159
  locate 165–68
  names 162
vector(s) 41
  detokenization 41, 324
  error routine 385–86
  evaluate operand or handle monadic operator 150
  floating point to integer conversion 91
  initialization 89–95
  integer to floating point conversion 150
  IRQ 41
  main BASIC loop 97
  statement execution 41, 279–309
  tokenization 41, 44, 103–109
VERIFY 358
VIA chips 90, 372
WAIT 369–70
warm start 89, 94–95
wedges 41
where to put machine language programs 84
X-register
  convert floating point to integer in range 0–255 in X-register, 22
  convert X-register and accumulator to floating point to ASCII string 247, 268
  passing to/from SYS 375
Y-register
  add to text pointer 299–301
  convert floating point to integer in Y-register and accumulator 241–43
  convert Y-register to floating point 247
  convert Y-register and accumulator to floating point 246
  passing to/from SYS 375
zero
  testing for during multiplication, division, addition, subtraction 149–50
zero page pointers and variables, initialization 91–93

Building with BASIC

Your computer's BASIC ROM routines, those magical parts of BASIC that actually make it work, may seem like a mystery to you. If you've been programming for any length of time, you know how to use BASIC, but you probably don't know *how* it works inside the computer. That knowledge is powerful—understanding how BASIC's routines actually operate can open up an entire new world of programming.

VIC-20 and Commodore 64 Tool Kit: BASIC is a comprehensive guide to those BASIC ROM routines in the Commodore 64 and VIC-20 computers. Thoroughly documented and clearly written, it shows you how to call the BASIC routines from your own BASIC or machine language programs. But it's not simply a how-to; the *Tool Kit* is also an extensive reference guide to your machine's operating language.

The BASIC rqutines can be powerful, versatile programming tools. This book shows you how to build a complete programming kit with these ready-to-use routines. If you already know machine language, you'll soon be picking the routines you need, avoiding BASIC's interpretation of each statement, and creating machine language programs that do the same things as BASIC (and much more) but that execute far faster. If you're programming in BASIC, you'll see how to call its ROM routines from within your own programs to make them even more powerful.

What's more, by studying the explanations of the BASIC ROM routines, you'll even begin to understand the techniques used to create BASIC itself. A few of the other things included are:

- How to modify BASIC
- Techniques which pass values between BASIC and machine language
- Detailed discussions of BASIC SYS and USR
- Clear step-by-step explanations of the BASIC ROM routines.

The Tool Kit is a significant resource for all programmers. If you're interested in strengthening your programming skills on the VIC-20 or Commodore 64, you'll find this book an essential reference guide that you'll return to again and again.

ISBN 0-942386-32-9

COMPUTE!'s VIC-20 and Commodore 64

Tool Kit: BASIC

COMPUTE! Books